The Virtual Environment Operating System: Derivation, Function and Form.

by

Geoffrey P. Coco

A thesis submitted in partial fulfillment
of the requirements for the degree of

Master of Science in Engineering

University of Washington
1993

Approved by____

Program Authorized
to Offer Degree____

Date____

University of Washington

Abstract

The Virtual Environment Operating System: Derivation, Function and Form.

by Geoffrey P. Coco

Chairperson of the Supervisory Committee: Professor Steven L. Tanimoto
Department of Computer Science and Engineering

The computation of virtual environments poses a host of challenges for modern computing technology. These challenges include real-time 3D binocular rendering, voice input and synthesis, body tracking, and sensible interface design. Each of these is a deep research area unto itself and many specialists have already demonstrated independent working solutions.

Another set of problems associated with virtual environment computation are more general in nature. 1) Most VE applications require simultaneous access to diverse resources (i.e. technologies surrounding the aforementioned capabilities) which are often available only from heterogeneous platforms. 2) Often data is disparate in cases of telecommunications and other interactive on-line applications. 3) Virtual environments can demand significant computational resources for each user, and the potential benefit of parallel computing is great. Each of these problems are addressed by a distributed computing model.

This paper presents VEOS, a highly portable programming facility which utilizes common Unix services for distributed and coarse-grain parallel programs. VEOS was designed for fast prototyping of distributed virtual environment applications across heterogeneous workstation clusters. The VEOS programming model emphasizes asynchronous communication and distribution based on *entities*, a mechanism for non-preemptive task decomposition.

Contents

List of Figures

List of Tables

Glossary

- 6D (Six Degrees of Freedom) - the ideal data format given by simple spatial tracking devices regardless of the tracking technology. A 6D is composed of three axes of Cartesian position (x,y,z) and three axes of orientation (roll, pitch, yaw).
- Distributed Shared Memory (DSM) - a single address space that is (conceptually) shared between nodes in a distributed system. DSM usually requires explicit hardware and/or operating system low level support such as cache coherence, message passing, and process synchronization.
- Distributed Virtual Memory (DVM) - like DSM, except that DVM incorporates the additional benefits of virtual memory - large address spaces (larger than physical memory) and automatic paging (possibly across the network).
- Greatest Common Denominator - a guiding principle during VEOS development which supported our goals toward portability and ease of use. The principle states loosely that when all instances of VEOS can equally support a feature or capability, the feature is above the GCD and can be incorporated into the system. But when a feature or capability is only available in certain cases (for example, shared-memory services from Unix), the system architecture cannot rely on the feature.
- Human Interface Technology Laboratory (HITL) - the research lab under the Washington Technology Center at the University of Washington where the research summarized in this paper was conducted.
- Immersive or Inclusive - describes an information environment that perceptually surrounds the participant and addresses the participant's sense of presence.
- Intention - a style of process communication where the instigator of the communication actively influences the workings of the receiving process. Computationally equivalent to Interpretation (below), but the distinction is made for reasoning about process interactions.
- Interpretation - a style of process communication where the receiver of a communication actively alters it's own workings because of information contained in the message. Computationally equivalent to Intention (above), but the distinction is made for reasoning about process interactions.
- MIDI - the Musical Instrument Digital Interface standard allows computers, musical keyboards, samplers, and other electronic devices to exchange discretized music information. Basic concepts of MIDI are triggering samples with specific pitch, velocity, and channel. MIDI channels are a mechanism for modularizing the digital data stream from actual sounds that are heard.
- MIMD - the Multiple Instruction, Multiple Data multiprocessing paradigm supports heavyweight (compared to SIMD) parallelism via independent threads of control.
- Participant - a virtual reality *user*. Emphasizes that there is significant interaction between the user and the virtual environment as well as other human participants.

- Presence - the degree to which the participant feels as though they are really in the simulated environment. A VE which manifests many critical perceptual cues will offer a strong *sense of presence*.
- Primitive - a built-in system utility that performs some common or fundamental task. In addition, primitives, ideally, are composable with one another. For example, in arithmetic the primitive operators are: +, -, *, /. In Lisp, cond, car, cdr, cons, eval, lambda, etc. are primitives. And, in a word processor, Insert, Cut, Copy, and Paste.
- Rapid Prototyping - another guiding principle of VEOS. Refers to building sufficient working applications with a minimal investment of time, effort and skill. This attribute is desirable when there are many ideas to implement while resources are limited, and when ideas and infrastructure are expected to keep evolving steadily.
- Tracking - the problem of accurately monitoring the participant's position and movements. Tracking is further complicated by application specific demands of accuracy, comfort, range of sensitivity, and long-term human exposure. Common tracking solutions exploit electromagnetic induction, ultrasound w/triangulation, inertia differential, mechanical linkage, image processing, and optical w/triangulation.
- Tuple - in its most generic form, a generic data type, similar to a list, which may contain any number of heterogeneous data elements.
- Virtual Environment - less popularized term preferred in academia, referring primarily to the objective quantifiable experience.
- Virtual Reality - used under many definitions. For this paper it shall refer to the technology and culture associated with computer simulations designed to enhance sense of presence, whether it be accomplished by high-resolution displays, inclusion, high interactivity, soothing voice feedback, etc.
- Wand - broadly refers to a common VR interface (hardware and software) by which the participant can perform most actions in the virtual environment. Like it's cousin, the mouse-based pointer, the wand allows its operator to evoke actions through direct manipulation, to make commands with context-sensitive movements, and to rely more on recognition of choices and less on recollection of utterances. The wand differs from previous devices such as the mouse or joystick because when combined with a 3D display it can provide a full representation of our natural movements - six degrees of freedom.
- Workstation - general purpose computers falling somewhere between minicomputers and personal computers in price and performance. Typically workstations support a form of Unix (providing networking and multitasking) upon which many GUI interfaces are available.

Abbreviations

6D - Six Degrees of Freedom.

DSM - Distributed Shared Memory.

DVM - Distributed Virtual Memory.

GCD - Greatest Common Denominator.

GUI - Graphical User Interface.

HITL - The Human Interface Technology Laboratory.

Mac - Macintosh Personal Computer.

MIDI - The Musical Instrument Digital Interface standard.

MIMD - the Multiple Instruction, Multiple Data multiprocessing paradigm.

PC - IBM-compatible Personal Computer.

VE - Virtual Environment.

VR - Virtual Reality.

Preface

In the summer of 1990 when I arrived in Seattle to work on the VEOS project, little did I know that there was no real VEOS, only the concept. At the time, the still small HITLab was composed mostly of visionaries and administrators - no software engineers. I was part of a pilot program for summer interns. It was a good chance for students and other youthful contributors to test their affinity for VR and the HITLab. Meanwhile, the HITLab was hoping for some cost-effective contributions from these interns.

I quickly gravitated to the role in which I still reside today - that of the pragmatic and resourceful software engineer preferring to work on ambitious yet achievable projects using available resources. It was then that I began writing the first beginnings of the VEOS with the assistance of another summer intern and veteran Unix hacker, Dav Lion. Primarily, I wrote fundamental operations code, while Dav worked on graphics and applications.

The VEOS ideals, since it's inception, have been put forth by William Bricken, HITL's principal scientist. Although these ideals are often fleet and difficult to express in words, they were the guiding stars as we wandered into implementation. At the highest abstraction the ideals are: *simplicity*, as in a small set of clear fundamentals or primitives, *integration of concept* such that all relating primitives are composable, and *universality of function*, where the set of possible combinations of these primitives is a broad and expressible program space.

There have been two major interpretations of the VEOS ideals. The first is embodied in the VEOS as it stands today and will be covered in this paper. The second is MOSES (the Meta Operating System and Entity Shell) which stands as an interface specification [MOSES]. The MOSES design embodies particular foresight in networking and scalability, but remains untested.

Students came and went for the next two and a half years, each making contributions to the VEOS direction, while I cultivated and maintained the VEOS implementation throughout. Since the first day that VEOS ran in October 1990, countless experimental applications have driven VEOS's development in varying directions.

Hence, VEOS stands as the product of many gradual and catastrophic changes. Now incorporating components developed by many other engineers at HITL, VEOS is well worthy of the title, VEOS 3.0.

This paper details our goals for VEOS from the outset to the present as well as the constraints which influenced the design and implementation. The discourse is written from an interdisciplinary viewpoint, stressing aspects of VEOS both quantifiable (such as performance and features) and subjective (such as simplicity and ease of reasoning).

Acknowledgments

I wish to thank to those with burning creative energies that fueled my development of their answer, Colin Bricken, Ari Hollander, Dan Pirone, Random Reay, and Craig Rosenberg.

Thanks to those who provided the true interface capabilities for anything to reach the physical world, Mark Cygnus, Brian Karr, Dav Lion, Andy MacDonald, Max Minkoff, Jesus Savage, and Fran Taylor.

And those who provided valuable input during the VEOS development, Michael Almquist, Johan Anderson, Julian Bleecker, Alan Borning, Colin Bricken, Meredith Bricken, Marc Cygnus, Charles Grimsdale, Ari Hollander, Jeff James, Brian Karr, Hank Levy, Dav Lion, Andy MacDonald, Max Minkoff, Dan Pezely, Dan Pirone, Jerry Prothero, Random Reay, Dan Shapiro, Chris Shaw, and Suzanne Weghorst.

To my advisors, Steve Tanimoto, Tom Furness, and Ed Lazaowska from whom I absorb wisdom and knowledge by merely being in their presence.

Above all, I bid an everlasting thanks to William Bricken for his continual support and guidance.

Chapter 1: Introduction

HITL

The Washington Technology Center (WTC) at The University of Washington acts as a conduit for innovation by carrying emerging technology from academic research to local industry. WTC research groups at the University of Washington enjoy support from companies that hope to profit from new approaches, designs and techniques. The Human Interface Technology Laboratory (HITL) is a WTC research group that focuses on virtual reality technologies.

Virtual reality immerses the participant into an information environment where natural behavior is the interaction paradigm. "Virtual reality is the direct experience of a digital environment" [VEOS]. HITL develops virtual environments designed to accelerate learning, facilitate rapid information assimilation and manipulation, extend creative capabilities, enhance communications, and aid the disabled.

Like many of the academic VR labs now forming nationwide, HITL is working on basic research. Some of HITL's primary development areas are: 3D binocular imaging hardware, 3D binaural sound, virtual reality software systems, 3D position tracking, and human performance and perception study. HITL hopes to transfer advances to areas such as: design and manufacturing, telecommunication, medicine and prostheses, entertainment, and education.

Virtual Environments at HITL

HITL Research Patterns

One of the goals of virtual reality technologies is to empower more people with computers. Just as the advent of the GUI brought computers closer to people, virtual reality stands to bring computing even closer to home with it's attention to natural interaction and issues of immersion. But since there will be a significant wait before dedicated VR technology is readily available, current research at HITL involves bootstrapping to conventional computing technology using conventional programming techniques to pave the way for more integrated solutions of the future.

Research at HITL is carried out by staff researchers, adjunct engineering faculty, industrial fellows, graduate students, and student interns. Many of these contributors work with HITL only a short time, ranging from a few months for industrial fellows to 2 years for graduate students. Research time is even tighter than that. As adequate VR interface and computing resources are sparse, running a VE inevitably incurs resource conflicts and bottlenecks. These constraints coupled with the implicit mandate for simplified access to evolving VR technology form the fundamental need for a rapid prototyping system.

Continued funding for research in the VR community is often contingent on proof of progress. Giving demonstrations of new ideas and components is a regular activity and further demands a system for rapid prototyping of virtual environments.

Neophytes to formal specification, such as designers, experimenters, and artists characterize many VE builders at HITL. These people are usually computer users but are not necessarily computer programmers. The VE building population demands an interface which caters to rapid prototyping, allows modular reuse of parts, and is simple and easy to use. At the same time, these users can be the most demanding, often trying things a system architect never expected. The interface must also be sufficiently general and provide the computational flexibility for many ways of thinking and many application types.

Commitment to LANs

At HITL, virtual reality is a distributed computing problem for many reasons. Information is inherently disparate. Whether it be financial market data updates, movement data from a co-participant, or sensor data from the real world, data is rarely centralized. Many crucial goals are easily attainable under a distributed paradigm using workstation clusters. To name a few: support for multi-participant interactions, heterogeneous resource topologies, and coarse-grain parallelism, all while allowing simultaneous internet activity.

Today's workstation clusters rely on easily reconfigurable Ethernet. Conveniently, the same network technology can be used over the internet for wider area networking. As the primary VE computing resource, HITL chose to develop for clusters of commodity workstations. Most workstations come Unix-ready, providing reliable multi-user and networking services. Networking these workstations is a cost-effective way to incrementally build sufficient computing resources for multiparticipant VEs.[1]

There are also a growing number of VR research labs that also work with workstation clusters. By supporting this model of computation, HITL hopes to provide an effective software foundation for VE development that others can use.

Computation Components of VR

Each VE application demands different resources and program complexity, but there are some common concerns across applications. Usually, several i/o streams to and from the participant are necessary. These data streams incur considerable overhead in terms of processor cycles, physical i/o ports, and auxiliary hardware expense.[2] The rate and accuracy of the i/o streams are gauged by the application's objectives, resource constraints, and empirically determined perception tolerances. Other usual computational tasks are: interaction, navigation, physical simulation, solid objects (collision maintenance), dynamics and constraints, data modeling and visualization, autonomous agents, and communications.

Interaction refers to the mappings, commands, languages and feedback cues that define the i/o streams to and from the participant. This could also include measurement and statistics computation for human performance study. This task is most directly concerned with human interface issues.

Navigation comes into play when the VE employs a strong spatial metaphor. The 'space' is larger than the participant's ability to see and interact with it all. One natural compliment to this potentially huge virtual real estate is spatial navigation. Successful navigation interfaces have included multiple speeds, varying flight semantics, and navigation through scale.

---

[1] These resources include compute cycles, i/o ports, graphics capabilities, memory, hard-disk, etc.

[2] Additional hardware includes graphics pipelines, interface peripherals, etc.

Physical simulation is the disciplined Newtonian interpretation of motion and object interaction. This is a broad term that applies whenever finely deterministic simulation techniques are desired over perception-based methodologies.

From an engineering point of view, solid object computation is simply a component of physical simulation. From a design point of view there are situations when physical simulation techniques oversolve the specific VE objective, but the interface itself may rely on accurate collision detection. Solid object computation is a primitive capability independent of full physical simulation.

Another category of computation tasks that crosses into other categories is dynamics and constraints. Interaction can specified with dynamic or constraint-based representation [ThingLab], but there are languages clearly better for human interaction, and languages better for object interaction. Dynamics and constraints may solve both problems, [VPL] [VEOS].

Data modeling and manipulation applies to applications that interface the participant with data streams, real-time or static sets for compelling visualization. Moreover, these applications allow the participant to interact with computer generated abstractions and in so doing make actual changes in the data or data streams. Areas where these techniques apply are CAD, financial visualization, architecture, training, libraries, and databases.

Many applications also contain autonomous computation. Autonomy provides interactive actors for instructional purposes and overall richness to a VE. Autonomous computation can be partially decoupled from the continual negotiations with the participant, creating an application for parallel computing.

Communications is involved in multiparticipant applications such as in cooperative tasks, competitive scenarios, telecommunications, etc. Communications is a category of computation that relies heavily on hardware specifics, in this case network technologies.

Project Goals and Constraints

Rapid Prototyping

Rapid prototyping was a fundamental design consideration at every step of VEOS development. The ability to quickly and easily build and reuse VEs is invaluable in an environment where neophyte programmers face a strong demand for new applications while ideas, infrastructure, and resource configurations keep changing.

From an engineering standpoint, rapid prototyping suggests attention to modularity, generality, and simplicity. For example, since most VE applications require i/o streams to and from the participant, modular drivers are built to produce standard data streams that can be 'plugged' into application-specific computation. Similarly, other components of an application are written as parameterized modular units.

Distributed Computing

Distribution achieves several goals at once. Distributed computing serves to bridge disparate participants, data, and devices. Distributed computing also provides a foundation for coarse-grain parallelism across loosely-connected multicomputers.

Uniprocessor LANs are a common choice for flexible and cost-effective computing. The workstation nodes typically run a version of Unix and support common Unix services such as reliable networking, virtual memory, and multiprogramming.

Heterogeneous Platforms

It is often advantageous for software to run on many platforms, especially for VEOS. Different platforms offer different strengths, specialties, and costs. Utilizing the capabilities of many different platforms simultaneously is desirable and reiterates a goal of distributed computing.[3] Unlike tightly coupled multiprocessors, workstation clusters provide flexible connectivity and can tolerate heterogeneous processing elements.

Generic Compute Model

---

[3] Although the basic system does not take special advantage of different platforms, system-specific modules are encouraged in order that specific capabilities can be exploited by applications such as real-time 3D graphics, or voice recognition.

Although VEOS was envisioned for VE computation, that is a very broad range of possible applications. One could imagine VR applied to almost any conventional computer application. Thus, the compute model should reflect practical programming concepts that can be used in many ways to accomplish many unknown problems of the future.

The ideals for the VEOS compute model are that it be complete or universal, that it be simple, and that as much as possible it be decoupled from the physical particularities of computation.[4] This is accomplished through a strong task decomposition model. A few paradigms which embody a generic compute model are object-oriented systems[5], Linda systems, and Mathematica.

Free Software

Initially, it was held to be a high objective that the system be unproprietary and freely distributable. This meant not integrating with any production software to avoid legal tangles. As such, VEOS was written from scratch using many borrowed methods and techniques in order to attain an unproprietary implementation. This strategy has often been confused with the 'not-invented-here' syndrome where a group tends to prefer in-house ideas and implementations to borrowed or collaborative work.

Portability

Portability is always an advantage for software, especially for VEOS. In the hope that VEOS can provide an infrastructure for other research labs now forming, VEOS remains dogmatically portable. More practically, since support for heterogeneous platforms was already a goal, attention to portability follows naturally.

From an interface perspective, it was hoped that portability could come at no cost to the interface. The interface could be consistent across differing platforms.

Portability is the primary basis for the greatest common denominator principle (GCD). This states that the general system remain hardware independent and that generic

[4] such as location of computation, native instruction set, cpu throughput rate, details of i/o handling, etc.

computing semantics remain the same over all types of computing nodes. Practically, portability means relying only on the most common operating system services.[6] Platform independence also discourages user-level context switching since it requires hardware specificity. Finally, portability suggests assuming only uniprocessor nodes, using only a single process per processor.[7]

---

[5] Flavors of object-oriented systems include agents and actors.

[6] In this case, Unix services.

[7] Assuming uniprocessors does not preclude using VEOS on multiprocessors, see Chapter 7.

Chapter 2: Related Work

The development endeavors of others have provided a tremendous foundation upon which to work with the problem of virtual environment computing. However, since the discipline at hand (i.e. virtual reality) is still quite young, most previous research only sheds light on a piece or two of the puzzle. The problem of (distributed) VE computation is multi-faceted, and warrants looking at approaches to many interrelated areas.

Since VEOS was planned for LANs of commodity workstations, HITL's choices of computing hardware were simply a matter of preference and availability. Far less obvious and more at the heart of this work was the form that the coordination software would take. Hence, this section is devoted to relevant software technologies.

Interface Components

Great strides have been made in recent years toward technology designed for high-bandwidth human interaction. Binocular display, 6D position sensing, fine manual sensing, voice recognition and generation, and musical input (MIDI). The combination of these advances define the bottom-line capabilities of a potential virtual environment. For purposes of discussion regarding VEOS, these interface technologies will often be referred to as device drivers or i/o modules, and are considered black boxes that support specific capabilities. They are mentioned here because without such diverse and complimentary developments, VR coordination software such as VEOS would have no value.

VR Building Systems

There is great confidence that VR can be useful in many application domains. But as of yet, relatively few turnkey VR systems have arisen and of those, most have been entertainment applications.   Notable examples are LucasArt's Habitat™, W Industries Game Parlors, Battletech Video Arcades, and Network Spector™ for home computers. Virtus Walkthrough™ is one of the first successful commercial turnkey VR systems for home and office computers. Virtus can be used for viewing CAD files more intuitively with natural navigation and viewpoint control.

How and to what extent VR can be utilized by existing areas is yet to be learned. Perhaps it is for this reason that predominant VR software development has been in VR *building* tools, with which people can experiment and understand how to integrate the benefits of VE technology. These tools usually offer as much flexibility as possible within certain well-defined constraints. Often VR building tools emphasize some particular design aspect based on the original impetus for developing the tool.

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

VR Kits

MR is a tool kit for building VEs and other 3D user interfaces developed by academic researchers at the University of Alberta. The tool kit takes the form of subroutine libraries which provide common VR services such as tracking, geometry management, process and data distribution, performance analysis, and interaction paradigms. The MR Tool kit fits many goals of the VEOS design such as modularity, portability and some support for distribution. However, MR does not strongly emphasize multi-participant applications or rapid prototyping (MR programmers use C, C++, and FORTRAN). MR was developed contemporaneously with VEOS, and so was not considered as a potential implementation.

Researchers at the University of North Carolina at Chapel Hill have created a similar toolbox called VLib. VLib is a suite of libraries that handle tracking, rigid geometry transformations, and 3D rendering. VLib is another programmer kit requiring C or C++ programming, placing it outside the scope of high level interfaces.

A young British company named Division, manufactures VR stations and software. Division's ProVision™ VR station is based on a transputer ring and through the aid of remote PC controller runs dVS, a director/actors process model. The station's capabilities are augmented by real-time 3D rendering hardware and the standard suite of interface hardware (eyephones, head-tracking, and wand). ProVision's internal architecture is designed to scale for more functionality. Stations are used one each per participant and can be networked for multiparticipant VEs. Division developed the dVS software system alongside their transputer technology. Although the dVS model of process and data distribution is a strong design for transputers, it is not evident that the same issues apply to workstation LANs, the target for the VEOS project. Moreover, VEOS was to remain unproprietary and, aside from unavoidable device dependencies, platform independent.

Sense8 a small company based in Northern California, produces an extensive software library called WorldToolKit™ which can be purchased with 3D rendering and texture acceleration hardware. This library runs on PCs, Macintoshes, and Silicon Graphics and supplies functions for interaction, data modeling, and navigation.

Silicon Graphics, an industry leader in high-end 3D graphics hardware manufacturing, has recently released the *Performer* library which augments GL[8] designed specifically for interactive graphics and VR applications for Silicon Graphics platforms.

As [State] points out, much current work is focused on particular problems of VR, be they tools, drivers, renderers, interface paradigms, or other specific capabilities. These efforts are valuable, and they define the various specialties of which VEOS aims to coordinate.

Integrated Systems

At the outset of the VEOS project, VPL Research, Inc. manufactured RB2™, perhaps the only commercially available integrated VR solution.[9] At the time, RB2 supported a composite software suite which coordinated 3D modeling on a Macintosh, real-time binocular image generation by two Silicon Graphics workstations, head and hand tracking by proprietary devices, dynamics and interaction on the Macintosh, and runtime communication over Ethernet.

Although there were many begrudged downfalls of the VPL system, its pioneering attempt paved the way for subsequent, more integrated VR systems to follow. As applied to the VEOS project, the VPL system offered some insight. Most important was the VR

[8] Graphics language supported by SG, quickly becoming industry standard.

[9] VPL, Inc has since undergone business restructuring (Q1, 1993).

design experience HITL researchers gained while using the VPL system. This knowledge helped warn VEOS architects of VE building issues that were not otherwise self-evident.

Autodesk, a leading manufacturer of CAD software, has been developing an integrated VR developing environment that fits tightly with existing Autodesk products [State]. As with all commercial VR products, it's application to VEOS is limited because of it's commercial costs and proprietary technologies.

This is in no way a complete list of current VR software efforts. [State] and [ArchVR] provide more comprehensive overviews. Rather, it is to acknowledge that while many specific areas are being investigated, very few aim to integrate the loose technologies into a complete VE building program. Furthermore, most other VR work is either proprietary or began after VEOS development began, and so lent little to the project.

Linda Systems

[Coord] defines a *coordination language* as a language that augments existing sequential programming languages for building multiprograms. Linda systems are a class of coordination languages. Linda implementations can be used in conjunction with many other sequential programming languages as a mechanism for interprocess communication and generic task decomposition [Tuplex] [Linda] [Tuple]. Linda uses the *tuple* as the basic unit of communication and process.

Independent sequential processes in a Linda program access a location transparent shared *tuplespace* with semantics similar to those of a shared virtual memory architecture. Processes perform *in* and *out* operations to read and write tuples to the shared tuplespace. This tuple communication mechanism is further streamlined by *association matching*. When a process performs an *in* operation, it can block, waiting for a particular kind of tuple (specified by a key) to arrive in the tuplespace. When a matching tuple has been posted, the waiting process receives the tuple which may contain additional data or parameters. On the other end of communication, processes may perform simple *out* operations to post tuples to the tuplespace. Or, processes may fork a new process, called an *eval*, that computes a discrete function which, upon completion, generates and posts a new tuple to the tuplespace.

The Linda coordination model is particularly well suited for distributed computing environments where inter-node communication is relatively costly and only coarse-grain interprocess communication and parallelism are practical [NetLinda]. Tuple-based distribution paradigms appear advantageous from a programming standpoint. 1) the semantics of a shared tuplespace are tolerant of topology changes, 2) properly implemented, tuple-systems assure inherent overlap of computation with communication, 3) it is arguably easier to reason about several sequential programs than one monolithic parallel program, and 4) communication semantics are greatly simplified by abstract primitives.

A close comparison can be made between tuple-based coordination languages and conventional approaches using distributed processes accessing shared virtual memory, or DVM. The differences are clear in their origin and usage philosophy. DVM systems arose out of the operating systems community where innovations were sought at a very low level, namely in virtual memory and threads. Tuple-based systems arose from the AI and inference community where the advances came at the application level, namely in rule-based architectures. Based on the author's experience with implementations of both types, the tradeoff is characterized by the efficiency and well-known usage idioms under DVM, versus the simplicity and ease of reasoning with tuple-architectures.

From the outset, VEOS architects chose not to specifically embrace the Linda model, but instead designed a generic foundation upon which one could implement many abstract communication paradigms including a shared tuplespace.

Object Systems

With object-oriented systems staking a firm claim in modern programming methodology, VEOS architects also followed this precedent.

Smalltalk

For many, Smalltalk represents the canonical object-oriented language. For the VEOS project, Smalltalk represents the strict ideals of an object-oriented paradigm. In Smalltalk, all data and process is discretized into objects. All parameter passing and transfer of control is done through messages and methods. The VEOS architects incorporated the Smalltalk ideals of modular process and data and hierarchical code derivation (classes), but chose not to enforce the object-oriented metaphor throughout every aspect of the VEOS programming environment.

Emerald

The Emerald system demonstrated that a distributed object system is practical and can achieve good performance [Emerald]. It did so through object mobility and compiler support for tight integration of the runtime model with the programming language. Emerald implemented intelligent system features like location-transparent object communication and automatic object movement for communication or load optimization. Simultaneously, Emerald permits programmer knowledge of object location for fine-tuning applications.

The constraints of VEOS project were unique enough to warrant a tailored implementation, but Emerald provided a strong example to follow. The Emerald work 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 object-like units called entities. In keeping with the VEOS ideals of platform independence, VEOS architects steered away from some Emerald features such as a compiler and tight integration with the network technology. In contrast, VEOS required an interface language with simple semantics that supported runtime code generation and evaluation.

Eden

Eden, another distributed object system, provided both coarse and fine grain object definitions for different application needs. Coarse grain objects corresponded roughly to address spaces or protection domains. Fine grain objects were lightweight constructs that shared the same address space, but had considerable performance advantages for inter-object communications.

Rewrite Systems

For purposes of this discussion, rewrite systems shall roughly 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 over a rule-based or equational representation.

Declarative languages such as FP, Prolog, Lambda Calculus, Mathematica, and constraint-based languages operate on the principle of beginning from a consequent and, in reverse, traversing the logical tree of declared antecedents. These languages each display the same trademark attribute - that their control structure is *implicit* in the structure of a program's logical dependencies.

Expert systems employ a match/substitute/execute inference regime over large rule-bases to propose solutions to complex domain-specific problems.

Similar to expert systems, blackboard systems use inference and inference control to solve domain-specific problems. Similar to Linda systems and their tuplespace, blackboard systems coordinate cooperating processes through a shared *blackboard*. However, the crux of blackboard systems is the incremental solving techniques that are applied to the common blackboard while independent knowledge sources generate new problems.

All these rewrite systems have an inherent similarity. That is, knowledge equals program, and is represented by rules or equations. Many rewrite systems infer rules only in one direction.[10] As a result, rules are often seen conceptually as consequents (or head of the rule) and antecedents (or tail of the rule). Classical goal-directed inference is the process of selecting a consequent, trying to achieve the antecedents, and binding antecedents to partial solutions, thus approaching a complete solution. In true equational form, either side of a rule can be substituted for the other. Of course, two-way reduction raises difficult implementation issues [Constraint].

Since so many interesting and powerful forms of inference have been designed and studied for rule-based systems, the VEOS architects sought an implementation that allowed experimentation with inference and meta-inference control structure.

[10] such as FP, Prolog, Lambda-calculus, and some constraint-languages.

Furthermore, a disciplined knowledge representation would surely be compatible with a general tuplespace implementation.[11]

Languages

Many of the systems already discussed were complete systems for performing complex computational tasks which provided both an implementation and, in most cases, a novel interface language. This section focuses primarily on language aspects which influenced VEOS.

Prolog

Prolog, the definitive declarative programming language, roughly manifests the ideal of inductive reasoning. Programs in Prolog consist of a batch of interrelated rules, which the Prolog interpreter (solver) scans to 'prove' the preconditions of an end goal. The solver iterates this process and in doing so descends recursively into the solution space, bringing back variable bindings and satisfied constraints. The decision path of the solver represents the only formal mechanism for program flow control in Prolog. Flow control can be tailored through ordering of preconditions and through the *cut* operator.

Mathematica

In addition to it's extensive built-in library of composable math primitives, Mathematica provides a declarative equational language, which, like Prolog, depends on the fundamentals mechanics of algebra. Mathematica, however, offers significantly more flexibility in controlling program flow and limiting the search space.

Since Prolog and Mathematica are in principle both rewrite languages, languages akin to them could be constructed on top of a generic rule database. This possibility reinforced the affinity between VEOS and a generic tuple architecture.

Lisp

Lisp, the time-honored prototyping language of choice, invites programmers of all levels of skill and experience. Lisp encourages prototyping partly because Lisp's

[11] e.g. [head]tail -or- lhs=rhs.

interpreted nature, which makes it quite easy to modify a working program in place 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 it's code-data equivalence. In other words, program fragments can be manipulated as data and data can be interpreted as program.[12]

In terms of availability, Lisp has been implemented in every imaginable context. 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 a public domain interpreter [XLISP]. Upon close inspection, the XLISP implementation is finely-tuned, fully extendible, and fanatically portable.

Linda

As detailed above, Linda is an extension to existing sequential languages for abstract inter-process communication. As a coordination language, the Linda model presupposes multiple processes. Whether via preemptive multitasking on a uniprocessor, or parallel processing on distributed or multi-processors, processes in a Linda program access a shared tuplespace and thus a common workspace.

Again, the potential is clear for applying Linda semantics to a generic tuple-architecture.

Smalltalk

Smalltalk is another language that demonstrated that a simple metaphor can provide a lot of expressibility. In Smalltalk's case, the metaphor is the *object*. Everything in Smalltalk is an object, including constant integers themselves. Smalltalk stands as the canonical object-oriented language whose object-oriented semantics and mechanisms VEOS borrows from.

---

[12] As data, these structures can be stored in the database or passed as messages; as code, they can be used in genetic algorithms or as active messages.

Network-based Multicomputers

The benefits of parallel computing are apparent, but the economic costs of parallel computing resources often outweigh the benefits. The need for more available parallel computing accompanies a growing trend toward component parallel processing systems such as workstation clusters. The progression from [Spector] to [Ivy] to [Nectar] has demonstrated a strong push toward this end. VEOS architects chose to deemphasize short-term performance issues trusting that network-based systems would continue to improve and choosing instead to focus on conceptual issues of semantics and protocols.

Distributed Shared Memory

The operating systems community has devoted great effort toward providing seamless extensions for distribution to virtual memory and multiprocess shared memory. These developments have been proven feasible and have spurred further development in hardware support for distributed 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 VEOS's needs, many of the same issues resurface at the application level.[13]

Ivy

Ivy 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. This tendency compensates for lost performance due to disk latency. In Ivy, locality of reference compensates for network latency as well. Furthermore, the increase in total physical memory in an Ivy program (due to more nodes), sometimes afforded a superlinear speedup over sequential execution.

[13] In particular, coherence protocols.

Second, Ivy demonstrated the performance and semantic implications of various memory coherence schemes. These coherence protocols, such as *release consistency*, are applicable to many domains beyond DVM, in particular distributed tuplespace implementations.

Munin and Midway

Munin and Midway represented 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 [Munin]. 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. All these protocols, including entry consistency, when used strictly, yield correct deterministic behavior [Midway].

These systems demonstrated an attention to experimentation. By providing language extensions for parameterizing how the system-level shared memory was implemented, application programmers were empowered to modify the runtime constraints in order to achieve maximum performance or correctness. For the VEOS project, these systems reinforced the commitment to flexibility and experimental services.

Process Models

Threads

Threads, in this context, shall mean cooperating tasks each specified by a sequential program. Furthermore, threads can be implemented at user level and often share single address spaces for better data sharing semantics and 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.

This generic process capability is widely used and has been thoroughly studied and optimized. However, threads implementations normally have system dependencies such as the assembly set of the host cpu, and the operating system kernel interface. This inherent platform specificity combined with the admission that generic threads may be too strong a mechanism for VEOS requirements kept VEOS architects looking for another process model.

Chores

Chores is a novel approach to parallel computing that takes advantage of highly decomposable task domains. In Chores, *workers* take work from the problem *heap* and incrementally reduce the problem in parallel. Workers take on more work with attention to running load averages in order to maximize parallelism.

The Chores perspective of process decomposition was refreshing and stimulated further thought toward VEOS tasks.

Cyclic Executive

In many application domains, including all forms of signal processing, the overall problem can be represented 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. It is easy to see why the cyclic executive is the de facto process model for many small real-time systems [Shaw].

This model was taken very seriously by VEOS architects for two reasons. It provided a process model that is implementable in a single process, making it highly general and portable. Moreover, it directly addressed the cyclic and repetitious nature of the majority of VE computation. This cyclic concept, known as *frames*, will be discussed thoroughly.

Smart Ideas

Active Messages

The concept behind active messages is using a network data format that is immediately evaluable thus reducing wasted lookups and protocol stacks [Active]. VEOS architects took this concept to heart when considering to use Lisp as the interface language. Lisp supports program/data equivalence, which suggests a perfect application of the active message paradigm. Lisp expressions can be linearized and passed as messages across address space boundaries and then evaluated on the other side by an awaiting Lisp interpreter.

Chapter 3: The VEOS Approach: Simplicity and Hybridization

As discussed earlier, the VEOS design had to address numerous and sometimes conflicting demands. The result is a broad, general, and baseline design. In other words, in the face of so many goals and constraints, the most prudent design was a simple one that would support the most possible purposes. This line of reasoning has become known among VEOS architects as the greatest common denominator principal and succinctly represents many of the VEOS goals.

The GCD principle derives the following two caveats. First, GCD leads to small libraries of fully-generalized services. With a small system, there is less code to maintain, it is easier to change major underpinnings, and being simpler, the programmer's learning cycle is shorter. Second, GCD suggests a strong distinction between the application and the system layers.[14] In other words, if a service is required by most every application, it is a good candidate to become a system service and not otherwise. This second aspect of GCD encourages resourceful utilization of common services and in many cases leads to innovation.

In system design, the elegance of purity comes at considerable risk. Just as purebred hounds are more vulnerable to certain diseases, and as dictatorships are more susceptible to unrest, pure languages systems are susceptible to traps and pitfalls in many problems domains. These traps and pitfalls usually manifest as problems with performance or expressibility. Since VR is such a young area of study and the defining characteristics of typical applications are yet unstudied, prudence encouraged a diverse design incorporating multiple methodologies. Again, this attention to hybridization was an attempt to provide multiple solutions to unforeseen applications.

The remainder of this chapter discusses the various methodologies that were chosen and how they were hybridized to address all the primary goals and constraints. The next chapter details how these methodologies culminated into a cohesive implementation.

[14] although both may run at so-called Unix user-level.

Computing Platform

Among the goals and constraints presented above, perhaps the most restrictive is the portability constraint. Roughly speaking, following the GCD principle ensures portability. Although VEOS would initially run on Unix workstation clusters, there has been a strong interest in running VEOS on PC and Macintosh platforms in the future. This context helped to define the extent to which GCD would apply to the VEOS design, and in particular determined three premises.

1) To assume only uniprocessor network nodes.

Although multiprocessors are quickly gaining acceptance and popularity, uniprocessors still typify the most common node in workstation clusters. Although this assumption reflects the most common case of uniprocessors, it does not preclude using VEOS on a multiprocessor. Chapter 7 discusses several approaches for using VEOS on a multiprocessor.

2) To assume only one process per node.

Although many standard operating systems have incorporated support for multiprogramming, their approaches and interfaces vary widely making it difficult to implement multiprocess applications that are portable.[15] Furthermore, it was doubtful that VEOS required fully general process constructs with separate address spaces, memory protection, and sequential program semantics, such as those associated with traditional Unix processes.

Thus, VEOS assumes only a single sequential process per node using a simple process model.[16] With this premise, the VEOS architects aimed to achieve portability, efficiency, and perhaps an intuitive mapping to the tasks of VE applications. The single process assumption still permits that when using VEOS in a multitasking environment

[15] Multiprogramming services range from pre-emptive processes in Unix, to cooperative multi-tasking in Macintosh, to the still immature multiprogramming on PCs.

[16] i.e. no platform-specific multitasking.

such as Unix, other users can simultaneously use the workstations for separate work with uncompromised protection.[17]

In sum, the desired effect of mapping one VEOS node (process) per workstation is maximum overall utilization of each workstation. This increased utilization is expected due to efficient internal process management and reduced operating system overhead spent on heavyweight multiprocessing.[18]

3) To avoid using OS kernel or hardware assistance for process management.

Again, this assumption serves the GCD principal by avoiding platform specific features, instead encouraging a portable process management mechanism within VEOS.

Kernel

The VEOS architects strove to define a small set of primitive capabilities that in combination would serve a broad range of computational needs. Again, the premise is that a small yet complete set of *primitives* should remain easily maintainable and well understood. The fundamental capabilities of VEOS manifest as composable primitives for generic data, process, and communication. Together these primitives form the VEOS Kernel.

Match & Substitute (Data)

As suggested earlier, a generic tuplespace data model offered the potential for simplicity, structure, and flexibility for working with databases, inference, and process coordination. In order to establish complete generality, VEOS incorporated a recursive tuplespace, which supports tuples within tuples, or *grouples*. Moreover, VEOS sought to provide full access to data through generic pattern matching and substitution over the *grouplespace*.

---

[17] Of course, VEOS nodes running on any multiprogrammed workstations must share cycles with other applications. For occasions when optimum performance is necessary, background processes can be suspended or reduced in priority in order to achieve the fullest utilization of the underlying processor.

[18] In particular, overhead spent on pre-emptive context switching and VM paging.

With a fully general access language to the grouplespace, VEOS sought to provide a testbed for Linda-like process models, rewrite languages, and database techniques in the context of VR.

Program Control (Process)

As a prototyping system, VEOS needed a simple yet flexible language to easily experiment with ideas. Lisp suited this need perfectly and became the primary language of VEOS. Although Lisp's syntax and semantics are divergent from those of many other conventional programming regimes, Lisp is a well established language with common usage idioms. Furthermore, Lisp is arguably easier to learn and use than most programming languages, and so could provide a bootstrapping layer for less technically inclined VE designers.

Lisp is extendible such that new functions can be incorporated which use the basic Lisp syntax and semantics. The match and substitute primitives mentioned above are built into VEOS's Lisp interface, providing a bridge between user program control and database capabilities. Lisp's fundamental data type, the list, has a direct mapping to the grouple. Both lists and grouples may contain any combination of atomic data types or more lists or grouples respectively. For all intents and purposes, the terms *list* and *grouple* can be used interchangeably.

Lisp also provides code-data equivalence. This means that Lisp code can be addressed as data, placed in the database, and retrieved at some later time as instructions.[19] This simple capability, in combination with generic tuple primitives forms the basis for a match/substitute/execute paradigm such as those used in expert systems and other rewrite systems.

Message Passing (Communication)

Since VEOS was to support distributed applications, inter-node communications comprised the third fundamental Kernel component. Again, primitives for message passing are tightly coupled to the Lisp interface. Just as Lisp's native data structure (the

[19] e.g. as rules or evaluable expressions.

list and atomic data types) smoothly translates in and out of the grouplespace, the Lisp data format easily translates to and from the network. This translation (also called *marshaling*) happens transparently to the Lisp level, thus providing a seamless link between multiple VEOS nodes and their corresponding Lisp environments.

The VEOS architects chose a simple and baseline communication model that provided the building blocks for more complex communication paradigms. The defining attributes of the Kernel message passing primitives are:

- Reliable and robust.
- Asynchronous.
- Point-to-point.

Armed with simple and well-implemented primitives for process, data, and communication, the first programmers of VEOS could experiment with more complex semantics such as non-preemptive lightweight process constructs that share address spaces, asynchronous variations of RPC, asynchronous messages among distributed objects, and Linda style communications. Hopefully, experimentation with these fundamentals would guide the next level of systematization. And indeed, early experiences with these primitives obviated the form of Fern, the VEOS Kernel macro component.

Application Specific Libraries

With the VEOS Kernel taking on a Lisp edifice, it was clear how to build custom capabilities for use with VEOS. Alongside the Kernel services, application-specific services take their stations as Lisp primitives. Many custom services are used frequently enough to be 'standard' but remain distinct from the Kernel either because their implementations are platform-specific, or they are not stable fundamentals but evolving developments.

Physical Transducers

Device drivers or *transducers* are examples of capabilities that a VEOS user would almost always use, yet they are usually platform specific, and so remain separate and modular with the Kernel. Transducers for VR can be categorized into *drivers* and *renderers*. Drivers sample the physical world and generate input to the VE, whereby *driving* the dataflow of the VE. Examples of drivers are the Dataglove™, the wand, and voice recognition. Renderers translate some aspect of the VE to the proper physical stimulus for the participant, whereby *rendering* the state of the VE. Examples of renderers are 3D binocular image generator, spatial sound generator, and voice synthesis.

These capabilities are stated generally to reemphasize that VEOS users receive abstract access to primitive capabilities. Thus, transducer primitives are as abstract and simplified as the Kernel primitives. From the VEOS user's point of view, renderers and drivers are further extensions of the Lisp interface language that have desirable side-effects.

All physical transducers have one main similarity; that they are leaves of the data flow graph of any VR program. Transducer primitives are the primary data sources and sinks. Again in the data flow metaphor, the participant completes the cycle by interacting with physical devices. This similarity of many data flows to and from the participant suggests that transducer designs can share methodologies. Moreover, the VE programmer's mental load is reduced when transducers use similar semantics.

Bridges to Complete Packages

Like physical transducers, many custom libraries provide a channel to from the abstract VEOS Lisp environment to existing full-featured packages.[20] Occasionally, an application will need to integrate the services of an existing sophisticated software tool with the experimental VE capabilities that VEOS provides. Again, the Lisp interface language can provide a common ground where VEOS capabilities and imported services can meld.

Optimized Toolboxes

Other custom services come about not because of imported or exotic services but simply for performance reasons. Regularly, VEOS designers use Lisp to prototype an algorithm or system for doing some needed computation. Over time, the prototype may

[20] External interfaces have been built to Mathematica and the AutoMod simulation package.

become more refined and may begin to receive heavy use. In such cases, more experienced programmers can optimally reimplement the service at a lower level[21] while retaining the abstract Lisp interface.

This package refinement phenomenon has occurred several times with VEOS. Examples are a collision detection algorithm for billiard ball simulation, autonomous flocking algorithms, the Mercury participant system, and the Fern task management module of VEOS.

Meta Concepts

Of particular note are custom toolboxes which embody important processing idioms. Mercury and Fern are prime examples of idioms of VEOS that became so central to standard VE programs that they were reimplemented at a much lower level with great care toward performance without compromising the simple interface.

Mercury is an integrated participant interaction tool kit which handles i/o and certain participant interactions with greater efficiency than was possible in the generic prototyping structure of VEOS. Like other custom tool kits, Mercury can be used in conjunction with other VEOS generic capabilities.

Fern

Fern is a tool kit that became fundamental for practically every VEOS program. Fern provides a system of meta-Kernel capabilities in a platform independent way. The Fern methodology embodies programming idioms that arose from using the Kernel on its own. And because Fern follows the GCD principle, VEOS subsumed Fern as the basic mechanism for distributed task decomposition. In keeping with the design principle of hybridization, Fern provides multiple features for data, process, and communication.

Nodes

A single invocation of VEOS is called a *node*. Each node corresponds to exactly one Unix process. The VEOS Kernel capabilities operate entirely within the context of a

[21] e.g. C language.

single node. The Kernel match and substitute primitives operate only on the local grouplespace, the native Lisp primitives operate only in the local Lisp environment, and the Kernel communication primitives send and receive from the point of view of the calling node. Fern addresses the whole distributed program, which may consist of many nodes.

Pools

Fern manages sets of distributed VEOS nodes into *pools*. A Fern pool implements a virtual multiprocessor which VEOS programs can utilize with greater conceptual ease than a set of independent VEOS invocations around the network. Although the nodes in a Fern pool may be heterogeneous workstations, VEOS programs experience uniform access to the Fern task capabilities of each node in a pool.

Entities

Fern employs a task decomposition scheme akin to distributed objects in Emerald. Fern process/data objects are called *entities*. Entities provide lightweight process constructs, as well as lightweight data partitioning constructs. A Fern pool is a distributed execution environment for user-programmed entities. As in pure object languages such as Smalltalk, a VEOS program is completely described by the bodies of all the entities involved in the overall task.

Each entity receives a partition of the Lisp environment for private functions and data storage. Each entity receives a private partition of the grouplespace for data management that takes advantage of match and substitute primitives. Entities also have access to a partition of the grouplespace that is shared among many entities.

Entities can have persistent processes which cooperatively share processing cycles between other entities in a pool. Every Fern entity consists minimally of a unique name called an *entID*. Entities use entIDs like traditional *capabilities* to reference one another in several ways.

Acting as conventional 'objects', entities can transfer process control with varying semantics around a VEOS program with messages as in Smalltalk or Emerald. Like objects waiting to receive messages, entities can remain inactive until prespecified data arrives in the shared grouplespace.[22]

Frames and Persist Procs

At the heart of many simulation techniques is the concept of a *frame*. Roughly, a frame is a cycle of computation during which the entire simulation advances one time step. Updates are propagated around the system at the end of each frame. VEOS embraces a flavor of frames for VE computation.

To make the frame concept work without preemption, VEOS entities perform discrete, atomic and repeatable tasks called *persist procs*. Each VEOS node employs a cyclic executive algorithm to schedule persist proc execution. Fern interleaves persist proc execution with other node activity such as handling incoming messages. A node's 'frame-rate' is determined primarily by the amount of work involved in performing all persist procs once.

Object-Oriented Primitives

Entities can be programmed under conventional object-oriented paradigms. Fern provides mechanisms for defining classes and subclasses of entities, creating instances of entities, defining methods as abstract interfaces to entity functions, and sending messages to entities' methods.

The VEOS architects incorporated these object-oriented features because they provide programmers with a familiar and well established programming paradigm. Emerald showed that a distributed object system was practical and it further outlined many of the implementation pitfalls. These object-oriented features serve as conventional fallbacks within VEOS's hybrid design.

Shared Virtual Grouplespace

Fern implements a virtual grouplespace that is shared between many entities, possibly residing on different nodes. This shared virtual grouplespace provides an

[22] Here, inactive means idle, not using cycles.

abstract programming model of shared memory and, like it's operating system cousin DSM, is implemented with message passing. Fern's virtual grouplespace combines tuple and pattern matching methodologies from Linda with coherence algorithms from DSM.

The shared grouplespace is an experiment in communication models. As in both DSM and Linda, Fern's grouplespace allows 'memory' to be a medium of communication between processes. Shared memory has advantages over message oriented communication because memory[23] is more naturally accessible from a programming language than the network. With respect to process communication, programmers can, in principle, focus more on the content of communication and less on the details of transporting data.

[23] In this case, a database acts as memory.

Chapter 4: VEOS 3.0 in Detail

This chapter outlines the specifics of the VEOS implementation as it evolved and as it stood in the Spring of 1993. The title VEOS 3.0 encompasses a system of components, all of which are tied together by the VEOS Kernel. Nearly every VEOS application makes use of each of these components to achieve a complete VE. The primary components of VEOS 3.0 are the VEOS Kernel, common application-specific libraries, and the Fern composite of Kernel services.

Platforms

Unix platforms were chosen for the initial development, although the VEOS design makes minimal assumptions about the computing platform. In addition to Unix's standard services for uniform file access, networking, and heavy virtual memory usage, Unix provides multi-user capabilities and allows resource sharing and protection between VEOS and other users' work.

The current VEOS implementation has run on varieties of Unix implementations including Silicon Graphics Irix, SunOs, and DEC Ultrix. Although the code packaging explicitly accounts for slight differences in these platforms, the implementation is dogmatically generic and would require minimal effort to build VEOS on other flavors of Unix or Unix server.[24] VEOS and most all the included extensions are written in C to conform as well as possible to most compilers.

The full VEOS package includes examples, meta-tools, and application-specific libraries. The full VEOS also incorporates supported emulators for significant platform-specific libraries, such as the Graphics Language for non-Silicon Graphics hardware.

Kernel

System Dependencies

[24] Such as A/UX, NT, NeXTStep, or ThinkC Unix libraries.

For ease of reasoning, users are not required to consider memory consumption when using both Lisp and the grouplespace.[25] Also for simplicity, users can pass any Lisp structure in an inter-node message. In other words, Lisp structures, grouplespace storage, and inter-node messages can be of any size. In order to achieve this ideal, the corresponding Kernel components make heavy use of virtual memory, especially on workstations where physical memory is limited or shared between many applications.

Of course, relying naively on Unix memory services can yield unbearable performance. The VEOS Kernel uses Unix memory management[26] sparingly in order to reduce costly OS kernel interface crossings. The VEOS Kernel reuses all allocated memory via multiple freelists and takes advantage of known Kernel usage patterns. Although no explicit provisions are made inside the VEOS Kernel to encourage locality of reference, informal tests have revealed reasonable locality behavior for small programs.

VEOS inter-node message passing is implemented on top of the TCP/IP network protocol. TCP is accessed through Unix *sockets*, a relatively standard service. TCP is reliable, point-to-point, and implements these semantics over local and wide areas. Although TCP is sometimes considered inefficient, it provides a standard interface to the very semantics VEOS required. Furthermore, the VEOS architects reasoned, in order to provide reliable, point-to-point message passing on top of a more efficient network protocol such as UDP, VEOS would have to make performance compromises similar to those in the TCP implementation.

Nodes

As stated in Chapter 3, an invocation of the VEOS Kernel operates within a single Unix process and address space. This invocation, or *node*, contains the full set of Kernel capabilities: Lisp program control, match and substitute over a local grouplespace, and communication with other nodes. The remaining discussion of the Kernel implemenation takes the point of view of a single invocation of the Kernel. The section on Fern component of VEOS discusses issues regarding multiple nodes.

[25] However, users have access to this information for fine tuning programs.

[26] e.g. malloc().

Lisp Program Control

VEOS incorporates XLisp, a freely redistributable Lisp interpreter that, like the rest of the VEOS Kernel, relies only on the most common operating system services, namely memory management and uniform file access. Although XLisp includes neither a compiler nor most CommonLisp extensions, it is quite efficient and extendible. XLisp is written in C and can be easily extended from both C or Lisp.

XLisp provides hundreds of built-in, or *native* functions which are written in C. The Lisp programmer can define more functions in Lisp which behave the same as these native functions, except that they exist as interpreted Lisp code rather than 'hard-wired' primitives.[27] In addition, native Lisp primitives are included automatically when a VEOS Kernel is invoked whereas user-defined functions must be reloaded each time the user invokes VEOS.

The VEOS Kernel primitives are written in C and are bound to XLisp as native functions. In a full VEOS package, other standard primitives in addition to the Kernel's are also bound in. These primitives, such as 3D graphical rendering primitives, are used in almost every VEOS application, and have become part of the standard VEOS package.

With a simple programming model and no compile step, Lisp provides designers with a forgiving environment for prototyping ideas. At the bottom line, these ideas manifest as Lisp functions and their associated data structures. When this experimental code becomes more established and serves more users, the ideas can be reimplemented by experienced programmers in C to achieve better efficiency and generality. The reimplemented code can manifest the same Lisp interface. Consequently, code that has grown to rely on the prototype module can continue functioning as before but experiences increased performance. This process of upgrading Lisp modules to C occurred many times during VEOS development for meta-Kernel utilities and application-specific modules.

Match & Substitute

[27] (defun ...) is used in Lisp to define primitives.

Within a node, there is a single grouplespace which is empty upon initialization. Just as grouples are nearly equivalent to Lisp lists, the entire grouplespace can be considered as a single list. Just as lists can contain any data item including more lists, so can the grouplespace. Items can be inserted, deleted, replaced, or copied from the grouplespace. Thus the single grouplespace is an indefinitely large database that VEOS Kernel users can partition and expand ad infinitum for their particular requirements.

The grouplespace is accessed through Kernel primitives for *matching and substitution*. These terms are used broadly to mean both explicit data access based on the *placement* of data in the grouplespace, and implicit data access based on the *content* of the data in the grouplespace. In other words, users can access the grouplespace based on the order of the data, or based on the content of the data, or a combination of both.[28]

The Kernel module that implements these capabilities is called *Nancy*.[29] The three Nancy primitives are *vput*, *vget*, and *vcopy*. Vput inserts a single element, possibly a list, into the grouplespace. The destination (where to put the element) is specified by a pattern. Vput can insert the element between two existing elements in a grouple or it can replace some element(s) with the new element. Vcopy simply retrieves some element(s) from somewhere in the grouplespace specified by a pattern. Vget behaves like vcopy except that it destructively retrieves the element(s) that it matches. All three of these primitives incorporate matching, but only vput performs substitution.

The Kernel must be explicitly initialized with a Lisp call to (vinit ...) before using any of its primitives. During this initialization, the Kernel initializes the grouplespace to the empty list, represented by (). This is the one and only grouplespace for the node and the matching primitives apply only to it. The syntax of the primitives is as follows:

```
( vput <data element> <nancy pattern> :freq <"all"> )
        <data element> - single data item to insert. can be an integer, float, string,
                                   symbol, array, or list.
        <nancy pattern> - specifies the destination to insert or replace data.
        :freq "all" - optional argument requests exaustive substitution.
```

---

[28] By order, the fourth element of the third list is obtainable; by content, the first list that contains the string "key" is obtainable.

[29] The name was chosen to suggest a variant of Linda systems.

returns - for simple insertions, T/NIL; for substitutes, the data that was replaced.

(vcopy <npattern> :test-time <time-stamp> :freq <"all">)
  <nancy pattern> - specifies the source of the data.
  <time-stamp> - optional argument for time-selective matching.
  :freq "all" - optional argument requests exaustive matching.
  returns - list containing all data elements matched.

(vget <nancy pattern> :test-time <time-stamp> :freq <"all">)
  <nancy pattern> - specifies the source of the data.
  <time-stamp> - optional argument for time-selective matching.
  :freq "all" - optional argument requests exaustive matching.
  returns - list containing all data elements matched.

(vmintime)
  returns - nancy timestamp, guaranteed oldest time.

The Nancy primitives allow access to the grouplespaces through a pattern matching language. This language provides a succinct format to express the *site* in the grouplespace that transactions are to take place.[30] What transaction takes place at a site is determined by which primitive is being used - vput, vget, or vcopy. The rules of the Nancy pattern matching language are as follows:

A site of action is specified by a pattern. Patterns consist of information about where the site is and/or specific data that the site contains.

- The ^ (void) symbol specifies a location within a grouple for inserting. It points to the void between data elements for insertion operations (only vput). Technically, the ^ always matches.
- The > (mark) symbol points to a piece of data within a grouple for retrieval operations (vcopy or vget) or substitution operations (vput). It designates the immediately following element of the pattern as the site of action. The > does not itself match data.

For a pattern to be meaningful, one of these symbols (^ or >) must appear somewhere in the pattern. In other words, the pattern must always specify a site of action. Patterns also match against data. To specify how to match, the pattern can either specify actual data to compare, or one the wild card symbols below. Wild card symbols are content-blind and they match only on the existence of data element(s).

[30] Possible transactions are: insert, copy, delete, or replace.

- The @ (this) symbol specifies a single data element at a specific location within a grouple. This will match any single element including a grouple.
- The @n (these) symbol specifies exactly n sequential data elements within a grouple. This will match the next n elements. @1 is equivalent to @.
- The @@ (these all) symbol specifies zero or more data elements within a grouple. This will match all the remaining elements in a grouple. This special form is allowed only at the end of a pattern grouple.
- The ** (any) symbol specifies zero or more data elements *anywhere* within a grouple. This will match all the remaining unmatched elements in a grouple. This special form is allowed only at the end of a pattern grouple.

Patterns may also contain annotations for marking data elements in the grouplespace as new. Time-marked data provides a basis for logging the user's recent pattern match history.

- The ~ (touch) symbol specifies that the immediately following data element be 'touched' during a (vput ...) operation. That is, the data that matches the pattern symbol following the ~ is marked as having been recently modified. There can be any number of ~ in a pattern.

Anything else that appears in a Nancy pattern is taken literally and matched against the actual data in the grouplespace. Note that Nancy patterns are recursive. That is, a pattern may contain a grouple that contains wild cards elements and data elements, some of which are more grouples with more wild cards and data, etc. Appendix A contains a detailed session with examples of Nancy primitives and their usage.

It should be noted here that the grouplespace has been used as a local database in Fern and in combination with message passing to implement a DSM construct in Fern, but never to date as a substrate for a rewrite system as per the design. Early Kernel programming tended to use the grouplespace as a proving ground for different data organization and coordination schemes.

Message Passing

In multiprogramming systems such as VEOS, the choice of message passing semantics depends heavily on the process model. Generalized process models such as threads use preemptive multitasking to address complex multiprogramming issues such as perceived parallelism and processor utilization. In keeping with goals of portability and simplicity, the VEOS Kernel does not incorporate a generalized process model of multiple sequential processes. Thus the choice of native message passing semantics for the Kernel was limited.

For example, traditional synchronous RPC semantics would be difficult without some form of process preemption. Synchronous semantics imply that once a process makes a request over the network, the process waits (spinning or blocked) until the reply comes back. With preemption, the waiting process can be 'switched' away and another process can do useful work in the interim. Without support for preemption, all other work on the node must wait until the waiting process has completed its communication (e.g. received a reply).

The VEOS architects took this opportunity to explore communication paradigms alternative to that of the classical synchronous request-reply. It was reasoned that the following message passing semantics would be compatible with VEOS's simple process model.

- Point-to-point.
- Reliable and robust.
- Asynchronous.

Point-to-point semantics provide a simple communication model that can be iterated for multicast schemes. Reliable communication is also important in keeping simple programming semantics. More importantly, reliability implies that the sender can be assured of delivery without awaiting a reply. For example, reliable transmission semantics provide for *broadcast-only* communication models, where the sender does not wait for acknowledgments. Kernel message passing primitives guarantee further simplifications. First, that the user need not be concerned with message size. The Kernel discretizes large messages and transmits the separate pieces automatically. Second, that messages from a given node arrive in the order they were transmitted.

Asynchronous message passing semantics means that send and receive operations do not block for any reason. Although Kernel message passing primitives may return errors, they always return immediately.[31] Again, this asynchronous property is imperative in VEOS, where there is no support for generalized preemptive processes.

[31] Errors include 'destination unknown' or 'bad arguments'.

The two primitives that implement these capabilities are *vthrow* and *vcatch*. Vthrow transmits a single Lisp expression to the named destination. Vcatch retrieves the oldest waiting message from all destinations. The following describes the actual Lisp interface to the Kernel communication primitives.

The Kernel must be explicitly initialized with a Lisp call to (vinit ...) before using any of its primitives. During this initialization, the Kernel chooses a unique *port* on which to receive messages from other nodes. The combination of the hostname[32] and this port number uniquely identifies a VEOS node across the internet. The identifier of a node, called a *Uid*, is returned from (vinit ...). The following excerpts from trivial VEOS Lisp sessions demonstrate how to use the Kernel communication primitives.

~ ~ ~ ~ ~

At unix prompt of host 'jabberwock', a VEOS node is invoked:

```
jabberwock: /usera/veos/ %  veos
XLISP version 2.1, Copyright (c) 1989, by David Betz
>
```

Kernel uses the named port. Server nodes can choose known host and port.

```
> (vinit 9000)
```

Kernel returns the Uid of this node, the address for the entire invokation.

```
#("jabberwock" 9000)
>
```

~ ~ ~ ~ ~

At unix prompt of host 'vorpal', another VEOS node is invoked:

```
vorpal: /usera/veos/ %  veos
XLISP version 2.1, Copyright (c) 1989, by David Betz
>
```

No argument, Kernel chooses any free port.

```
> (vinit)
#("vorpal" 5500)
>
```

[32] Hostname can be specified as a string or IP number.

Send a simple message to node on jabberwock....

```
> (vthrow #("jabberwock" 9000) '(print "ping"))
T
```

~ ~ ~ ~ ~

From the node #("jabberwock" 9000), poll for messages...

```
> (setq msg (vcatch))
(PRINT "ping")
>
```

Call Lisp evaluator on 'active' message...

```
> (eval msg)
"ping"
```

~ ~ ~ ~ ~

Immediately evident is a tradeoff between performance and simplicity. For example, there is no separate 'control line' for system-level or high-priority messages.[33] As another example, messages are guaranteed to arrive regardless of size but no guarantees can be made about when messages arrive except that transmission order is preserved. These semantics do not provide highly deterministic real-time behavior, but in practice are sufficient for prototyping. Again, these Kernel primitives provide the building blocks for more sophisticated communication models that are discussed later.

The following is a more practical example with vthrow and vcatch. It shows one node acting as a 'dumb' server and another node sending asynchronous requests. The server is dumb because it blindly evaluates all incoming messages without any checking for authorization or syntax errors.

~ ~ ~ ~ ~

Startup the server node on a known host and port.

```
> (vinit 9000)
#("jabberwock" 9000)
```

Go into an infinite polling loop, evaluating all messages..

[33] However, Fern multiplexes this Kernel service to provide a user channel and a system channel.

```
> (loop
      (setq msg (vcatch)
      ;; vcatch returns NIL when no messages available
      (if msg (print "result: " (eval (msg))))
      )
```

~ ~ ~ ~ ~

Startup a client node.

```
> (vinit)
#("vorpal" 5500)
> (setq server #("jabberwock" 9000))
```

Could also use #("128.95.95.74" 9000) for explicit naming.

```
>
```

Send a some data to server node...

```
> (vthrow server "ping")
T
```

Send a function declaration to server node. Note, code is quoted so that it is evaluated for the first time on the remote node.

```
> (vthrow server '(defun f (a b) (+ a b)))
T
```

Send a call to the function...

```
> (vthrow server '(print (f 3 4)))
T
```

Send a self-returning message...

```
> (vthrow server '(vthrow #("vorpal" 5500) "reply"))
T
```

~ ~ ~ ~ ~

Meanwhile, the server node has received, evaluated, and printed the messages in that order they were sent...

Strings evaluate to themselves.

```
result: "ping"
```

Function definitions return the new funtion name.

```
result: F
```

The result of (f 3 4).

```
result: 7
```

The return vthrow was successful.

```
result: T
```

~ ~ ~ ~ ~

The client node #("vorpal" 5500) has received the self-returning message...

```
> (print (vcatch))
"reply"
```

~ ~ ~ ~ ~

Kernel Idioms

Programmers using the VEOS Kernel began to discover methods of using these basic primitives for more sophisticated problems. Many algorithms used main loops similar to the server example above. Variants were tried that interleaved other specific computation such as polling interface devices and refreshing output displays between message handling. These schemes, all of which are forms of the *cyclic executive* algorithm, eventually gave rise to the Fern process model. The Lisp code example below represents the essence of these cycle (or *frame*) oriented algorithms.

```
> (setq process-list '(poll-network
                       poll-devices
                       compute-simulation-step
                       post-deltas
                       update-display))
> (loop (mapcar 'funcall process-list))
```

Application Specific Libraries

Once the VEOS Kernel augmented Lisp with a reliable and expressive means of organizing a database via the grouplespace and coordinating distributed tasks via message passing, Lisp became the preferred common ground for other developing capabilities such as 3D rendering and spatial tracking. Through Lisp, these application specific libraries could share a generic interface, thus allowing natural connectivity and composability with each other.

Many significant modules are written in C in order to achieve tolerable performance or to utilize existing C level tools and libraries. By building Lisp 'wrappers' to these developing modules, programmers also enjoy the testing and debugging benefits of an abstract Lisp interface. Module builders often use the Lisp interface as an uncompiled scripting layer to work out ideas or to test specific features with various parameters.

Imager

The imager is perhaps the single most important non-Kernel module associated with VEOS. It provides an abstract interface to screen-based and binocular 3D graphical rendering capabilities. The imager is a real-time rendering package based on rigid polygon-based models. Runtime performance of the imager is optimized for the Silicon Graphics platforms where there is hardware support for primitive graphics functions. However, the imager does adhere to the GCD principle. It does so for non-SG platforms by incorporating VOGL, a public domain software emulator for the low-level graphics capabilities of the Silicon Graphics.[34]

Because in VR each participant 'sees' potentially different sets of objects from different viewpoints, there is typically a separate invocation of the imager for each participant. The Lisp interface to the imager is relatively simple. It creates and deletes objects that are referenced by integer IDs. Transforms are specified by matrices which can be manipulated abstractly in Lisp with associated transform utilities.

Sound

Aside from an explicit voice interface, there are three ways that sound is incorporated into VEOS applications. By triggering simple monophonic samples through MIDI, triggering possibly pre-spatialized stereo sounds, and computing spatial sound in real-time. These capabilities are accessible in Lisp through primitives of various

[34] VOGL requires only X windows and a fast CPU.

modules. The spatial-sound interface is similar to that of the imager because both modules address the issues of objects in the VE, their positions, their properties, and the position of participant (viewpoint).

SensorLib

In addition to the display capabilities of the imager and sound renderers, VE applications also require complimentary input capabilities. SensorLib implements a generalized module for tracking the participant's movements and actions. SensorLib is an integrated collection of drivers for the myriad of spatial tracking devices currently available.

The SensorLib Lisp interface provides a simplified abstraction to the task of collecting user input. The data items that are passed across the Lisp interface to and from SensorLib are compatible with other modules that use spatial information. For example, SensorLib returns quaternions, a succinct representation for orientation.[35] Using associated Lisp primitives, the quaternion representation can be manipulated and reorganized into matrices, and then passed directly into the imager or spatial sound renderer.

Voice IO

Another interface capability that proves useful for many VE applications is voice input and voice synthesis. HITL programmers have wrapped developing voice i/o modules into Lisp primitives.

Newtonian Dynamics

The Newtonian dynamics package is the definitive application-specific module that is not directly concerned with interfacing to the participant. Though VEOS does not currently incorporate a module that implements a full set of Newtonian laws, many smaller modules have been designed for performing specific computations. Like most other modules, these are implemented in C for efficiency, then wrapped into abstract Lisp primitives. Examples of such capabilities are: calculating the frames of a billiard ball

[35] Positions are used for the participant's head and hands.

simulation, iterating the participant's viewpoint along spline-based 'flight' paths, computing the iterative steps of flocking behavior, and smoothing and prediction of data streams.

Fern

For over a year, the Kernel stood as the fullest extent of VEOS's generic computing facilities. Early VEOS programmers used these basic primitives in combination with the application specific libraries to piece together the first VEOS applications. These first integration efforts helped to define the set of requirements that characterized typical VEOS applications. Broadly, these requirements were:

- Simple task decomposition and process management.
- Uniform process communication (location transparency).
- Distributed database.
- Uniform protocols between modules.
- Distributed resource administration.
- General automation and reliability.

Again, there was significant motivation to raise VE building capabilities to a level that designers and inexperienced programmers could use. The Fern system makes many semantic commitments in order to provide a simplified interface to VE design. Fern addresses these requirements by providing a complete meta-layer to the Kernel, yet still allowing more ambitious VEOS users to access to the lower level Kernel primitives already described.

System Dependencies

Fern can be thought of as a macro-facility to the Kernel. As such, Fern was initially written completely in Lisp using only the basic Kernel primitives and Lisp program constructs. As Fern evolved, it experienced more and more users and usages. This drove the need for refinement and attention to performance. Soon, much of the Fern implementation moved into C in order to achieve performance and robustness. Although the Fern implementation moved closer to the underlying platform, the concepts remain platform independent.

Fern depends on two standard Unix services similar to those provided in other operating systems. These are timing services and remote shell invocation. Timing services are used to track the frame rate of each node. Fern uses remote shell services[36] to bootstrap the Fern processor pool to distributed Unix processes at the beginning of each invocation of a Fern program. Once the distributed processes are started via remote shell operations, the Kernel provides all remaining communication through Unix sockets.

Node

The node is the Fern unit of processor allocation. Fern manages sets of uniprocessors (e.g. workstations) as pools of nodes. Fern nodes map to Unix processes which ideally map to workstation processors. Under this model, ideal conditions are met when each Fern node is the only process running on each workstation respectively. Fern utilizes processes as nodes by multiplexing the processes for many lightweight tasks, each playing some part in the whole Fern program.

Using a single process per processor has two benefits. First, it suits the GCD principal. Second, because Fern does not require a fully general process model, Fern can use one process to utilize each uniprocessor more effectively than by using multiple processes per node. Fully general process implementations incur considerable overhead for their generality due to preemptive context switching, and virtual memory usage when many processes share the same physical memory. By analogy, Fern's internal utilization strategy corresponds to well-known operating system memory management strategies that trade external fragmentation with segments for internal fragmentation with pages.

Within a Fern program, each node is self-regulated and autonomous. During normal runtime, each Fern node carries out a variant of the cyclic executive algorithm. This amounts to performing a series of operations over and over as fast as possible. These operations are the basic tasks of each Fern node:

- handling incoming network messages,
- giving time to application tasks,
- propagating changes in the Fern shared memory.

---

[36] e.g. /bin/rsh

The Fern node main loop is written in C and its numerous complexities are not relevant to this discussion. The following is a simplified XLisp excerpt that demonstrates the basic form of the Fern cyclic executive:

~ ~ ~ ~ ~

These functions are system code, included as a package by the user.

```
(defun cyc-exec-main-loop ()
  (loop
    (cyc-exec-do-frame)))

(defun cyc-exec-do-frame ()
  ;; evaluate active messages
  (mapcar 'eval (reverse (cyc-exec-gather-waiting-msgs)))
  ;; give one frame's worth of processing to user code
  (mapcar 'funcall user-process-list)
  ;; pass database changes to other nodes
  (cyc-exec-propogate-deltas)
  ))

(defun cyc-exec-gather-waiting-msgs ()
  ;; conceptually, retrieve all messages that
  ;; have arrived since call to this function.
  (let ((msg (vcatch)))
    (cond (msg (cons msg (cyc-exec-gather-waiting-msgs)))
          (t NIL))))

;; the system tracks grouplespace modification history
(setq cyc-exec-timestamp (vmintime))

(defun cyc-exec-propogate-deltas ()
  ;; get a copy of what is 'new' in the grouplespace
  ;; since last time user code was evaluated
  ;; send deltas to other grouplespaces
  (let ((deltas (vcopy '(> @@)
                :test-time cyc-exec-timestamp)))
      (dolist (node other-nodes)
        (vthrow node
               `(cyc-exec-incorporate-deltas ,deltas)))))

(defun cyc-exec-incorporate-deltas (deltas)
  ;; adding deltas is not a trival operation,
  ;; this unspecified function helps out
  (cyc-exec-put-deltas-into-gspace deltas)
  ;; update what is 'new' in the grouplespace to
  ;; distinguish from local changes made by user
  (vcopy '(> @@) :test-time cyc-exec-timestamp)
  t)
```

Application code, written by the user.

```
(setq user-process-list '(poll-devices
                          calculate-next-frame
                          update-display))
```

Begin processing.

```
(cyc-exec-main-loop) ...
```

~ ~ ~ ~ ~

Again, this example demonstrates the basic framework of the Fern cyclic executive. In the actual implementation, there are more sophisticated mechanisms for doing these tasks. For example, the C version of Fern checks internal queues to determine if there is any useful work to do each time through the loop. If no local work remains[37], new work can only be generated from incoming network messages. Thus, when no work remains, Fern blocks the entire process waiting for network messages. Another refinement that the actual implementation makes is handling system control messages which are used for inter-node message pacing.

[37] e.g. user process or grouplespace changes to propogate.

The above Lisp prototype only partially addresses the general issue of how much work to do each frame. Fern must retrieve and evaluate exactly one frame's worth of messages each cycle. This complicates the implementation because more messages may arrive while handling the messages that arrived since the previous frame. The Fern process model explicitly uses the frame as the quanta of computation. As such, it is unambiguous how much application processing Fern does each frame. The Fern process model is further elaborated later in this chapter.

As for propagating virtual grouplespace updates each frame, this coherence algorithm depends on many factors. The primary constraint is the relative work loads of the other nodes that will receive updates. In other words, each node's coherence algorithm is sensitive to the effect it has on other nodes. The coherence protocol and its flow control mechanism will be detailed with respect to the shared grouplespace.

Fern's cyclic executive algorithm has some other inherencies to note. Fern makes simplistic assumptions about runtime circumstances that lead to limited determinism and real-time performance. For example, there is no explicit control over the frame rate of any given node. Instead, Fern nodes simply cycle through their tasks for each frame as fast as possible. As such, achieving a specific frame-rate is a matter of restructuring the application tasks, coding things discretely and efficiently.

Fern does guarantee some degree of determinism, however. For example, it is assured that a message sent at some time will be handled by the receiving node within one frame. Although a single frame can take a suboptimal amount time, that time is finite. Fern's inter-node message flow control helps to keep consistent frame rates.

Pool

A Fern program begins when the user manually runs a Fern node. This first node, called the *console node*, is initialized with a list of hosts on which to build nodes. Fern automatically launches and harnesses a distributed set of Unix processes around the network each running a VEOS node, thus creating a pool of nodes. This pool provides the distributed bed of computation, akin to that of a multiprocessor.

Semantically, Fern makes little distinction between local and remote operations within the pool. That is, Fern provides location-transparent access to resources across the pool. These resources are: files from the hosts that the nodes run on, processing cycles on those hosts, diverse peripheral devices, and platform specific capabilities like 3D rendering or voice recognition. The pool is the implicit context for all generic process and data operations (e.g. entity creation, destruction, and communication).

For simplicity, the number of nodes in a Fern pool remains fixed over the course of a program run. Programs that drastically change resource requirements at runtime can overallocate the number of nodes, and hence host cpus, in a pool. This practice is supported because nodes that are doing no work remain suspended at the Unix level thus incurring zero cost to other users of those workstations.[38]

Other situations demand more dynamic node allocation. For example, like a continuously running server, a Fern program may remain running indefinitely with its fixed node pool. Then, other users may sporadically invoke separate and distinct Fern programs that exchange services and information with the continuously running server pool. The server pool may be used to preserve certain circumstances across runs or between many users. In any case, Fern allows separate pools to *merge* at runtime in order to facilitate more flexible resource sharing after the initial pool invocations.

During a Fern program execution, each node experiences different computational circumstances based on three factors:

- different *processor speeds* due to heterogeneous workstations.
- different *Unix loads* due to other users sharing the workstations.
- different *application loads* due to the program's specific task distribution.

Consequently, nodes have inherently independent frame rates. This inevitability warned the VEOS architects that high processor utilization would be difficult to attain in parallel applications. This problem is partially addressed by using an asynchronous communication model. Each node in a Fern pool 'respects' the frame rates of the other nodes in the pool. In other words, nodes actively limit the amount of outgoing stream-type communication in order to fit the receiver's ability to digest messages. This inter-

---

[38] The Kernel calls blocking select() for network messages.

node flow-control is implemented transparently to the user. Flow control messages are an example of system control messages that use a different effective communication channel than application-level messages. For more sophisticated programs, the user also has access to this flow-control mechanism.

Fern provides uniform access to a large pool of data and computation resources, namely that of all the associated Unix processes and their respective address spaces. These resources are partitioned and utilized by different components of the same Fern program, or potentially by distinct programs and even different users in the case of merged pools. It is in this context that the issue of protection arises.

On multi-user platforms such as Unix workstations, VEOS and all the nodes in a pool are strongly isolated from other users of the workstations by the standard Unix protection mechanisms of separate addresses spaces, etc. However, within a Fern pool, every capability is accessible by any entity.[39] In particular, Fern supports the root-level capability of distributed Lisp code evaluation. Entities can unabashedly send Lisp expressions to be evaluated in another entity's context, local or remote. This gives all application code direct control of database organization, process implementation, and message passing semantics anywhere in the pool.

The reasoning behind this self-disciplined protection methodology is twofold. First, while Fern provides inexperienced programmers with simple, structured mechanisms that don't allow programs to run amok, Fern also gives more ambitious programmers enough rope to hang themselves in the form of direct access to lower level primitives. This freedom facilitates rapid prototyping and experimentation. Of course, this philosophy cannot work in all development settings. At HITL, there is small community of users that develop modules cooperatively and so the added flexibility of no protection outweighs the loss in robustness.

Second, an end goal of VEOS is to bootstrap into the VE, where new protection issues arise. These forms of protection will implemented by the VE application and whatever virtual tools the participant carries. Consequently, it was decided not to

[39] Provided the person using Fern has Unix access privaleges to host specific resources, such as files and data ports.

overengineer VEOS with respect to protection, and that more attention be focused toward the corresponding issues in VR.

Entity

The *entity* is the basic unit of task decomposition in Fern. Fern's hybrid design gives entities multiple constructs for data and process. In terms of data, Entities can employ combinations of these four constructs:

• Private storage in the local Lisp environment.

This is the most efficient storage for variables and data structures that are global to all parts of the entity, but not visible to any other entities.

• Private storage in the local grouplespace.

This is less efficient than Lisp storage, but provides pattern matching over the entity's local data. Again, the local partition of grouplespace is only visible to the entity.

• Public storage in the local Lisp environment.

This is the most efficient way to share data between entities on the same node. As with any shared memory in a multiprogrammed environment, the user must take special care to ensure proper sharing semantics since it is not always obvious when entities will be 'running' and manipulating the shared memory locations. Non-atomic data access is not a troublesome issue since VEOS does not support preemption. Object-oriented method calls can also be used for sharing data between local or remote entities. This mechanism is less efficient but semantically more structured because the data arguments are passed between entities along with program control.

• Public storage in the shared virtual grouplespace.

This is the least efficient, but most general form of data sharing in Fern. Changes to an entity's partition of the grouplespace automatically propagate around the pool to subscribing entities. This shared grouplespace works within the concept of the Fern *space*. Fern spaces serve to isolate sets of selected entities that share related data. Within spaces, included entities can subscribe to the changes in each others grouplespaces.

These various forms of program memory and data storage are further detailed in a section to follow. In terms of process, entities can employ combinations of these three constructs:

• Persistent processes.

Persistent processes or *persist procs* are discrete tasks that Fern repeatedly executes once per frame. These can be started, executed, and stopped quickly and thus provide a very lightweight process construct. Like all Fern application code, persist process are executed atomically, as there is no preemption.

• Methods.

Methods are well-defined interfaces to specific tasks that entities can perform. Methods are invoked by other entities sending messages. Methods calls can be made with various semantics depending on the whether the entities are local, remote, or the message is only one in an indefinite stream. As with all Fern application code, methods are discrete and are evaluated atomically.

• React processes.

React processes or *react procs* are triggers that entities supply to respond to specific changes in the shared virtual grouplespace. Entities install these callbacks in advance, so that when matching data arrives Fern dispatches the appropriate entity code immediately. This differs from methods in that methods are explicitly invoked by a single calling entity, whereas react procs respond 'anonymously' to changes in the grouplespace. That is, the entity that makes changes to its partition of the grouplespace does not have to know about the responding entity(s).

In Fern, all application code executes within some entity's *context*, except during system initialization. This context is made explicit in that at all times, the global Lisp variable *self* points to the entID of the currently executing entity. Since all code (persists, methods, or reacts) belongs to some entity, all data or memory space (grouplespace or Lisp) is created by some entity's code and is thus associated with that entity. Again, an entity's context is the sum of all that entity's resources and capabilities as specified by its data and process.

The entity's context is specified by a discrete set of Lisp code that initializes memory usage, declares methods and processes, and so on. This entity definition provides a mechanism for distributing and instanciating the entire task that the entity performs. All Fern inter-entity sharing and communication mechanisms exhibit uniform semantics for local and remote entities. The exception, as mentioned above, is that the efficient Lisp data sharing mechanisms only work with proximal entities sharing the same node. And, although the semantics are mostly preserved in local and remote interactions, application performance varies greatly with different distribution strategies.

A Fern program begins when the user manually runs the *console* node. This node evaluates a few lines of Lisp which bootstrap Fern into full-scale operation. First, the node initializes the local Fern by calling (fern-init ...) with a list of hosts to run nodes on. Second, the node instantiates the first entity of the program by calling (fern-run ...) with name of that first entity. This first entity is called the *spore* of a Fern program. As with all new entities, Fern evaluates the spore's entity definition code on the chosen node in the freshly initialized pool. In addition to creating data and process constructs, the spore may instantiate more new entities, and the program unfolds in this manner.

The minimum overhead an entity can incur is an empty and unused partition in the grouplespace and some locations in low-level hash tables. Fern references all entities' data structures through their entIDs. EntIDs consist of the Uid of node the entity resides on and a unique entity index for that node.[40] EntIDs are resolved in real-time upon reference for communication or other Fern services. Note that since EntIDs contain the node that they reside, entities are not mobile. As a result of trail and error, it was found that in the prototyping setting, it was most practical to use this location-evident entID format for unambiguous and efficient destination resolution. The previous alternative involved paying the performance and complexity costs of location-independent endIDs in expectation of rare applications when dynamic entity mobility is required.

Process Model

[40] The node's host is encoded in 32 bits by its IP number, the node's port in 16 bits, and the entity index in 16 bits.

When an entity is instantiated, the Lisp *entity definition* is evaluated atomically or to completion. An entity definition is simply a file of Lisp code which, when evaluated, creates all the entity's initial data and process constructs. Technically, the entity definition is a form of process where the entity could do meaningful computation. However, for Fern to perform properly, the entity definition should be a discrete set of configuration commands. These initializing commands make requests for data allocation, process initialization, or more new entities.

The entity definition code provides an informal mechanism for entity classes and instances. In essence, entity definitions are classes of entities. The entity definition itself is a first class citizen that can be loaded unevaluated, bound to a symbol, stored in the grouplespace, or sent as a message. Furthermore, within an entity definition, the code can include other entity definitions, thus providing an inheritance mechanism. Instances of entities are created by loading and evaluating an entity definition via (fern-new-ent ...)

Aside from the entity definition, there are three components to the Fern process model: persist procs, methods, and react procs. All Fern application tasks are implemented as one of these forms. The following describes the individual differences between the three forms of Fern process:

<u>Persist Procs</u>

During a single frame, Fern's cyclic executive evaluates every persist proc installed on that node exactly once. For smoother node performance, Fern interleaves the evaluation of persist proc with evaluation of queued asynchronous messages. In round-robin style, each persist proc gets a turn to execute on the node each frame. When a persist proc executes, it runs to completion like a procedure call on the node's only program stack. In contrast, preemptive threads each have their own stack where they can leave state information between context switches. Persist procs can save state between invocations in caches based on any of the entity data constructs.

Persist procs can be used, as with the signal processing metaphor, to perform some discrete computation on a frame of data. For example, to apply repeated transformations to some object or viewpoint. Or, persist procs can be used in polling for data, such as from devices or some other source external to VEOS. Then, when data arrives, a persist proc can cascade program control to the data driven parts of the program through one of the other process mechanisms below. The following is a simple persist proc example.

```
;; define the body of the persist proc
(defun my-proc ()
  ;; poll physical device for data from dataglove
  (setq raw-data (read-data-from-hand))
  (cond (raw-data
     ;; asynchronously update user's view
     (fern-send renderer "relocate-hand" raw-data)
     ;; parse the gesture
     (setq cooked-data (parse-gesture raw-data))
     (if (cooked-data
         (fern-send cmd-engine "digest" cooked-data)))
        ))
)
;; install the persist proc with Fern
(fern-persist '(my-proc))
```

This persist proc represents application code that begins data (or event) driven processing. Note the two tests in the code that save wasted work if the data is insufficient to continue. Because persist procs often involve polling, they often call application specific primitives written in C. The (read-data-from-hand) primitive would most likely be written in C since it accesses devices and requires C level constructs for efficient data management.

The embedded calls to (fern-send ...) make asynchronous calls to particular methods of other entities, thus propagating program flow. 3D rendering is a relatively heavy computational task, so the renderer entity would probably reside on a remote node. The gesture parser could practically reside on the local node. If the gesture entity indeed resides locally, the (fern-send ...) will have posted a message in the form of a method call to that entity. Fern will handle that message during the message evaluation phase of the next frame.[41]

<u>Methods</u>

Like Smalltalk methods, Fern methods are used to pass data and program control between entities. An entity can invoke methods of other entities by sending messages. The destination entity can be local or remote to the calling entity and is specified by the destination entity's location-evident entID. A method is simply a block of code that an entity provides with a well-defined interface. Methods are usually defined during in the entity definition code, but can also be installed at any time during runtime. Below is a sample method for a robot entity that moves through some space:

```
(let (robot-current-position)

  (fern-def-meth "robot-take-a-step"
      (lambda (direction)
           (robot-leave current-position)
           (setq current-position (robot-move direction))
           (robot-take-inventory current-position)))

  (fern-def-meth "..." ... )
)
```

Note that Fern methods rely on the XLisp implementation of anonymous functions via the (lambda ...) function. This example uses the (let ...) construct to provide variables that are global only to a set of methods for this entity. The mechanism for calling methods can be used with three different semantics:

Asynchronously...

```
(fern-send robot-entid "robot-take-a-step" 'north)
```

Asynchronous method calls are the most common type and ensure the smoothest overall performance. This semantic is defined such that the calling entity gets program control

[41] Handling messages involves dispatching to the appropriate entity.

back immediately regardless of when the method invocation is handled by the receiving entity. When the receiving entity is remote, a message is passed to the Kernel inter-node communication module, marshaled into network form, and sent to the node where the receiving entity resides. When the remote node receives the message, it unmarshalls the message back into Lisp form and posts it on the asynchronous message queue. As described earlier, Fern empties this message queue each frame in the order of message arrival. 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. However, no network conversion takes place, making the local message pass very efficient.

Although asynchronous message delivery is guaranteed, there is no guarantee when the entity will actually receive the method invocation and execute the method code. As such, this asynchronous semantic is used when timing is not critical for correctness. In other words, it is good enough that the method occurs as soon as possible. In cases where timing is critical, there are common idioms for using asynchronous semantics to do synchronization. Or, when necessary, Fern also provides a synchronous semantic.

Synchronously...

```
(setq inventory
 (fern-seq-send robot-entid "robot-take-a-step" 'north))
```

'seq' is short for sequential, and suggests serial computation.

The synchronous semantic means that upon making the call, the calling entity blocks, control is immediately passed to the receiving entity, and returns only when the method is complete. Unlike asynchronous method calls, synchronous calls also return the exit value of the method itself.

Although the VEOS communication model is inherently asynchronous, there are two occasions when the synchronous semantic may be desirable. When the calling entity needs the return value of the method, or when the calling entity needs to know exactly when the method completes. There are asynchronous solutions for these situations. For example, for the request-reply problem, the calling entity can provide a reply method for the receiver of the request to send back a reply. For the synchronization problem, a similar arrangement can be made so that when the caller's reply method is invoked, the caller knows that the receiving entity has completed the method.

Although both of these requirements can be handled by asynchronous means, it 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 semantics is whether the destination entity is local or remote. In the remote case, synchronous semantics will sacrifice local processor utilization because the entire node blocks waiting for the reply, but in doing so the calling entity is assured 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.

Synchronous method calls also run the risk of infinite recursion. In the following trivial example, the method calls itself asynchronously, thus generating a self-sustaining process, much like a persist proc.

```
(fern-def-meth "go" (lambda ()
                      (print "going...")
                      (fern-send self "go"))
```

But in this variant, the recursive call is synchronous and will result in a stack overflow.

```
(fern-def-meth "go" (lambda ()
                      (print "going...")
                      (fern-seq-send self "go"))
```

There is another circumstance that requires yet a third method calling semantic. It is that of using methods repeatedly to implement a data stream between entities. Because the entities may reside on different nodes with different frame rates, the cooperating entities may experience different abilities to produce or digest messages. In this case, methods can be used with a flow-control mechanism.

Sensitive to frame-rates...

```
(fern-persist `(progn
   (setq next-direction (get-next-direction))
   (fern-str-send ,robot-entid
                 "robot-take-a-step next-direction)))
```

Here, the persist proc may be able to generate messages faster than the node containing the robot entity can dispatch them to the right entity. (fern-str-send ...) will send the method call only if the *stream* between the two nodes is not full. The user can set the size of the stream which indicates how many buffered messages to allow. As detailed in Chapter 6, a larger stream gives better throughput because of the pipelining effect, but also results in bursty performance due to message *convoying*.

This stream semantic is usually used for transmission of 'delta' information. That is, a stream of data where some of the data items can be dropped without loss of correctness. This delta principle is used for the coherence of the shared virtual grouplespace. If data items cannot be dropped with method calls, the application can become smarter by checking the stream first, again guaranteeing arrival.

Here, the code only creates a delta when the stream is available.

```
(fern-persist `(if (fern-stream-clrp ,robot-entid)
                   (fern-str-send ,robot-entid
                                  "robot-take-a-step"
                                  (get-next-direction))))
```

React Procs

Like methods, react procs are callback functions installed by the application that Fern invokes upon receiving specific events. Also like methods, react procs are used to pass data and program control between entities. However, rather than being called directly by some other entity, react procs are triggered as a side effect when entities make changes to the shared grouplespace. In this way, react procs provide an anonymous mechanism of communication between many entities.

Each entity has write access to a partition of the shared grouplespace called the *boundary*. What one entity writes in its boundary partition of the shared grouplespace appears to other entities in their read-only *external* partition. This mapping occurs automatically via the shared grouplespace coherence algorithm. The boundary partition is made up of an indefinite number of the entity's public *attributes*. Attributes are pairs, a tag field and a data field. The tag or attribute name is unique to each entity and is used for matching in the shared grouplespace.

For, example an entity may need to know the changing value of some other entity's "color" attribute. So, once declaring its subscriptions, the entity might declare a persist proc that repeatedly matches in the shared grouplespace. This example uses a Nancy time stamp so that only new changes are found.

First, declare a subscription to the given attribute.

```
(fern-perceive "color")
```

Then, setup a process to wait for changes in the attribute.

```
(fern-persist '(progn
   (setq val (fern-copy.sib.attr the-entid
                                 "color"
                                 :test-time ts))
   (if val (printf "new color: " val))
   ))
```

This strategy amounts to polling the grouplespace. When many entities on the same node are polling simultaneously, a lot of process time is wasted on matching. With react procs, entities can specify which attribute they are interested in, and supply a callback function. Fern dispatches the user's code immediately when data for a matching attribute arrives in the entity's external.

Here, declare a subscription and a callback function at the same time.

```
(fern-perceive "color"
              :react (lambda (entid attr-val)
                       (if (equal entid the-entid)
                          (printf "new color: " val)))
)
```

The above react proc saves the user from writing the repeated pattern matching code and ensures minimum latency between a grouplespace change and the entity code reacting to it. The form and semantics of the shared grouplespace are detailed further in the next section.

Address Space

This section describes the data services of Fern. As mentioned earlier, there are four avenues for entity memory allocation: private Lisp storage, shared Lisp storage, private grouplespace storage, and shared grouplespace storage.

Private Lisp storage features are provided completely by XLisp. An earlier example showed how the (let ...) primitive could be used to allocate variables global to only one entity.

Shared Lisp storage is also implemented completely by XLisp. In fact, unless the application code does something special to limit the scope of it's Lisp variables such as using a (let ...), that entity's Lisp variables are accessible by all other code, and hence entities on the same node. Of course, this trivial data sharing through the local Lisp environment can only work between entities on the same node.

Private grouplespace storage is provided by Fern. Each entity receives a partition in the grouplespace which is subdivided into the *external*, the *boundary*, and the *internal*. One of the parts of the internal is an exclusive grouplespace for the entity's disposal called the *local*. Note that the grouplespace and the Kernel pattern matching primitives are recursive, so any subpartition of the grouplespace is effectively a full grouplespace. However, there are performance costs associated with using a deeper partition of the grouplespace.

Fern provides abstract primitives for accessing the separate partitions of the grouplespace. These Fern primitives (detailed in Appendix B) know which entity called them and access the correct partition for any given call. For example, an entity accessing it's local, might use code that looks like this:

First, setup an organization:

```
> (fern-put.locl '("my-db" ())
                 '(^ @@))
```

Next, add a couple of records:

```
> (fern-put.locl '("frog" 'GREEN #(0.0 1.0 0.0))
                 '(("my-db" (^ @@)) **)))
> (fern-put.locl '("bird" 'BLUE #(0.0 0.0 1.0))
                 '(("my-db" (^ @@)) **)))
```

Then, make a query:

```
> (fern-copy.locl '(("my-db" (("frog" > @2) **)) **))
('GREEN #(0.0 1.0 0.0))
```

Note that the Nancy pattern matching language still applies, but within a subpartition of the entire grouplespace. The Fern primitives encapsulate the user's pattern into a larger pattern that directs the action (put, get or copy) to entity's partition. Because the Nancy pattern matching language has proven difficult to learn and compose correctly, Fern also provides more structured primitives that offer put, get, copy services without requiring any patterns. Instead, they address simple named locations or *attributes* in the local. The example becomes:

The local is already initialized for attributes...

Add a couple of records:

```
> (fern-put.locl.attr '("frog" '(GREEN #(0.0 1.0 0.0)))
> (fern-put.locl.attr '("bird" '(BLUE #(0.0 0.0 1.0)))
```

Then, make a query:

```
> (fern-copy.locl.attr "frog")
('GREEN #(0.0 1.0 0.0))
```

In either case, underlying is a deeply nested pattern match specification which costs significantly more time to execute than simple pattern matches such as those described in the section on Kernel pattern matching. As such, this private grouplespace partition is used in place of private Lisp storage only when there is significant semantic benefit in pattern matching over complex sets of data.

Lastly, shared grouplespace storage is also provided by Fern. This is the most general and least efficient form of data sharing in Fern. The organization of the shared grouplespace reflects the metaphor of perception. That is, the entity's perception. As such, the shared grouplespace is designed around the perception or the context of the entity. Each entity has its own *boundary* partition of the local grouplespace. Fern partitions the boundary into an indefinite number of *attributes*. Entities also have their own *external* partitions. The external is made up of the boundaries of other *sibling* entities. In this way, an entity can write public attributes in its boundary, while other entities can read those attributes in their external.

As with conventional DSM, entities write to the shared memory and the data changes propagate around the pool automatically. But unlike DSM, each entity can write to its own boundary but can only read from its external which is made up of other entities' boundaries. For this reason, data always propagates in the same direction. That direction is: *from* entities writing to their boundaries *to* entities reading from their externals.

This basic shared grouplespace mechanism can be modulated by the user to further organize and restrict attribute sharing. First, to receive updates from other entities' boundary changes, an entity must subscribe or *perceive* specific attributes. This corresponds to prespecifying a pattern match that Fern carries out automatically. More practically, this automatic data propagation incurs significant processing an network overhead. Therefore, the user requests only the attributes they require. An entity specifies the attributes that it is interested in thus:

```
    (fern-perceive "color"
                   :react (lambda (entid attr-val)
                            (print "new color: " attr-val)))
```

Here, the entity has also supplied a react proc so that when any entity changes its color attribute, Fern invokes the callback function passing it the new data.

The second refinement is that, to be visible to each other, entities must be members of a common *space*. Entities that share a common space are called *siblings*. At any time, entities can enter or exit spaces in order to join or detach from these shared grouplespaces. Entities may be *entered* in multiple spaces at once, thus exchanging attributes with many entities of possibly disjoint spaces. This concept of spaces becomes very useful for multiple participant VEs where many environments may be simultaneously running. Spaces can be used to control the overlap between simultaneously running environments with respect to data exchange between the participant and the environments.

Space themselves are also entities. In fact, any entity can be a space. Sibling entities need only to *enter* another entity, a designated space entity, in order to begin sharing a grouplespace. Fern manages the automatic attribute transport between all sibling entities in the system. Primarily the space entity is a common reference point. In many applications an entity will create several new entities which can discover their creator entity using it as a convenient space. Fern uses the space entity's internal partition to manage the *subling* statistics.

The space entity requires no extra application code to act as a space. However, the space entity can, if desired, access the internal partition in order to actively interact with the subling entities in an application specific way. In this way, the space entity can implement properties that otherwise would have to be implemented in each entity in the space. For example, the effect of gravity could be implemented such that the space entity sends repeated streamed method calls to each entity to fall one unit for each message.

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 merely a data pool partitioning mechanism. Below is an example situation with a shared grouplespace.

~ ~ ~ ~ ~

Initialize begin the program with a spore entity:

```
(fern-init)
(fern-run "spore")
```

The spore entity creates two entities and waits to be a space.

```
(fern-new-ent "tic")
(fern-new-ent "toc")
```

Each entity enters their creator entity as a space:

```
(setq space (fern-copy.src))
(fern-enter space)
```

Each entity creates an attribute and subscribes the other attribute.

```
;; tic
(fern-put.attr '("tic" 1))
(fern-perceive "toc"
              :react (lambda (ent val)
                          (print "tic sees: " val)))
;; toc
(fern-put.attr '("toc" 1000))
(fern-perceive "tic"
              :react (lambda (ent val)
                      (print "toc sees: " val)))
```

Each entity creates a process to make changes to its boundary.

```
;; tic
(fern-persist '(progn
                (setq val (1+ (fern-copy.attr "tic")))
                (print "tic says: " val)
                (fern-put.attr `("tic" ,val)))
;; toc
(fern-persist '(progn
                (setq val (1- (fern-copy.attr "toc")))
                (print "toc says: " val)
                (fern-put.attr `("toc" ,val)))
```

The output.

```
tic sees 1000
toc sees 1
tic says 999
toc says 2
tic sees 999
toc sees 2
tic says 998
toc says ...
```

~ ~ ~ ~ ~

Fern provides a simple coherence mechanism for this shared grouplespace that is based on the same message flow control facility as streamed methods. 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 local entities. Some of these sibling will be local, 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 siblings entities, the grouplespace deltas are sent to the nodes on which those entities reside where they are incorporated into the siblings' externals.

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

This coherence protocol guarantees the following things. If an entity makes a single change to its boundary[42], the change will reach all subscribing sibling entities. Also, the last change an entity makes to it boundary[43] will reach its siblings. This protocol does not guarantee the intermediate deltas because Fern cannot control how many changes an entity makes to its boundary each frame, but it must limit the stack of work that it creates for interacting nodes.

Under well structured operating conditions, Fern's mechanism for attributes provides usable performance. This mechanism for sharing data can be used in combination with other communication mechanisms to achieve more deterministic behavior, but it is best used for propagating streams of data where missing intermediate values present no loss in correctness. An example of this circumstance is where some entity polls a position tracker using a persist proc. The entity may put the absolute position into a boundary attribute each time new data is available. Another entity may have a react proc installed to receive the position data for a data driven algorithm. Other entities may also enter the same space, subscribe to the position attribute, and do different things with the data, with no changes to any of the other entity code.

To tie all the Fern features together, Figure 1 provides a graphical overview of the Fern programming model.

---

[42] e.g. the first message in an sufficient amount of time.

[43] e.g. last message for a sufficient period of time.

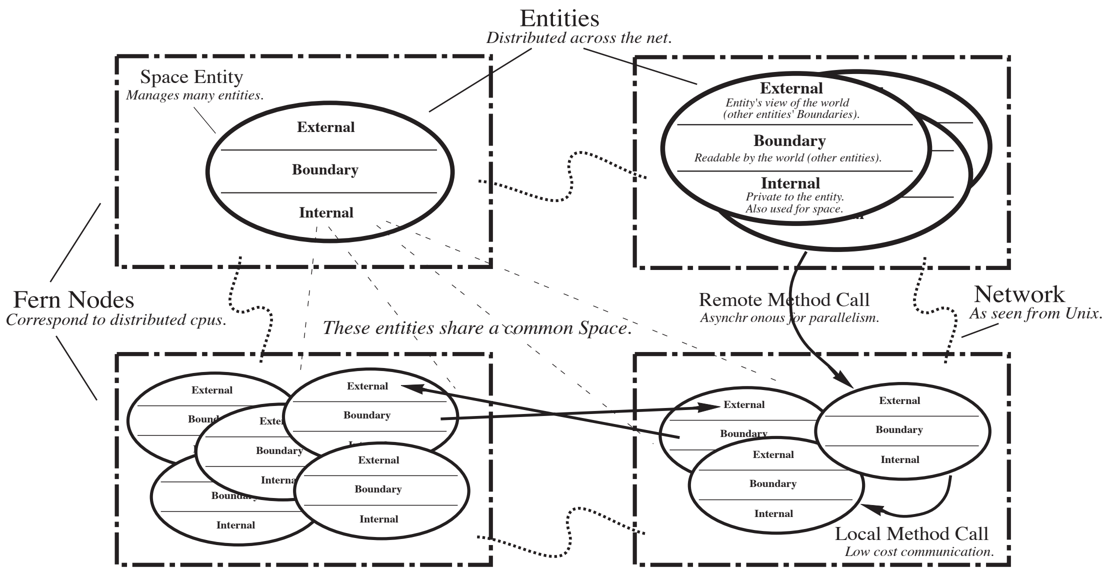
Entities
Distributed across the net.
Space Entity
Manages many entities.
External
Boundary
Internal
External
Entity's view of the world (other entities' Boundaries).
Boundary
Readable by the world (other entities).
Internal
Private to the entity. Also used for space.
Fern Nodes
Correspond to distributed 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 1: Fern Topology and Entity Structure

Fern Programming Practices

Because Fern provides so many different task decomposition mechanisms, the following idiomatic discussion is presented to suggest ways to use Fern services to fit the requirements of the particular application. First, there is a methodological issue to consider when programming Fern code.

The object-oriented nature of VEOS and of many other contemporary general systems encourages programmers to take the perspective of the object or entity that is doing the action in a program. In taking the perspective of the object, the programmer naturally employs their own style or aesthetic toward entity interactions. The style of entity interactions can be divided into two categories, *intentional* and *interpretational*. That is, interactions where an entity imposes or *intends* changes onto other entities, and interactions where entities *interpret* changes in other entities and make changes to themselves. Computationally, these styles are equivalent. One way or another, the code that moves data, passes messages, computes new values, and interfaces with hardware must still execute. However, certain interactions may be easier to model and reason about using one methodology rather than the other. At HITL, this issue was surprisingly significant, especially to non-programmers who brought a great deal of real-world thinking to their software design.

The strict object-oriented ideal embraces the intentional model, where objects simply send method calls to other objects, thus intending their commands. Reactive or responsive systems such as artificial life systems and cellular automata embrace an interpretational model, where objects change themselves because of internal disposition to changes that other objects make to themselves. In Fern, the varied primitives support both paradigms. In the object-oriented style, Fern methods support the intentional model. Fern perceive and react features support the interpretational model. The choice of is a matter of which methodology the programmer is more comfortable with.

Of course, programmers must choose to use certain features for other reasons than philosophical elegance. Other important criterion are performance, code modularity, and robustness in the face of non-ideal resource configurations. The performance section of Chapter 6 elaborates specific heuristics for achieving optimal application behavior with Fern.

Meta Systems of VEOS

After several revisions, the Fern interface to VEOS has reached a plateau of functionality and stability. The most common application-specific libraries such as SensorLib and the Imager have also reached a proportionate level of refinement. As VEOS became more accessible and better understood, HITL programmers and designers began to build significant applications based on VEOS. The commonalities of these applications were recognized as such and drove the construction of complete systems for doing specific tasks with VEOS. These systems are characterized by suites of entities which work together to accomplish some modular component of VE processing. As these packages are designed not to stand alone, but to work with many applications, they include well-defined interfaces consisting of known attributes, methods, and interaction semantics on which application programmers can rely.

Body & Wand

The first recurring application component was one that managed participant interactions in a consistent way across VEOS applications. This application component, initially implemented by Max Minkoff at HITL as a suite of entities, provides a virtual body though which the participant interacts with the VE. This module became known simply as the *body*. During initialization, any ordinary VE application can instantiate a body entity on an available node. In turn, the body entity instantiates its associated entities on the proper nodes depending on specific interface requirements such as inclusive display, screen based display, 3D sound, head tracking, or mouse navigation. The body entities load specific code to poll the proper devices and send data to either specified methods or to put the data into specified attributes. In any case the data stream interface is well defined and can be easily integrated into new applications.

One of the important components of the body module is the concept of the tool. The current tool represents the mode of interaction that the participant currently experiences. Through a known and well-defined interface, the application can choose or add new tools to the body. One tool that has become standard issue in HITL VEs is the *wand* tool. HITL engineers have developed a hardware wand which supplies position and orientation data as well as trigger and three button input. The software wand tool is the module that interfaces with the hardware wand and supports spatial navigation, teleportation, manipulating objects in the VE, and other custom application specific features.

The body suite of entities became central to nearly every subsequent VEOS application, and like other often used modules underwent significant evolutions [Body]. Andy MacDonald of HITL reimplemented the lower level mechanics of the body into the Mercury Participant System. With Mercury in place underneath, the body entities continued to supply virtual body services through the same entity interface. Only this time, the capabilities were implemented nearer to the hardware. VEOS users experienced a vast improvement in performance but only minors changes in the interface to the body module. Furthermore, Mercury is modular with respect to VEOS, and is poised to interface with general purpose computing systems other than VEOS.

Although VEOS programmers no longer use the implementation of the body composed entirely of generic VEOS constructs, that prototype provides an excellent example of a practical use of VEOS constructs. A simplified view of a body implementation is given. In abstract, the body continuously performs these fundamental functions:

1) Tracking the participant for movement and commands.
2) Integrating new input from (1) into the VE.
3) Collecting new output from the VE in preparation for (4).
4) Updating the participant's display.

With this overview, many implementations can be imagined. For example, each of these continuous tasks could be implemented as a separate persist proc, which exchange data asynchronously. On the surface, this strategy appears advantageous because it allows the potential for parallelism among the tasks. However, because the tasks are highly related, such independent processes may burn excess processing cycles performing duplicate work when the state has not changed.

A better approach uses a data driven methodology. Primarily, data flow initially emanates from autonomous entities and from participant input. All other data flows are simply the filtering or recombination of these initial data flows. Leaving aside the complication of autonomous entities for the moment, only the participant generates change (e.g. data) in the VE. Therefore, all computation not directly related to acquiring participant input can be idle until triggered by new data emanating from the participant. Hence, the data driven implementation uses entities with persist procs for tracking the participant's actions (1) and inputs, thus generating new data flows. While the other tasks (2), (3), (4) can be implemented as entities with react procs or methods, thus executing only when new data is present.

Directly following the arrival of new participant input, the body entities process these inputs in a number of ways. First, the body entities interpret the inputs as potential *commands*, such as for navigation through the VE or for directly manipulating objects in the VE. Second, the body interprets the inputs simply as incidental *movement*, which results primarily in updates to the database representation of the participant's virtual model; specifically, as the participant's head moves, the rendered viewpoint must change correspondingly, and as the participant's hand moves, the virtual representation of the hand must change also. Again, these tasks, summed by (2), occur as a direct result of data flow generated by sampling the participant.

Looking at the output side of participant interaction, the mechanics of tasks (3) and (4) are more complex. The complication of autonomous entities again arises. In particular, the participant display incorporates not only data emanating from the participant herself, such as new hand and head positions, but also other changes in the VE emanating from autonomous entities and other participants. In this context, autonomous entities and other participants are treated the same way because they can both generate new data in the VE database.

The important point is that the task of updating the participant display can be driven by data from other entities. Thus, (3) and (4) are implemented by body entities with react procs. React procs are preferred for these tasks because they provide a more flexible mechanism for responding to any kind of change in the VE. When the body entities use react procs, other entities that make changes in the VE[44] need not keep track of which entities to send their changes to.[45] Instead, participant body entities and autonomous entities belong to a common space and VEOS automatically propagates the data flow between entities.

The actual implementation addresses some further subtleties that are beyond this scope. For example, because (4) directly affects the participant's experience, the task of updating the participant display must perform steady and consistent processing. Consequently, the implementation takes measures to control the rate at which data flow arrives and integrates with the participant's view. Another subtlety that the implementation must address is that some changes in the VE are to be viewed *atomically*, that is, at the same moment in time. Measures are taken so that atomic data events remain atomic even if they arrive at different times.

UM

---

[44] e.g. generate data flow.

[45] as would be the case if using methods.

Many applications require direct mappings from participant actions to changes in the VE. Colin Bricken at HITL has designed a generalized system for building applications around these direct mappings called the Universal Motivator. The UM provides a graphical interface that allows users to specify functional relationships between attributes of various entities such as the participant's wand position and orientation, position, scale, orientation, and colors of objects in the VE. Users of the UM specify these relationships by parameterizing discretized two-axis graphs to form linear and non-linear functions. These functions are invoked in a data driven fashion, the data flow eminating from the participant (e.g. Mercury) or from autonomous entities elsewhere in the VE.

SPAM

Jeff James at HITL has designed a system with VEOS to address fundamental spatial interactions. The SPAM system (Spatial Perception And Movement) provides a structured specification environment based on entities. In abstract, SPAM ensures that each entity perceives the spatial world from its own perspective or frame of reference with respect to position, scale, topological relations, intersection and containment. The SPAM also provides a computational environment that manages collision events, containment hierarchies, space partitioning, and dynamic display control.

Antechamber

With the growing number of working VEOS applications and eager VE users, it became desirable to have a continuously running multiparticipant common ground where users could 'log in' and use different VEs simultaneously. This concept, called the *antechamber*, is currently under development at HITL. The antechamber makes use of Fern spaces to partition different VEs so that the participants can switch between a choice of running VEs. The antechamber also uses Fern's capability to merge and unmerge node pools in order to bring up and down participant and VE pools without disturbing other running participants and VEs. The antechamber itself is a VE that incorporates *portals* which participants use to enter annexed VEs. Participants may come and go in these antechamber VEs each sharing the various functionalities associated with the VE.

Chapter 5: Applications

In order to establish a foreground to the VEOS development, an overview of the areas to which VEOS has been applied is presented. In accord with VEOS's purpose of prototyping, most of these applications were proofs of concept rather than refined industrial grade applications.

Manufacturing

For her graduate thesis, Karen Jones worked with HITL engineer Marc Cygnus in developing a factory simulation application. The program incorporated an external interface to the AutoMod simulation package. The resulting VE 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* suite of entities for navigation through the simulation.

Communications

In the early stages of VEOS development, the interface capabilities of display and input suffered from lack of special purpose hardware. At that time, VEOS architects Geoff Coco and Dav Lion designed an application to demonstrate the conceptual capabilities of multiparticipant interaction and independent views of the VE. This VE, called Block World, allowed four participants to independently navigate and manipulate moveable objects in the same virtual space. With limited display resources, participants appeared as different colored blocks in each participant's screen based display. Block World allowed for interactions such as 'tug-of-war' when two participants attempted to obtain the same object. This application preceded Fern and building it helped the VEOS architects learn common VE tasks and pitfalls.

The most recent large scale application written in VEOS provided multiparticipant interactions in the forms of voice exchange and catch with a virtual ball. 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. This was mediated by the participants themselves in the first phase of the experience. Participants customized their personal view of the VE in terms of color, shape, scale, and texture thus encouraging a more subjective experience. Participants experienced the Catch VE two at a time, and could compare their experiences at runtime through spatialized voice communication. This voice interaction was novel because the spatial filtering provided each participant with additional cues about the location of the other participant in the VE.

Entertainment

Perhaps the most promising domain for VR is arts and entertainment. Ari Hollander has designed the VR equivalent of the classic parlor video game, Lunar Lander. This application employs existing VEOS code modules such as the body and vehicle code. The Lunar Lander application implements surface collision algorithms and simulates inertial forces in a low gravity setting.

Creative Design

Also in the category of arts and entertainment are applications designed for creative expression. Using the Universal Motivator system, Colin Bricken has designed several applications for purely creative ends. These VEs are characterized by many dynamic virtual objects which display complex behavior based on their autonomous dispositions and their reactivity to participant movements.

Education

Chris Byrne lead a program at HITL to give local youth the chance to build their own VEs. The program emphasized the cooperative design process of building VEs. These VEOS worlds employed the standard navigation techniques of the wand and many provided interesting interactive features. The implementations these kids generated reflected their inexperience, yet the themes were quite clever and showed natural insight into the possibilities of VR. It was interesting to note that none of the designs embodied battles or fighting contrary to the stereotype of popular kids' entertainment.

Perception

Most HITL researchers work in particular areas of technical development, each with an strong overall awareness of human interface issues. Daniel Henry, coming from an architectural rather than a technical background, sought an in-depth understanding of human perception with respect to interior space in VR. Daniel's Henry Art application closely simulated the Henry Art Gallery on the University of Washington Campus. The subsequent study involved comparisons of subjects' perceptions of size, form, and distance in both the real and virtual spaces. The Henry Art application used VEOS, and again the standard body module for navigation through the VE. Regrettably, insufficient tools existed at the time for non-programmers to implement the natural dynamics of collision with the rigid architectural structure. Consequently, the virtual gallery possessed unwanted inconsistencies to the real gallery.

Data Visualization

Data visualization is another promising domain for VR. Many applications were built in VEOS for visualizing large or complex data sets.

The first data visualization application was a satellite collected data set 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 more practical 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.

A pending application in conjunction with the Boeing corporation will attempt to visualize a laser and mirror approach for tracking the positions of factory workers.

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. Tour applications are easy to build because navigation modules such as the body can be employed with little modification. All that need be built is the interesting terrain or environment. These VE often feature autonomous virtual objects, but they do not significantly interact with the participant.

Examples of these types of applications built in VEOS were: the V22 tilt-wing 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, and the popular Metro application where the participant could ride the ever-chugging future train around a terrain of rolling hills and tunnels.

Simulation

Because physical simulations require very precise control of the computation, they have been a challenging application domain for VEOS users, many of whom are non-programmers. However, precise program control is achievable with many of VEOS's constructs. For their graduate Computer Science coursework, Geoff Coco and Dav Lion implemented a billiard ball simulation to demonstrate VEOS and measure VEOS's performance for a familiar application. In particular, to measure the tradeoffs between parallelism and message passing overhead [Async].

The simulation included fifteen billiard balls, each modeled by an entity. One entity provided an interface to screen based rendering facilities, another provided access to a spaceball input device, and another provided a command console. The balls communicated via asynchronous messages which triggered methods in the other balls to compute the next simulation step. The rendering and spaceball entities worked together much like the body suite of entities. The spaceball entity used a persist proc to sample the physical spaceball device and made changes to the graphical viewpoint. The imager entity updated the screen-based view after each viewpoint change made by the spaceball entity.

Asynchronous to the participant interaction described above, the ball entities continually recomputed their positions. The ball entities sent their new positions via method calls to the imager entity which incorporated the changes into the next view update. Within a frame, each ball, upon receiving updates from other balls, checked for collisions with the other balls. When each ball had received an update from every other ball (the end of the frame), it would compute a movement updates for the next frame. The granularity of time in this simulation was a frame, so all forces acting in a given frame are assumed to be coincident in time.

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 (method calls). 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.

Most of the entity code for this application was written in Lisp, except ball collision detection and resolution, which was written in C to reduce the overhead of the calculations. Samples from the application code is shown below.

~ ~ ~ ~ ~

```
;; Spore entity
;; this entity loads the other entities and
;; initializes them to random values.

(setq spaceball
      (fern-new-ent "spaceball" :run-host home))
(setq imager
      (fern-new-ent "render" :run-host "iris2"))
(defun new-ball (run-host)
      (fern-new-ent "ball" :run-host run-host)))
(setq balls (mapcar 'new-ball run-hosts))

(dolist (ball balls)
      (random-init-vals pos vel color)
      (fern-send (car b-list) "init-ball"
                balls pos vel color imager))
```

~ ~ ~ ~ ~

```
;; Imager entity
;; this entity provide an entity interface to
;; the rendering hardware.

;; setup imaging environment,
;; complete with tabletop and grid
(im-imager-init)
(im-window-view 1000)
(im-set-void-color 0.0 0.2 0.8)
(setq the-wall (im-new-object self "cube"))))
(im-scale-object the-wall #(11.0 0.1 8.5))
(im-xform-object the-wall
       #(1.0 0.0 0.0 0.0
         0.0 1.0 0.0 0.0
         0.0 0.0 1.0 0.0
         0.0 0.0 0.0 1.0))
(im-new-object "grid")

;; define methods for working with graphical objects
(fern-def-meth "picture" (lambda (ent attr-val)
                       (im-new-object ent attr-val)))
(fern-def-meth "color" (lambda (ent attr-val)
                     (im-color-object ent attr-val)))
(fern-def-meth "pos" (lambda (ent attr-val)
                     (im-xform-object ent attr-val)))

;; define method for updating participant view
(fern-def-meth "eye" (lambda (mat)
                     (im-xform-view matrix)))
(fern-def-meth "update" (lambda ()
                     (im-drawframe)))
```

~ ~ ~ ~ ~

```
;; Spaceball entity
;; entity interface to the physical spaceball
(sball-init "/dev/ttyd2")

(defun trak-view ()
   (sball-read-matrix m)
  (fern-send imager "eye" m)
  (fern-send imager "update")
  )
(fern-persist '(trak-view))
```

~ ~ ~ ~ ~

```
;; Ball entity
;; this entity definition is loaded n times.
;; no persistent process, only responsive process.
;; spore entity sends init to get things going.

;; create a private lisp context for each ball
(let (pos col vel count)

(fern-def-meth "init-ball" (lambda (ball-entids
                                    init-position
                                    init-velocity
                                    init-color
                                    imager-entid)
      ;; save list of other balls
      (setq balls (all-but ball-entids self))
      (setq num-balls (length (fgetvar balls)))
      (setq count 0)

      ;; store initial vals
      (setq pos init-position)
      (setq vel init-velocity)
      (setq col init-color)
      ;; setup graphical representation
      (setq imager imager-entid)
      (fern-send imager "picture" self "sphere")
      (fern-send imager "pos" self pos)
      (fern-send imager "color" self col)

      ;; kick-start the simulation
      (send-notify)
      )) ;def-meth

(defun send-notify ()
      (dolist (ball (all-but self balls))
           (fern-send ball "notify" self pos vel))
      (fern-send imager "pos" self pos))

(fern-def-meth "notify" (lambda (entid pos vel)

      (setq count (1+ count))

      ;; check for collision (call to C primitive)
      (collide pos vel other-pos other-vel)

      ;; check for last update this frame
      (cond ((equal count num-balls)

           ;; compute next movement (call to C)
           (make-move pos vel)

           ;; send out updates
           (send-notify)
           (setq count 0)

           )) ;cond
      )) "notify" method
```

```
) ;let context
```

Chapter 6: VEOS Usability Analysis

In designing VEOS, the general goals of simplicity and performance often undermined each other. Roughly, the simpler the programming model, the poorer the expected performance for many situations. This correlation occurs because a simple programming model may be overly generalized because it must make many assumptions about usage patterns. Consequently, the simple interface is not easily parameterized or tailored for the varying profiles of needs of individual applications. The other side of the tradeoff is that a highly sophisticated language allows the programmer to specify every operational detail, and therein lies the potential for greater efficiency.   But, the sophisticated language is inherently accessible to fewer users because it requires more training, more facility with the medium, and encourages the programmer to focus on the ways rather than the means.

In short, this balance can be summed as the tradeoff between mental workload and computational workload.[46] This chapter discusses VEOS's effectiveness with respect to the goals of accessibility to a broad range of VE designers and usable performance for VR typical applications.

Ease of Prototyping

Skill and Knowledge

Aside from VEOS constructs, the Lisp language itself attempts to cradle the shy programmer. For a linear programming language, Lisp is relatively forgiving. For example, one of the messier parts of programming is memory management. Lisp makes fair compromises in terms of memory management. If desired, the Lisp programmer can almost completely ignore memory allocation issues because Lisp incorporates garbage collection and abstract data primitives such as list, car, cdr. But, as is the philosophy of VEOS, Lisp also allows the ambitious programmer to achieve greater efficiency and

[46] Of course, this tradeoff presupposes a zero-sum problem. It is possible, in theory, that some simple yet expressive language could provide an inherently efficient computational regime yet requiring little skill by the programmer. Based on the history of programming regimes to date, this 'all-win' situation is unlikely.

smoother performance through a deeper understanding of the underlying memory structure.

Another way Lisp encourages the beginner is through runtime interaction with the Lisp system. In debugging a stopped program, the user can engage in a dialog with the Lisp environment to determine exactly what went wrong. Furthermore, that Lisp is interpreted removes the monotonous compile step from building programs and focuses the programmer's attention toward runtime behavior.

VEOS provides multiple levels of features for a range of programmers. Together with the base of standard modules, Fern's shared grouplespace services provide the beginner with simple mechanisms to build their first VE.[47] As programmers begin to understand the mechanics of distributed programming and begin to discriminate toward performance and correctness, they can learn the object oriented mechanisms which provide more subtle program control. Finally, when programmers fully understand the combination of primitive resources VEOS provides, they can devise their own task decomposition schemes, often in a lower level language. This progression has replayed itself again and again at HITL, where new students or visiting researchers begin as novices and develop to a comfortable level of understanding.

A VEOS feature of particular note is the persist proc which has proven appealing both to the beginner and the experienced programmer. The function and behavior of persist procs are easy for the novice to grasp, and yet they provide enough flexibility for the advanced programmer to implement sophisticated algorithms.

However, not every aspect of VEOS caters to the novice. Curiously, the two features which contain hierarchical components have both proven to be confusing. First, the Fern space. Novice programmers understand the premise of a space. But they are often misguided by the fact that any entity can be a space, which can be in a space, and so on.

The other feature is the Nancy pattern matching language. Again, the hierarchical nature of the grouplespace and hence the pattern language was a stumbling block even for

[47] In fact, mini courses at HITL entitled, "Build VR in a Day" emphasize these features for beginners.

experienced programmers. Furthermore, the Nancy pattern language was designed to be succinct, so much so that its form became cryptic. Complex Nancy pattern expressions wind up containing long sequences of symbols which, to the unpracticed eye, appear arbitrary.

Early in the development of Fern, this difficulty with the Nancy pattern language was discovered. Consequently, Fern provides abstract wrappers so that programmers are never required to write patterns. For ambitious programmers who require custom grouplespace usage, Fern also provides primitives that circumvent these high level abstractions (see Appendix B).

Perhaps the most elusive bit of understanding both for novice and experienced VEOS programmers is that of task distribution. There are many interrelating factors that determine the best strategy for building a distributed VEOS application. Among these factors are effective processor speeds, access to special devices or capabilities, and task communication patterns. Often, VEOS programmers concentrate much of their optimization efforts on simply restructuring the distribution.

Modularity and Reuse

A critical ingredient in a rapid prototyping system is the ability to reuse code and modules. Like objects in a pure object systems, Fern entities encourage modular programming. Whether the programmer chooses to uses object oriented methods or shared grouplespace constructs, well-defined interfaces can be established between entities and suites of entities so they can be used interchangeably. Even with the most abstract interfaces, there are certain caveats for modularity in VEOS.

First, entIDs. During runtime, the entIDs of communicating entities must somehow be passed around the program before the program can use Fern communication services. The interface protocols of Fern modules usually account for this bootstrapping phase in ways mentioned in Chapter 4.

Second, location-dependence. Ideally, Fern entities can run and function the same way in any distributed configuration. Indeed, the semantics will remain the same. In practice however, it is often useful in Fern code to assume the relative locations of entities in a program. And changing these relative locations will result in poor or incorrect performance. For example, if two entities can be assumed to be on the same machine, some communications between them may use synchronous semantics - essentially a local function call. If this same code were to execute when the entities are remote to each other, the same synchronous operation will unexpectedly block the entire node while the function call executes on the remote node.

The logical extension of modularity is code reuse. Object classes such as in Smalltalk provide a common mechanism for code reuse. Classes provide general functionality, and need only be implemented once. Subclasses provide further elaborations or alterations to the class. During runtime, the program creates instances of these various classes that function according to the class definition. Fern entity definition code, if organized properly, can be used in this way. The (subclass) entity definition can include other (superclass) entity definitions in order to inherit the characteristics of those entities.

There are other components of VEOS programs that can be readily reused such as graphical data. Many VEOS application use the same geometry information for a wand, grid, space needle, cube, and rabbit. Texture information for water, marble, moonscape, and star map are also used again and again.

Development Time

VEOS successfully speeds the prototyping process by offering features for task decomposition, configuration of VE logic, debugging and rerunning programs, and reuse of parts. However, there are some aspects of VE development that still require the careful and diligent attention of experienced implementors. For example, even using VEOS constructs, standard modules for participant interaction must be well-designed and robust since they will be used again and again. Another example is modeling new graphical objects or building new banks of application sounds. Especially for complex VEs, a good deal of storyboarding goes into the design before the implementation.

A valuable lesson in this regard was that rapid prototyping is only partially achieved through a simple programming model. This must be complimented with the ability to reuse working pieces of data and program modules. In any case, VE design requires the efforts of many people, each with expertise in different areas.

Performance

Although high performance was not a top level goal in the VEOS design, performance is always a practical concern for application designers. As such, the VEOS implementation incorporates significant optimizations that did not significantly compromise the design. Even with these optimizations, the inefficiency of Lisp compounded by VEOS's generality imposed limitations on the maximum performance achievable with VEOS. As mentioned earlier, specific performance-critical components of applications are often prototyped with generic VEOS constructs, and later rewritten in C. Such was the case with the Mercury participant system.

System Benchmarks

As most prototype application code and much of Fern is written in Lisp, the performance of VEOS applications depends heavily on the efficiency of the XLisp implementation. Below is a benchmark that compares the execution speeds of XLisp and C implementations of fibonacci calculations on a MIPS R3000 processor running DEC Ultrix. As shown by the critical code components, both implementations employ the same program structure. Both benchmarks were run in the most optimal way available. The C version was compiled with full optimizations, and the Lisp version was run in an custom fast evaluation mode. The VEOS version of XLisp supports a mode where the evaluator does not make its usual costly system calls to save every evaluation context.[48] This optimization trades execution speed for the runtime capability to recover and debug after Lisp evaluation errors.

The C implementation.

```
int fib(int x) {
 if (x <= 2)
      return (1);
 else
```

[48] Unix calls to setjmp(...) and longjmp(...) .

```
        return (fib(x-1) + fib(x-2));
 }
...
for (i=0;i<trials;i++)
 printf("%d\n", fib(i));
...
```

The Lisp implementation.

```
(defun fibonacci(x)
 (if (<= x 2)
      1
    (+ (fibonacci (1- x))
       (fibonacci (- x 2)))))
...
(dotimes (x Trials)
      (print (fibonacci x)))
...
```

Table 1: Fibonacci Performance

run of C implementation:	*run of Lisp implementation:*
`% fib 25`	`> (fib 25)`
`1`	`1`
`1`	`1`
`1`	`1`
`2`	`2`
`3`	`3`
`...`	`...`
`46368`	`46368`
`total: 0.117180 seconds`	`total: 26.95 seconds`

This benchmark stresses stack usage, loop speed, and integer arithmetic. Not surprisingly, this simple test reveals that users can expect their C implementations to execute two orders of magnitude faster than corresponding Lisp implementations.

As for VEOS system services, these include creating and disposing entities, sending messages between entities, and grouplespace access. The following benchmarks show statistics for these basic services under best and average operating conditions. The tests were run across a pool of three nodes, two MIPS R3000-based workstations running DEC Ultrix, and one Sparcstation 4/110 as the console node.

Entity Primitives

Among others things, entities serve as a context for VEOS persist procs. As a comparison to starting and stopping a null thread in a typical thread system, an entity with an empty persist proc is created and disposed. Entities are created, then immediately disposed from the spore entity. The same code was used in each of four trials: to create entities (1) local and (2) remote to the spore, and in each case, the node receiving the new entity was (1) free of any other process commitments and (2) sustained a typical process load. For this experiment, a typical process load consisted of seven other entities already residing on the node each with a non-trivial persist proc that performed a small Lisp computation.

Table 2: Entity Performance

all times in milliseconds	local no load	local typical load	remote no load	remote typical load
create latency	13.1	13.6	48.9	62.8
dispose latency	1.2	1.2	1.1	2.7
throughput	40.8	33.9	19.8	15.0

As mentioned earlier, during entity creation, Fern evaluates the entity definition code in the new entity's context before control is returned to the caller. As such, these times represent the absolute minimum overhead for entity creation since entities typically contain more than a trivial persist proc.

Entity creation is a synchronous operation because the caller needs to know the entID of the new entity. In addition, the caller needs to know when and if the operation succeeded. Because creating an entity is a synchronous operation, remote entity creation times reflect the increased overhead of network transport and synchronization with the remote node. Entity disposals, however, are asynchronous since it is less critical when the operation completes. Correspondingly, disposal times reflect only the time to dispatch the disposal request to Fern.

As shown, remote creation performance suffers heavily from competition for the receiving node's processing time under typical load conditions. This is expected because the synchronous creation request waits in the remote node's message queue until the beginning of the next frame when the request is processed and a reply is sent back to the waiting node. When the remote node's processing load is greater, the frame rate decreases and the wait in the message queue is longer.

Message Passing

VEOS provides multiple mechanisms for communication between entities. All of these mechanisms rely at some level on the VEOS Kernel message passing primitives. The Fern object-oriented message passing mechanisms use the Kernel message passing facilities most directly and so best reflect the native communication capabilities of VEOS.

All the following communication tests were implemented with round-trip semantics in order to accurately measure performance from a common frame of reference. Fern's synchronous message primitive incorporates round-trip semantics inherently. Although VEOS programs rarely use synchronous message passing semantics, testing them reveals the minimum round-trip time possible with the VEOS message passing implementation. Since most VEOS programs use asynchronous semantics, an asynchronous send-reply algorithm is used to achieve round-trip semantics. The significant parts of the test implementation is shown.

~ ~ ~ ~ ~

Code from client entity using synchronous semantics.

```
(defun bind-to-server (server-host)
  (setq server
  (fern-new-ent "sync_server" :run-host server-host)))
```

Client uses seq-send for synchronous semantics.

```
(defun client-stub (data)
  (fern-seq-send server "server-stub" data))
```

Code from server entity, designed for synchronous reqeusts. Does nothing but return data argument.

```
(fern-def-meth "server-stub"  (lambda (data) data))
```

~ ~ ~ ~ ~

Code from client entity using asynchronous semantics. The test algorithm installs its measurement function here; it also makes next call to client-stub.

```
(defun bind-to-server (server-host user-func)
  (setq server
     (fern-new-ent "async_server" :run-host server-host))
  (fern-def-meth "rpc-catch" (lambda (data)
                (eval (list user-func data)))))
```

Client uses send for synchronous semantics.

```
(defun client-stub (data)
  (fern-send server "server-stub" self data))
```

Code from server entity, designed for asynchronous reqeusts. Immediately sends asynchronous reply back to client.

```
(fern-def-meth "server-stub"  (lambda (ret-ent data)
                (fern-send ret-ent "rpc-catch" data)))
```

~ ~ ~ ~ ~

Table 3: Minimum Message Performance

all times in milliseconds	local no load	local typical load	remote no load	remote typical load
sync latency	4.8	4.9	9.0	19.8
async latency	6.1	8.2	11.0	29.5

As expected, local synchronous message passing is the most efficient partly because no network data conversion is necessary and partly because program control is passed immediately to the receiving method much like a local function call. Local asynchronous semantics take longer because both the request and the reply messages are posted to the node's message queue, and then evaluated at the beginning of the next frame. For this test, a typical load is implemented with seven other entities each running a non-trivial persist proc. Because message evaluation is interleaved between persist proc evaluation, the local asynchronous test shows a decrease in efficiency under a typical load. Whereas, the local synchronous test is not affected by load because the function call happens immediately without using the message queue.

In the remote cases, requests wait on the remote node's message queue. Thus, both synchronous and asynchronous cases are affected by load differences. In the remote synchronous cases, the local node spin-waits for replies on the requesting entity's behalf, thus yielding lower latencies than the asynchronous cases.

The above tests reveal basic system performance measured by sending one message at a time. However, VEOS programs also tend to send continuous streams of messages. In addition to latency, throughput performance is also important in this context.

VEOS program's use a flow-control semantic when sending continuous messages in order not to overburden other nodes. The user modulates this flow-control mechanism by specifying the outgoing *stream-width.* The stream-width represents the maximum number of outstanding messages allowed in the logical 'pipeline' between the sending node and any other node. For instance, if the stream-width is set at 2, two messages can be sent, but a third message cannot be sent until the first has been digested by the remote node's low-level message handler. This flow-control parameter strongly influences the behavior of stream-type asynchronous message passing.

The algorithm used to benchmark continuous message passing includes the same asynchronous server entity as above, but implements the client entity with a slight change. The client entity contains a persist proc that constantly tries to send messages to the server. These request messages are paced only VEOS's flow control mechanism. This algorithm keeps the number of outstanding requests approximately equal to the stream-width, producing a pipeline effect and maximizing throughput.

~ ~ ~ ~ ~

This time the client uses str-send to pace messages.

```
(defun client-stub (data)
  (fern-str-send server "server-stub" self data))
```

The client sends messages as fast as the local frame rate will allow.

```
(fern-persist '(client-stub (read-total-time)))
```

~ ~ ~ ~ ~

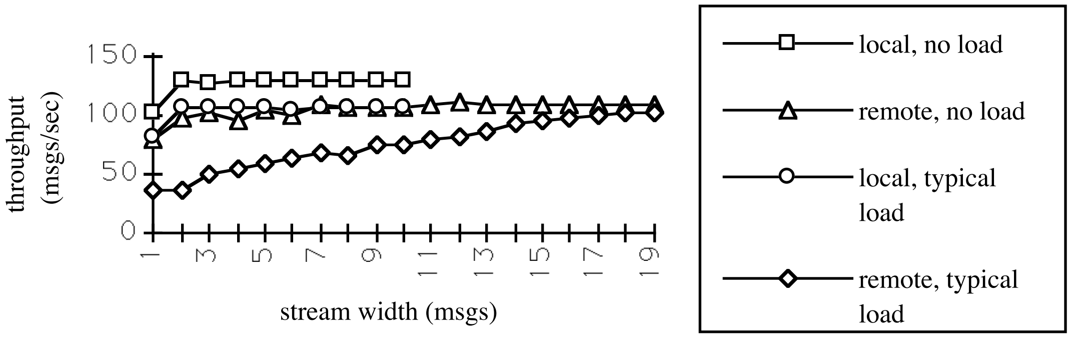

Figure 2: Message Throughput using Asynchronous Streams

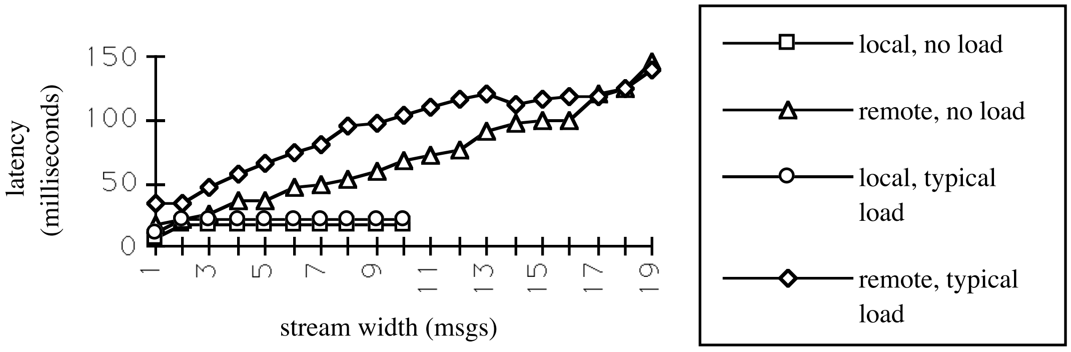

Figure 3: Message Latency using Asynchronous Streams

As in the base tests earlier, local performance is better on the whole than remote. Furthermore, local performance is only slightly influenced by flow-control. VEOS uses the same general flow-control algorithm for local message passing to prevent local asynchronous messages from piling up. However, because this algorithm was implemented with a persist proc, it is self regulating in the local case. That is, the persist proc generates at most one message each frame and the node can easily digest at least one message per frame. Note that the throughput for local messages increases to a plateau when the stream-width becomes 2. This is because the request and reply messages both occur on the same node each frame. Thus, a stream-width less than 2 prevents these messages from pipelining.

As for remote performance, the expected pipelining effect is shown. The greater the number of outstanding messages, the greater the dispatching efficiency. This effect appears to reach a point of diminishing returns especially in typical load cases. This can be seen where the latency continues to rise, while the throughput tapers off. Beyond a certain point, increased messages only incur a greater queuing overhead while messages must wait longer in queues.

What these tests do not show is that although a large stream-width can yield higher message throughput, it can also lead to sporadic transmissions. This circumstance, called *convoying*, is characterized by a entity sending multiple messages in rapid succession whereby filling the stream. Meanwhile, because the throughput is increased when many messages are queued, the receiving entity handles multiple messages in rapid succession. By this time, the sender notices that the stream is not full and rapidly sends enough messages to fill the stream again. Unless the application employs a sophisticated pacing algorithm, this effect is likely to occur.

Convoying is only a problem when smooth data transmission is required by the application, as is often the case in VE applications. In these cases, programs usually set the stream-width to a small number, thus trading greater throughput for smoother transmission.

Grouplespace

VEOS grouplespace access is tested in two ways. First, a single entity performs repeated operations to measure local pattern matching access speed. Second, two entities pass data between across the shared grouplespace using the same semantics as with the message passing benchmarks. The following are the important calls for the local tests.

This call adds new data to the calling entity's boundary.

```
(fern-put.attr '("hungriness" 6.7))
```

This call retrieves data from the calling entity's boundary.

```
(fern-copy.attr "hungriness")
```

Table 4: Local Grouplespace Performance

all times in milliseconds	no load	typical load
put latency	1.3	1.5
copy latency	1.0	1.2

For this test, typical load conditions were represented by seven other entities. Processes belonging to other entities could not influence these times since they were performed atomically so that no other process could intervene. Simply the existence of other entities on the node contributes to a higher load for pattern matching. This is because there is one grouplespace for the entire node and it is partitioned for each entity on that node. Each new entity on the node means another data item in the top level grouplespace.[49] The effect of more entities is shown by the second column of data.

Like the baseline tests shown above for message passing, the following tests reveal the minimum latency achievable when passing data across the shared grouplespace. The algorithm passes one message at a time as the entity's request attribute. Again, round-trip semantics are used in order to obtain accurate measurements.

~ ~ ~ ~ ~

Client entity code. Any entity can be a space, this algorithm uses the client entity as it is most convenient. This entity perceives reply attributes.

```
(defun bind-to-server (server-host user-func)
  (fern-enter self)
  (fern-perceive "reply" :react (lambda (entid data)
                (eval (list user-func data))))
  (fern-new-ent "gspace_server" :run-host server-host)
  )
(defun client-stub (data)
  (fern-put.attr `("request" ,data)))
```

[49] The single data item is a grouple containing the entity's entire structure.

The server code. This entity enters the client (also its parent) as a space. This entity perceives request attributes.

```
(fern-enter (fern-copy.src))
(fern-perceive "request" :react (lambda (entid data)
                     (fern-put.attr `("reply" ,data))))
```

~ ~ ~ ~ ~

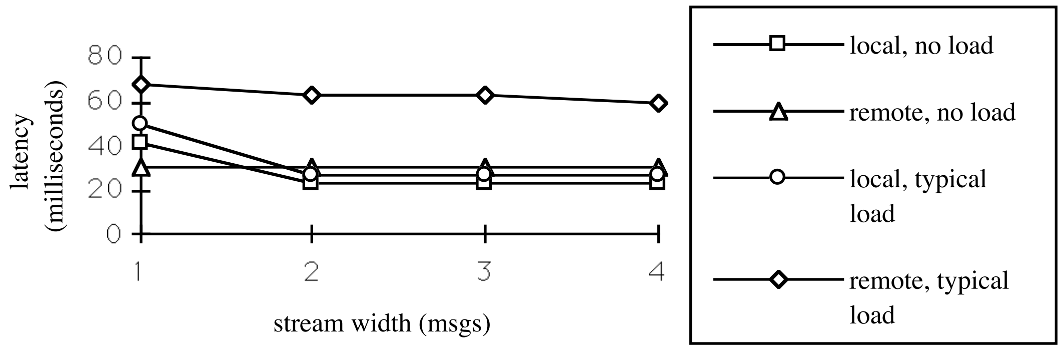

Figure 4: Minimum Data Latency using Shared Grouplespace

The shared grouplespace is implemented using the same flow-control mechanism as with stream messages. Consequently, the same performance increase occurs in the local case when the stream-width becomes greater than 1. Again, this testing algorithm is self regulating and so does not otherwise benefit from the increased stream-width. However, the next benchmark displays a strong response to this flow-control parameter.

In the following tests, the client makes requests by changing the request attribute as often as possible using a persist proc. Like with the streaming message tests earlier, these requests propagate to the receiving entity when the stream is not full. The only major change in the test implementation is that requests are made repeatedly from within a persist proc rather than one by one after receiving the previous reply.

Client entity code.

```
(fern-persist '(client-stub (read-total-time)))
```

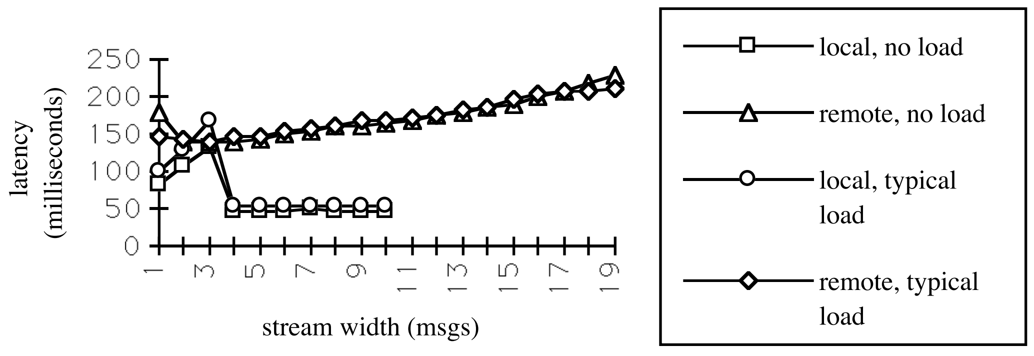

Figure 5: Data Latency using Shared Grouplespace

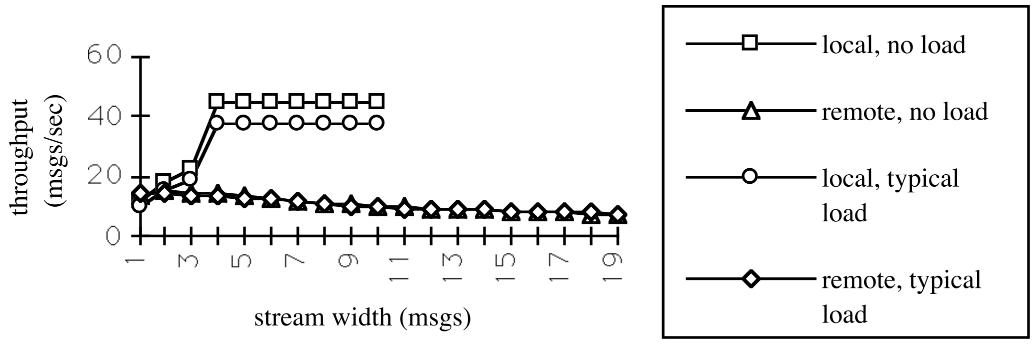

Figure 6: Data Throughput using Shared Grouplespace

Again, local performance is most efficient and on the whole is not affected by stream width. However, some interesting artifacts of the shared grouplespace coherence implementation are visible in the local cases. This is most likely due to the fact that the coherence mechanism requires additional messages for it's general algorithm. These messages apparently influence performance in the small stream-width cases. Also apparent is that process load is less significant in shared grouplespace performance. Perhaps this is because the shared grouplespace is inherently less efficient than direct message passing. And that the same process load variations which influenced message performance are lost in the noise of grouplespace performance.

Performance Implications

Although overall VEOS performance does not compete with sophisticated commercial operating systems, there is sufficient motivation to use VEOS for design.

Given VEOS's overall utility, it makes sense to use VEOS in the most optimal way possible. Below are some implicit guidelines for building application code. Fern application code that follows these principles can expect the most responsive performance and the most flexibility in application structure.

• Non-blocking.

All application code whether the entity definition, persist procs, methods, or react procs should take a finite amount of time, independent of inputs and state. In other words, none of these code forms should wait indefinitely for input, network replies, etc., unless the application is designed specifically to tie up whole nodes at a time.

• Polling semantics.

From the Lisp application level, data should emanate from polling. All other application data is a product of filtering and composing polled data. Low-level data generation code (e.g. for device drivers) is often implemented using interrupts, but the Lisp wrappers to these data producing primitives support a polling semantic. That is, when these Lisp primitives are called, they percolate the latest results out to Lisp. Every VEOS-compatible application specific primitive library supports this polling semantic. This principle works hand in hand with the non-blocking principle from above.

• Short duty cycle.

Another variation on the themes already mentioned is the idea of discrete operations. Code blocks should perform their function in short bursts, even if it means postponing less important work for later bursts. The reasoning behind this task discretation is that Fern must schedule all tasks each frame: method calls, persist procs, and react procs; hence, smaller task grain sizes lead to smoother perceived concurrency. For example, applications can incorporate garbage collection schemes to minimize immediate resource overhead. As another example, upon receiving program control, code blocks should do the most urgent work immediately, then make some asynchronous calls to queue other parts of the task until later, especially in the case of synchronous method calls.

• Check for necessity.

A good programming practice in any domain, application code should check the relevant inputs to see whether further work will be wasted. For example, if the source data is stale from last frame, especially with persist procs, the code can exit immediately giving the precious processor back to the next waiting task. This principle is not as relevant, when using data driven methodologies[50] since those code blocks are invoked upon the arrival of new data.

• Cache intermediate results.

This principle complements the above principle of breaking down long computations into several discrete tasks. In these situations, state information must be carried across evaluations of persist procs, or invocations of asynchronous methods. Application code should use fast caches that keep intermediate results handy across context switches.[51]

• Robust.

Since all local application tasks run on the same stack and in the same Lisp environment, it is especially important that application code be defined for all possible inputs. Furthermore, that all possible outputs are reasonable as defined by the rest of the application.

• Use the right metaphor.

For parts of an application requiring highly responsive behavior, the response code should reside on a particular node that has no continuous computation. This ensures that Fern is best prepared to receive messages and dispatch them to the right entity with minimum latency. Also for minimum latency, response code should and use event driven mechanisms like react procs or methods. For the parts of an application requiring steady computation, the code should be written to use paced mechanisms like persist procs and stream communication. Consistent computation (e.g. steady frame rate) is achieved when the nodes running the cyclic compute constructs[52] remain independent of bursty nodes.

For data driven algorithms where new computations are initiated by new data, the code should be structures around react procs, provided that dropped data items is acceptable. For more precise algorithms where every event is important, methods can be used with their varying semantics. For code modularity, the shared grouplespace and

[50] Methods or react procs provide data-driven primitives.

[51] Caches can be implemented in Lisp memory, or with C level hash tables.

[52] Cyclic constructs are persist procs or self-sustaining asynchronous method calls.

react procs provide a mechanism that allows trivial reconfiguration. Of course, Fern's object-oriented features support modular program structure as well.

Although entities and their associated data and process constructs are relatively lightweight, Fern applications must be written at the proper scale to effectively utilize these constructs. For example, for a simulation of autonomous creatures where there are a small number of complicated components, it is reasonable to decompose the problem so that each actor is an entity. Whereas in a simulation of a thousand bees, it would be more effective to decompose the problem such that the swarm was one entity that used smaller constructs internally to represent the individual bees.

• Localize communication.

As will be shown later, each network communication is a very costly operation. Entities that communicate consistently will do so most efficiently when located on the same node. While heeding to these other concerns, such as limited cycles and smooth versus bursty processing, care should be taken to distribute entities in such a way that the network is used as sparingly as possible.

This network constraint naturally encourages coarse-grain distribution strategies. Whereas, fine-grain parallelism cannot be practical unless interprocess communication is highly efficient. As shown by the performance tests above, local entities are at liberty to communicate more often and pass more data between them.

Resulting coarse-grain strategies include designing abstract low-bandwidth interfaces between remote entities. Although the VEOS interface provides a parallel programming model through entities, the only true parallelism occurs between VEOS uniprocessor nodes. Therefore, while entities provide local task structuring, nodes provide the true commodity of computation and hence coarse-grain parallelism.

An example of coarse-grain strategy involves device interaction. Some VE application may require input from two peripheral interface devices, voice input and wand (or glove) input. For the a compelling VE interface, this task may contain filters, cross-modal interactions, smoothing, prediction, etc. The suite of entities that handles this component of the VE could reside on its own node and perform consistent internal computation while passing only abstract pertinent event-driven messages to the rest of the application.

Chapter 7: Future Directions

During the development of VEOS, the VEOS architects learned a great deal more about the nature of the problem that VEOS attempts to address. Namely, that of rapid prototyping computationally intensive VR applications in a distributed environment.

Other Platforms

There is strong interest in porting VEOS to other platforms primarily because of its programming model rather than its performance or high utilization. In any case, desktop platforms such as PCs are often situated in LAN configurations which makes them good candidates for VEOS platforms. However, these platforms often do not possess the same processing power of workstations.

A more promising direction is that of true multicomputers. VEOS could be retrofit to multiprocessors in a number of ways. The VEOS application could treat each processor as a VEOS network node. Strictly mapping processors one-to-one to VEOS nodes in this way would utilize VEOS internal process services, but would forfeit any benefit from hardware shared memory support. Or, Fern could be extended such that the shared grouplespace were implemented through shared memory.

Another way to apply VEOS to multiprocessors is for the application to treat the entire multiprocessor as one network node and take advantage of multiple processors through application specific libraries. Additionally, Fern could be extended to map entities or persist procs directly to processors. With any of these approaches to multicomputers, much of the need for distributed computing still remains. Thus, VEOS would retain its generic networking capabilities for wider area applications.

Architectural Improvements

After several applications were developed with VEOS, the architectural deficiencies became apparent. While keeping the central VEOS design, many implementation changes would significantly improve VEOS's performance.

Kernel

A primary inefficiency in VEOS pattern matching is data conversion. Because the Kernel primitives provide a Lisp interface, Nancy patterns are first specified in XLisp's native list structure, but then are converted to the VEOS Kernel's internal grouple format each time a put, get, or copy is requested. The results of these operations initially take the form of the Kernel grouple format which must again be converted into Lisp as return values from the Kernel Lisp primitives. In addition to the heavy runtime cost of data conversion, this incompatibility between Lisp and the internal grouple format prevents direct access to the grouplespace data structures. In particular, the current implementation of (vcopy ...) makes a independent copy of the result of the matching operation. Ideally, the grouplespace would be implemented in the native XLisp data structures, and the copy operation would return a Lisp pointer to the actual matching data in the grouplespace. Then, the user could retain the pointer, inspect the data, and potentially modify it without incurring the repeated overhead of matching and copying.

Another limitation of the current grouplespace implementation is that elements within any given grouple remain in the position that they were initially inserted. Consequently, pattern matching cannot take advantage of sorting and so uses a linear search inside each grouple where content specific matching is requested.[53] As shown earlier, this algorithm performs poorly when grouples contain more than a handful of data elements. More sophisticated database techniques could assist in retaining sorting information, while also keeping ordinal information.

A recurring complaint of VEOS is the lack of explicit control of timing of various events. This attention to determinism is most appropriately addressed by the Kernel because timing issues apply to all Kernel operations. One of the first problems to address toward determinism is VEOS's inherently asynchronous communication model.

VEOS employs a point-to-point communication model at the Kernel level for simplicity. At a practical level however, point to point semantics undermine the shared medium property of ethernet. Of course, typical network controllers[54] do not interrupt the host computer when messages destined for some other node pass on the ethernet. But,

[53] Content specific Nancy patterns contain ** expressions.

[54] such as those embedded in most workstations.

during the time that message is being transmitted, the ethernet is effectively unavailable. This warrants a more efficient use of network transmission time. Perhaps a multicast model where more host computers and thus VEOS nodes are involved in each network communication, in expectation that the total number of transmissions would decrease. Such an implementation could still use standard Unix services such as UDP and could conceivably support an improved shared grouplespace.

Fern

As a distributed object system, Fern would benefit from many of the features that Emerald provides. Specifically, runtime mobility of entities for dynamic load balancing and communication optimization.

Without any loss in generality, Fern could provide an truly separate grouplespace for each entity, rather than partitioning the node's only grouplespace. This simple modification would greatly improve matching performance because the grouplespace context is already implicitly specified by the calling entity.

Fern uses some poor performing algorithms because they were simple to implement. These include the flow-control mechanism and the shared grouplespace coherence algorithm. The flow-control algorithm could be improved with attention to smooth data transport and alleviation of the convoying problem. This improvement would implicitly contribute to better shared grouplespace performance because the coherence protocol is based on flow-control.

Next Generation

Although the above modifications would lead to improved performance, many inherent limitations of VEOS would remain due to the original goals of the project. This section presents alternative design possibilities in the context of deeply revised objectives.

The interface limitations of VEOS[55] are primarily a result of striving to address many different user-types and programming methodologies simultaneously. Because VR will continue to demand multidisciplinary skills, VEOS should continue to address users with different skills and experience levels. However, trying to incorporate different programming paradigms in the *same* interface may have introduced unnecessary complexity rather than providing added flexibility.

Part of the reasoning for VEOS's hybrid interface was to provide multiple mechanisms for achieving the same tasks, giving user's their choice of methodology. Another part of the basis for hybridization was to experiment with and compare different approaches to the same problems. The next VEOS evolute should incorporate the more successful programming mechanisms, based on performance and reasonability, thus encouraging a narrower range of programming paradigms. This 'natural selection' of features would reduce the decision space of primitives from which to choose in accord with the knowledge gained from VEOS 3.0. This would invariably encourage increased standardization among code idioms and thus a better understanding of common strategies.

Specifically, the object oriented features have proven to be efficient and easy to reason about. Explicit message passing appears to help users understand what and when data moves through their distributed applications. The shared grouplespace offers some useful features, such as data spaces and anonymous subscriptions to data flow. However, the mechanism of transport in the shared grouplespace was neither obvious to users nor reliable enough to be transparent. The abstract data features appear worthwhile, but would be better reformed into a message passing regime, leaving aside the entire shared grouplespace concept. As for the local grouplespace, some applications may take advantage of it's pattern matching aspects[56]. However, it appears better suited as an application specific library that could be incorporated when needed rather than playing a central role in the Kernel.

The next VEOS would still apply the hybridization principle. But rather than hybridization across a single lisp interface as VEOS 3.0, VEOS should offer stratified

---

[55] at the same time appearing too complex for new users and yet too clumsy or limited for sophisticated users. Moreover, the interaction between feature sets is not always apparent.

[56] assuming that an efficient implementation could be developed.

interface. Different users seek different objectives in terms of creativity, precision, and complexity. Many users require only very high level primitives for *configuration* level tasks. Other users require a moderately complex set of primitives for *scripting* tasks yet still require a forgiving prototyping environment. Still other users require very low level primitives for *precision* and performance oriented implementations.

It has been suggested that the prototyping power of VEOS comes not from any simple interface language[57], but rather from the extensive library of reusable modules, many of which have been implemented in a lower level language. The ability to bring these modules together is currently supported only at the Lisp scripting level. As such, the more creative configuration level tasks are quite arduous because no higher level interface is present. The next VEOS could offer a three levels of interface, perhaps a GUI for configuration tasks, perhaps Lisp for scripting tasks, and perhaps C++ for efficiency and precision oriented tasks.

The performance limitations of VEOS[58] are partly a result of striving for platform independence and portability, and partly a result of striving for a simple high-level interface.

In order to utilize the performance capabilities of the computing hardware, that a high level interface, such as outlined above, not be simply a composition of primitives from the next lower level. This stacking of functionality generates the potential for misuse of the lower level primitives. Rather, the high-level primitives should be designed as much as possible in a precision system level language toward the expected use of the primitive. This issue arises for compilers that cannot achieve optimum performance because they compile a high level language into an intermediate language (requiring another compilation) instead of the native language of the hardware.

The goals of platform independence and portability put heavy constraints on performance. Were VEOS to assume a particular platform, the networking performance could be significantly improved through an implementation that was tailored to the specific characteristics of that platform. VEOS networking would also benefit from the

[57] e.g. Lisp and Fern.

[58] e.g. speed and consistency.

distinction between local area communication and wide area communication. In particular, VEOS nodes could use a broadcast scheme between nodes that were known to be proximal, but use a point to point scheme between node clusters in more remote locations. This idea was suggested for scalability in [MOSES].

Confining the domain of VEOS to a single platform has other advantages. The foremost being the possibility of preemption. As stated earlier, VEOS 3.0 offers no process preemption because the code to perform preemptive context switching relies on operating system kernel routines and the native instruction set of the hardware. As such, all current application process must remain discrete, running only in short duty cycles. A new VEOS would still encourage such short discrete tasks since they do lend themselves to effective processor utilization[59]. However, for tasks that demand longer duty cycles, VEOS could multiplex the node process via preemption such as is done with preemptive thread packages. Likewise, a new VEOS would still encourage asynchronous communication semantics to achieve maximum parallelism and hence processor utilization. But with the capability for preemption, VEOS could more completely support synchronous communication semantics for situations where asynchronous communication is awkward.

There are performance advantages to developing VEOS for a single platform. However, the disadvantages are great as well. VEOS could no longer serve to tie together heterogeneous platforms. Hence, VEOS would be less viable as a VR prototyping tool for other sites. One solution to this dilemma involves a larger support staff that could develop a VEOS implementation for multiple platforms. This approach would require that the network interface and programming interface remain consistent across implementations, yet the internals could take advantage of hardware specifics such as multiple processors, shared memory, and graphical user interfaces.

[59] requiring no context switching.

Chapter 8: Summary

VEOS was designed for fast prototyping of distributed virtual environment applications across heterogeneous workstation clusters. The VEOS programming model emphasizes asynchronous communication, and task distribution based on *entities*.

The architects of VEOS attempted to address the following primary goals and constraints in the VEOS design.

- Rapid prototyping.
- User accessibility.
- Heterogeneous workstation LANs.
- Free software.
- Portable.

The resulting VEOS design reflects these primary foundations of previous work.

- Lisp prototyping language.
- Linda process coordination model.
- Rewrite systems.
- Distributed object systems exemplified by Emerald.
- Cyclic executive operational regime.

The VEOS design is a hybrid of these systems, with the concept of entities providing an integrating structure. Entities support modular object-oriented task decomposition as in Emerald, abstract process coordination through a shared dataspace as in Linda, pattern matching inference over a database as in rewrite systems, and a cyclic process model as in the cyclic executive.

VEOS has provided a foundation for conceptual development at HITL by allowing users to focus more on the creative aspects of VE design rather than the recurring low level operations issues. Specifically, VEOS provides abstract services for task decomposition and scheduling, data organization and transport. Additionally, VEOS provides an abstract common ground for task-specific libraries. In sum, VEOS's generic services facilitate rapid prototyping.

For VEOS users, the cost of such abstraction has been performance. Consequently, VEOS users employ coarse-grain distribution strategies to achieve acceptable application behavior. Heeding to this shortcoming, VEOS supports incremental performance upgrades. The generic constructs of VEOS provide a proving ground for concepts. Once implemented, these concepts manifest as suites of entities, or modules. Modules can be further refined using general VEOS constructs or, without loss of generality, replaced with more efficient implementations.

Chapter 9: Conclusion

If anything is to be learned from the VEOS development, it is that the VEOS goals are ambitious since 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, VEOS's emphasis on modularity facilitates the process of rapid prototyping that was sought by the initial design.

The subtext of this discussion is that VE design is inherently a multidisciplinary process and requires the efforts of many people, each with expertise in different areas. Successful VE applications are brought about by system programmers implementing and abstracting bottleneck components, designers creating interesting objects and terrains, theatric minds composing story lines, dynamics experts implementing compelling behaviors, visionaries encouraging the whole process, psychology researchers focusing on perceptual understanding, and systems architects building automated and reliable infrastructures. Perhaps the upcoming generation of VR technology will foster a system like VEOS that allows experts in these complimentary disciplines to combine efforts for even greater potential.

References

[Active] T. von Eicken, D.E. Culler, S. C. Goldstein, K. E. Schauser; "Active Messages: a Mechanism for Integrated Communication and Computation"; University of California at Berkeley; ACM July 1992, pp 256-266.

[ArchVR] P. A. Appino, J. B. Lewis, L. Koved, D. T. Ling, D. Rabenhorst, C. F. Codella; "An Architecture for Virtual Worlds"; IBM Watson Research Center; Presence: Teleoperators and Virtual Environments, MIT Press, Vol. 1, No. 1, Winter 1992.

[Async] G. Coco, D. Lion; "Experiences with Asynchronous Communication Models in VEOS, a Distributed Programming Facility for Uniprocessor LANs"; Human Interface Technology Laboratory of the Washington Technology Center, University of Washington; Technical Report R-93-2, December 1992.

[ATM] C. A. Thekkath, H. Levy, E. Lazowska; "Efficient Support for Loosely Coupled Multicomputing on ATM Networks"; University of Washington Computer Science; December 1992; not yet published.

[Body] M. Minkoff; "The Participant System: Providing the Interface in Virtual Reality"; Master's Thesis, Human Interface Technology Laboratory of the Washington Technology Center, University of Washington; Technical Report R-93-5, June 1993.

[Boundary] W. Bricken; "Boundary Logic (boundary implementations)"; Human Interface Technology Laboratory of the Washington Technology Center, University of Washington; Technical Report P-90-3, March 1990.

[Chores] D. Eager et, J. Zahorjan; "Chores: Enhanced Run-Time Support for Shared-Memory Parallel Computing"; University of Washington Computer Science; course reading for University of Washington CSE 551, 1992.

[Concept] W. Bricken, "Software Architecture for Virtual Reality"; Human Interface Technology Laboratory of the Washington Technology Center, University of Washington; Technical Report P-90-4, 1990.

[Constraint] W. Leler; Constraint Programming Languages, Their Specification and Generation; Addison-Wesley, 1988.

[Coord] D. Gelertner, N. Carriero; "Coordination Languages and their Significance"; Communications of the ACM, Vol. 35, No. 2, February 1992, pp 97-107.

[Design] W. Bricken, "VEOS Design Goals"; Human Interface Technology Laboratory of the Washington Technology Center, University of Washington; Technical Report M-92-1, 1992.

[Emerald] E. Jul, H. Levy, N Hutchinson, A. Black; "Fine-Grained Mobility in the Emerald System"; University of Washington; ACM Transactions on Computer Systems, Vol. 6, No. 1, February 1988, pp 109-133.

[HIDRA] R. Kazman, "HIDRA: An Architecture for Highly Dynamic Physically Based Multi-Agent Simluations"; Software Engineering Institute, Carnegie Mellon University; to appear in the International Journal of Computer Simulation, 1993.

[Ivy] K. Li, P. Hudak; "Memory Coherence in Shared Virtual Memory Systems"; ACM Transactions on Computer Systems, Vol. 7, No. 4, November 1989, pp 321-359.

[Linda] "Kernel Linda Specification: version 4.0", Technical Note, Cogent Research, Inc.; Beaverton, Oregon, 1990.

[Mach] D. Black, D. B. Golub, D. P. Julin, R. F. Rashid, R. P. Draves, R. W. Dean, A. Forin, J. Barrera, H. Tokuda, G. Malan, D. Bohman; "Microkernel Operating System Architecture and Mach"; Proceedings USENIX Workshop on Microkernels and Other Kernel Architectures, April 1992, pp 11-30.

[Math] S. Wolfram, Mathematica: A System for Doing Mathemtics by Computer; Addison-Wesley, 1988.

[Midway] B. Bershad, M. J. Zekauskas, W. A. Swadon; "The Midway Distributed Shared Memory System"; School of Computer Science, Cargnegie Mellon University; course reading for University of Washington CSE 551, 1992.

[Moses] D. J. Pezely, M. D. Almquist, W. Bricken; "Design and Implementation of the Meta Operating System and Entity Shell"; Human Interface Technology Laboratory of the Washington Technology Center, University of Washington; Technical Report R-91-5A, 1991.

[Munin] J. B. Carter, J. K. Bennet, W. Zwaenepoel; "Implementation and Performance of Munin"; Computer Systems Laboratory, Rice University; course reading for University of Washington CSE 551, 1992.

[Nectar] H. T. Kung, R. Sansom, S. Schlick, P. Steenkiste, M. Arnould, F.J. Bitz, F. Christianson, E.C. Cooper, O. Menzilcioglu, D. Ombres, B. Zill; "Network-Based Multicomputers: An Emerging Parallel Architecture"; Computer Science, Carnegie Mellon University; ACM, July 1991, pp 664-673.

[NetLinda] M. Arango, D. Berndt, N. Carriero, D. Gelertner, D. Gilmore; "Adventures with Network Linda"; Supercomputing Review, October 1990, pp 42-46.

[Presto] B. Bershad, E. Lazowska, H. Levy; "PRESTO: A System for Object-oriented Parallel Programming"; University of Washington Computer Science; Software-Practice and Experience, Vol. 18(8), August 1988, pp 713-732.

[Rewrite] N. Dershowitz, J. P. Jouannaud; "Chapter 6: Rewite Systems"; Handbook of Theoretical Computer Science, Elsevier Science Publishers B.V., 1990; pp 245-320.

[RPC] A. Birrel, B. Nelson; "Implementing Remote Procedure Calls"; Xerox Palo Alto Research Center; ACM Transactions on Computer Systems, Vol. 2, No. 1, February 1984, pp 39-59.

[Sched] T. E. Anderson, B. N. Bershad, E. D. Lazowska, H. Levy; "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism"; ACM Transactions on Computer Systems, Vol. 10, No. 1, February 1992; pp 53-79.

[Simple] W. Bricken, "A Simple Space"; Advanced Decision Systems, 1986; reprinted as a Human Interface Technology Laboratory Technical Report R-86-3.

[Smalltalk] A. Goldberg, "Smalltalk-80"; Xerox Corporation; Addison Wesley, 1984.

[Spector] A. Z. Spector, "Performing Remote Operations Efficiently on a Local Computer Network"; Comunications of the ACM, Vol. 25, No. 4, April 1982, pp 246-260.

[State] M. J. Zyda, K. Akeley, N. Badler, W. Bricken, S. Bryson, A. van Dam, J. Thomas, J. Winget, A. Witkin, E. Wong, D. Zeltzer; "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, Commitee on Virtual Reality Research and Development; February 1993.

[Tuple] D. Gelertner, J. Philbin; "Spending Your Free Time"; Byte, May 1990.

[Tuplex] "Tuplex 2.0 Software Specification"; Torque Systems, Inc., Palo Alto, CA; November 1992.

[VEOS] W. Bricken, G. Coco; "The VEOS Project"; to appear in W. Barfield and T. Furness (eds), Advanced Interface Design and Virtual Environments; Oxford University Press, 1993.

[VPL] "Virtual Reality data-flow language and runtime system, Body Electric Manual 3.0"; VPL Research, Redwood City, CA; February 1991.

[XLisp] D. Betz, T. Almy; XLISP 2.1 User's Manual.

Appendix A
Nancy Pattern Language Tutorial

At the unix prompt, a VEOS node is invoked:

```
%  veos
XLISP version 2.1, Copyright (c) 1989, by David Betz
```

Initialize the VEOS Kernel, creating an empty grouplespace.

```
> (vinit)
>
```

To see the entire contents of the grouplespace, use a nondestructive query. The most succinct way to do this is to match the grouplespace itself and copy all of its elements.

```
> (vcopy '(> @@))
NIL
```

Indeed the grouplespace is empty (in Lisp, () is equivalent to NIL).

```
>
```

To use vput, match the grouplespace and point to the void within it. Thus, the (^) pattern. This is literally where the given data is put.

```
> (vput "first" '(^))
T
```

Here, we reqeusted a simple insert operation, so vput returns T or NIL depending on the success of the match. During a replace vput operation, vput returns what the action replaced in the grouplespace. Further along, this will come into play.

```
>
```

To see the entire contents of the grouplespace, do a vcopy as above:

```
> (vcopy '(> @@))
("first")
```

Note that the @@ can match a single element if there is only one, or it can match many elements if there are more than one. In the latter case, the results are conveniently returned in a list. In the interest of consistency, all Nancy match results are returned in a list.

```
>
```

Now insert a list after the first element of the grouplespace:

```
> (vput '("third") '(@ ^))
T
```

Again, the insert was successful. The entire contents:

```
> (vcopy '(> @@))
("first" ("third"))
>
```

Now insert an element between the first element and the rest of the elements in the grouplespace.

```
> (vput "second" '(@ ^ @@))
T
```

Note that (@ ^ @) would have also worked and would have been more precise. And inspecting the entire contents:

```
> (vcopy '(> @@))
("first" "second" ("third"))
>
```

Next, inserting a vector into the grouplespace. This demonstrates two other features. The @2, in the given position in the pattern matches the first two elements of the grouplespace. the "third" in the pattern matches the actual data in the grouplespace. This, of course, is unlike how an @ matches any element where no data is compared.

```
> (vput '#(1.4 3.9 9.0) '(@2 ("third" ^)))
T
```

The data now resides where the ^ was in the previous pattern.

```
> (vcopy '(> @@))
("first" "second" ("third" #(1.4 3.9 9)))
>
```

Sometimes it is useful to replace existing data in the grouplespace. This can be done by removing, then inserting correct data. Or it can be done with a replacing vput. Here, replace the first element of the grouplespace with the given data. Also, match the remaining elements in the grouplespace to achieve a successful match.

```
> (vput "uno" '(> @ @@))
("first")
```

Vput returns the removed data. And the first element has been replaced.

```
> (vcopy '(> @@))
("uno" "second" ("third" #(1.4 3.9 9)))
>
```

Now, the more subtle features of pattern matching. Begin by emptying the current contents of the grouplespace to begin a new session.

```
> (vget '(> @@))
("uno" "second" ("third" #(1.4 3.9 9)))
```

Confirm that the grouplespace is empty:

```
> (vcopy '(> @@))
NIL
>
```

Here, insert some new data:

```
> (vput '("animal" "giraffe") '(^))
> (vput '("plant" "fern") '(^ @))
T
> (vput '("animal" "lion") '(^ @2))
T
> (vcopy '(> @@))
(("animal" "lion") ("plant" "fern") ("animal" "giraffe"))
>
```

*Now some useful matches can be performed on the grouplespace. Suppose the user wants to find the tag associated with the "fern" data. Simply ask for the element immediately preceding the "fern" element. note the use of the ** wildcard. The ** pattern element has two functions in this pattern. First, like @@ it matches all the remaining elements in the grouple. Second, it explicitly makes the containing grouple an order-independent pattern. In other words, when Nancy sees a ** in a pattern grouple, it ignores the order of the source (grouplespace) elements when matching; it matches purely by content.*

```
> (vcopy '((> @ "fern") **))
("plant")
```

Note that although the marked element is "plant", but the result is contained within an extra grouple. As explained above, all Nancy results are contained within a grouple.

```
>
```

To find the data associated with the tag "animal", do a similar match.

```
> (vcopy '(("animal" > @) **))
```

```
("lion")
```

But this is only a partial answer. Because there is more than one instance of the tag "animal" at that level of the grouplespace. The user may want all the possible matches of this form.

```
> (vcopy '(("animal" > @) **) :freq "all")
("lion" "giraffe")
>
```

This feature can also be used with vput to do an exaustive replace. Here, XLisp allows the arguments to a function to appear on consecutive lines.

```
> (vput "mammal" '((> "animal" @) **)
                 :freq "all")
("animal" "animal")
```

Vput returns exactly what it replaced. Check that the replace was successful.

```
> (vcopy '(> @@))
(("mammal" "lion") ("plant" "fern") ("mammal" "giraffe"))
>
```

Next, the pattern matching features that correspond to the dynamic issues of maintaining the grouplespace. Specifically, matching that is sensitive to the relative ages of the data can be used for so-called 'delta matching'. The existing grouplespace contents will illustrate.

First, make a Nancy time stamp.

```
> (setq ts (vmintime))
1000
>
```

Use this time stamp with vcopy. When passing a time stamp to vcopy, it performs the given match but only returns matched data that is younger than the time stamp. Vmintime returns a guaranteed oldest time-stamp. This means that using it will guarantee seeing everything in the grouplespace.

```
> (vcopy '(> @@) :test-time ts)
(("mammal" "lion") ("plant" "fern") ("mammal" "giraffe"))
```

Vcopy compared all the matched data against the time stamp, all of it had been modified (with vput) after the time given by the time stamp. Vcopy modifies the argument time stamp in place to reflect that it matched with that pattern. The time stamp has now been modified.

```
> ts
1001
```

```
>
```

Making the same match again with the same time stamp returns no data. This corresponds to there being no change (or delta) in the data since the last match.

```
> (vcopy '(> @@) :test-time ts)
NIL
>
```

Insert some new data to see how the delta will be shown.

```
> (vput '("mammal" "fox") '(> (@ "lion") **))
(("mammal" "lion"))
>
```

Matching the entire grouplespace with the time stamp, only the new data is returned, since it was added since the last match.

```
>  (vcopy '(> @@) :test-time ts)
(("mammal" "fox"))
```

Again the time stamp has been modified in place to reflect a new matching. Inspect the time stamp and find that is has been changed. The numeric value of a timestamp is unimportant, only that the time values increase.

```
> ts
1002
>
```

Notice that above both the data and the tag ("mammal" "fox") were replaced in the grouplespace. To replace only the data:

```
> (vput "carrot" '((@ > "fern") **))
("fern")
> (vcopy '(> @@) :test-time ts)
(("carrot"))
```

As expected, vcopy returns only the changed data. Above, the changed data is ambiguous without its tag. But the tag was not modified and so is not returned. Use the 'touch' feature during a vput to mark elements in the grouplespace as having also been modified.

```
> (vput "palm" '((~ @ > "carrot") **))
("carrot")
```

Note the ~ pattern modifier in the above pattern. If the pattern successfully matches, Nancy'touches' the element following the ~ in the pattern as having been modified.

```
> (vcopy '(> @@) :test-time ts)
(("plant" "palm"))
```

*And this is the desired effect. Unlike the > or ^ symbols, any number of ~ can be used in a vput pattern. It is important to remember that a single time stamp represents the most recent match of a *particular* pattern from a *particular* point of view. For example, if writing program for a server to which many clients make matching requests, the server should maintain a time stamp for each client and each pattern type.*

Appendix B
The Fern Programmer's Interface

```
;;-------------------------------------------------------------
;; file: fuser.doc
;; The Application Programmer's Interface to the Fern System.
;;
;; FERN is the Fractal Entity Relativity Node.
;;
;; creation: August 10, 1992
;;
;; by Geoffrey P. Coco and Colin Bricken
;; Software Engineering by Geoff Coco
;; HITLab, Seattle
;;-------------------------------------------------------------

;;-------------------------------------------------------------
;; Copyright (c) 1992, Washington Technology Center
;;
;; This program's use is restricted under the terms of the
;; WTC LICENSE
;; which can be found the in root of the veos directory tree.
;;-------------------------------------------------------------


-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------

     Specification of The FERN Distributed Environment

-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------

-----------------------------------------------------------
- Create the fern distributed computing environment -

(fern-init <:host-pool host-list> <:display-host host-name>)
     <host-list>    - list of host-args, hosts in run pool.
     <host-name>    - STRING, host to display node windows
     returns:       - T/NIL
```

Once running, fern provides a homogenous, parallel computation environment. The host-pool defines the nodes which make up fern's distributed multiprocessor. In other words, the host-pool is the maximal set of hosts that your distributed fern program will require for this invokation.

Each item in the host-list must be either:
1) a string that names the chosen host, or
2) a list containaing the string from (1) and a
   string naming the binary executable to run as the node.

For example,

```
(fern-init :host-pool '("slithy"
               ("iris2" "/home/cygnus/bin/imager")
               ("water" "/home/cocteau/bin/swarmer")
               "hal")
```

On hosts slithy and hal, the default veos binary executable will be used. On hosts iris2 and water, the named custom binaries will be used. In any case, the startup lisp file is /home/veos/ote/veos2.2/src/tabula_rasa.lsp

A fern program should call (fern-init ...) only once. The node that begins a program by calling (fern-init ...) is called the 'console' node. fern-init automatically

```
launches and initializes nodes on all the remaining hosts
in the host-pool (the 'console' is always in the pool).

If the :host-pool argument is not specified, the default
host-pool contains only the console node.

If the :display-host argument is not specified, all node
xterm windows automatically display on the console's host.

A running fern pool can stay running through many runs of
the same program.  As long as the nodes haven't crashed
(bus error, etc..), programs can run again and again
without making new calls to (fern-init ... ).
-----------------------------------------------------------

-----------------------------------------------------------
- Takedown the entire distributed computing environment -

(fern-close)
     returns:       - does not return

Takes down each node in the fern pool including itself.
Other nodes must still be in (fern-go) to respond properly.
-----------------------------------------------------------

-----------------------------------------------------------
- Join with separately running node-pools.

(fern-merge-pool <terminal-node>)

     <terminal-node>     - any node (uid) in remote pool
     returns        - T/NIL

Used when fern application is composed of separate modular
running node-pools.  This is useful for devoting pools to
```

significant application tasks (e.g. participant suite, entire self-sufficient worlds, etc..).

The pool that results from starting a fern-program with (fern-init ...) is the native pool for those entities within that program.

Upon merge, join with the native node pool associated with the given node. Once a merge is completed, all nodes in the remote pool become reachable from the native pool also. All nodes in the new aggregate pool are on equal terms for entity communication. Note, for accountability purposes, Fern programs can only create or destroy entities in the original pool created with (fern-init ...).

- Detatch native pool from the aggregate pool.

```
(fern-detatch-pool <terminal-node>)

     <terminal-node>     - any node (uid) in remote pool
     returns        - T/NIL
```

Detatch from the native pool named by the remote node. There is exactly one native pool for every entity - the pool which was created directly with (fern-init ...). Relinquish access to all entities within the remote pool.

- Find out if the pool is still alive

```
(fern-ping-pool <terminal-node>)
```

```
     <terminal-node>     - any node (uid) in remote pool
     returns        - T/NIL
```

Send tickle packet liveness. The timeout is the same for (fern-seq-send ...).

```
-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------
```

Invokation of FERN Entities

```
-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------
```

- Run a fern program -

```
(fern-run  <spore-ent>  <:file-host  filehost>  <:run-host
runhost>)

     <spore-ent>    - entity definition for initial entity
     <filehost>     - host to find init ent description
     <runhost>      - name of host on which to run init ent
     returns        - does not return
```

Every fern program begins with an initial 'spore' entity. The only required argument to (fern-run ...) is the entity definition of your program's spore entity. This entity can make further entities, install processes for itself, etc. This initial entity is the main() of the fern distributed

```
program.

The arguments pertaining to the spore entity (init-expr,
file-host, run-host) have the same semantics as in
(fern-new-ent ... ).

Once you've called (fern-run ... ), EVERYTHING is done
within an entity context...

1. Fern performs pattern matches from the perspective of
   the calling entity.  Each entity sees a customized version
   of the grouplespace.

2. All process in fern is associated with some entity in
   the form of 'methods', 'persist procs', or 'react procs'.

   I. Methods are user-defined entry-points for inter-entity
   communication.  Entities can call their own or each
   other's methods with (fern-send ... ).  A Method call is
   accountable to the calling entity.

   II. Persist Procs are user-defined processes associated
   with fern entities.  An entity can install multiple
   persist procs which execute in that entity's context.
   See fern-persist for more details.

   III. React Procs are user-defined functions bound to
   data-update events.  An entity can install multiple react
   procs which execute in that entity's context when new data
   arrives from other entities.  See fern-perceive for more
   details.
-----------------------------------------------------------

-----------------------------------------------------------
- Create and invoke a new fern entity -
```

```
(fern-new-ent <init-expr> <:run-host runhost>
                         <:file-host filehost> )
     <init-expr>    - list of lisp expressions or filename
     <filehost>     - host to find init ent description
     <runhost>      - name of host on which to run init ent
     returns:       - new entid
```

The init-expression is a list of user-defined lisp expressions that will prepare the entity for normal execution. See (fern-entity ...) below for more.

Alternatively, the init-expr can be the filename of an initial entity definition. These files follow the same format - expressions which when evaluated determine an entity's character. Fern entity definition files must end in ".fent", and (fern-new-ent ...) automatically appends the ".fent" suffix onto filenames. Always use full pathnames for best portability.

To get emacs to edit ".fent" files as lisp, put this line in your .emacs file:

```
(setq auto-mode-alist
      (cons '("\\.fent" . lisp-mode) auto-mode-alist))
```

The optional :file-host argument can be used to instruct fern where to look for the named entity definition file. If the file-host is not specified, fern looks for the file on the console node.

The :run-host argument is also optional. It tells fern where to run the new entity. If run-host is not specified, it currently runs on a random host in the pool.

When the entity has been created and the initial expression has been evaluated, the new entid is returned to the caller.

```
-----------------------------------------------------------

-----------------------------------------------------------
- Takedown a fern entity and free all its resources -

(fern-dispose-ent <entid> )
     <entid>        - optional entid of entity to dispose
     returns:       - T/NIL
```

The entid argument is optional. If an entid is not given, the entity which made call has requested self-disposal.

```
-----------------------------------------------------------

-----------------------------------------------------------
- Generate an initial expression for a fern entity -

(fern-entity <expr> <expr> <expr> ... )
     <expr>         - unevaluated lisp expressions
     returns:       - an unevaluated list of the exprs
```

The argument expressions are quoted lisp code which a new Fern entity will evaluate upon startup. (fern-entity ...) returns an expression suitable to pass (fern-new-ent ...) for creating an entity with Fern.

These normally include:
1. Creating self attributes with (fern-put.bndry.attr ...)
2. Staking out private workspace with (fern-put.locl ...)
3. Creating methods of behavior with (fern-def-meth ...)
4. Creating entity processes with (fern-persist ...)
5. Declaring active database needs with (fern-perceive ...)
6. Initializing any peripheral connections, e.g. (sensor-init ...)
7. Defining functions behind the methods, e.g. (defun ...)

```
-----------------------------------------------------------


-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------

         Object-Oriented Aspects of Entities

-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------



-----------------------------------------------------------
- Define fern entity behavior -

(fern-def-meth <name> <lambda-expr> )
     <name>         - name of method, STRING argument
     <lambda-expr>  - lambda which defines method behavior
     returns:       - T/NIL
```

A method is a well-defined entry point to a fern entity. Entity can communicate and pass data directly using method calls.

(fern-def-meth ...) adds the given method to the calling entity. Other entities can use (fern-send ...) to have the entity perform a method on behalf of the calling entity.

```
Ex:  let's say ent1 defines a method:
     (fern-def-meth "print-plus" (lambda (x)
                                   (print (+ x 1))))
     then, ent2 can use the method:
```

```
     (fern-send ent1 "print-plus" 2)
     the screen when ent1 is running should print '3'.
-----------------------------------------------------------

-----------------------------------------------------------
- Undefine fern entity behavior -

(fern-undef-meth <name> )
     <name>         - name of method, STRING argument
-----------------------------------------------------------

-----------------------------------------------------------
- Asynchronous method call -

(fern-send <entid> <method-name> <arg1> <arg2> ... )
     <entid>   - id of destination entity
     <method-name>       - STRING name of method
     <args> ...     - the arguments to the method

Entities may call their own methods by passing 'self' as the
entid.
-----------------------------------------------------------

-----------------------------------------------------------
- Synchronous method call -

(fern-seq-send <entid> <method-name> <arg1> <arg2> ... )
     <entid>   - id of destination entity
     <method-name>       - STRING name of method
     <args> ...     - the arguments to the method


The calling semantics are the same as (fern-send ... )
except that (fern-seq-send ... ) dispatches the destination
entity's named method immediately.  (fern-seq-send ... )
```

returns only when the method has completed and has returned a result. (fern-seq-send ...) returns the result of the method call.

```
-----------------------------------------------------------

-----------------------------------------------------------
- Asynchronous method call -
- using streams for flow control -

(fern-str-send <entid> <method-name> <arg1> <arg2> ... )
     <entid>   - id of destination entity
     <method-name>       - STRING name of method
     <args> ...     - the arguments to the method
     returns:       - T/NIL
```

The calling semantics are the same as (fern-send ...) except that (fern-str-send ...) only performs the send if all outstanding sends to this destination have been serviced.

Fern uses a message pacing algorithm called streams. A stream is an logical connection from one entid to another that has a maximum carrying capacity. In other words, streams ensure that sender entities only send messages as fast as their receiver entities can digest them.

(fern-str-send ...) sends the given method call only if the stream to that entity is clear. A return value of true means that the method call was sent. A return value of NIL may mean that the stream was full and the caller should try again later. A return value of NIL could suggets others errors as well.

Fern users can save much overhead and ambiguity in using this pacing mechanism by testing the stream _before_

```
calling (ferr-str-send ...).  To test a stream for
clearness, use (fern-str-clrp entid) as described below.
-----------------------------------------------------------

-----------------------------------------------------------
- Test a stream -

(fern-str-clrp <entid> )
     <entid>        - stream destination
     returns:       - T/NIL

This function quickly checks whether the logical channel
between the calling entity and the destination entity is
sufficiently clear for passing messages.
-----------------------------------------------------------

-----------------------------------------------------------
- Asynchronous informal entity process call -

(fern-as <entid> <expr> )
     <entid>   - id of destination entity
     <expr>    - quoted evaluable expression
     returns:       - T/NIL

Same semantics as (fern-send ... ), but without the method
formality.  The given expression is evaluated in the
context of the named entity.
-----------------------------------------------------------

-----------------------------------------------------------
- Synchonous informal entity process call -

(fern-seq-as <entid> <expr> )
     <entid>   - id of destination entity
     <expr>    - quoted evaluable expression
     returns:       - T/NIL

Same semantics as (fern-seq-send ... ) revised as above.
-----------------------------------------------------------

-----------------------------------------------------------
- Asynchronous informal entity process call -
- using streams for flow control -

(fern-str-as <entid> <expr> )
     <entid>    - id of destination entity
     <expr>     - quoted evaluable expression
     returns:       - T/NIL

Same semantics as (fern-str-send ... ) revised as above.
-----------------------------------------------------------


-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------

           Specification of Entity Process

-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------


-----------------------------------------------------------
- Lightweight task creation -

(fern-persist <expr> <:name procname> )
     <expr>         - repeatable lisp expression.
     <procname>     - optional STRING name of proc
     returns:       - new proc name
```

Installs a new persist proc in the calling entity's proc table. This is similar to forking a process.

The persist concept implements a form of cooperative multitasking. To ensure proper multitasking and to approach the effect of parallelism, your persist expressions should:

1. Be fast-evaluating.

If the proc represents an ongoing process, try to break the task into discrete tasks which each take a small slice of time. Use quick state checks to exit the proc early in case there is no work to do. Watch out for loops where the number of iterations can become large at times.

2. Be atomic.

The job of one proc should be conceptually neat. This is so that when procs are evaluated in different orders, their relative behaviors remain consistent. If there's a job that two procs collectively perform, they should be one proc.

3. Never block.

A proc that blocks will starve other procs of valuable time slices. Remember, since frames rates in veos aren't enforced, please be a 'good citizen'.

There are indeed situations where it makes no sense to continue without a certain resource (like data from a remote query, or data from a hardware driver). But instead of freezing up the persist cycle while waiting for your resource, use an asynchronous method.

For example, when polling hardware, use a non-blocking scheme - if there's no data, return and check again next cycle. Also, when waiting for a remote query reply, send a request and check for the reply each cycle until it arrives.

- Lightweight task disposal -

```
(fern-desist <procname> )
     <procname>     - optional name of persist proc to kill
     returns        - the deleted proc-name
```

Terminate the given persist proc. If no proc is specified, terminate the calling proc.

If a procname is not specified, fern removes the currently running persist proc from the system proc list. That running proc is allowed to complete it's last evaluation normally.

(fern-desist ...) returns the name of the proc that was deleted from the run queue.

||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

Configuration of Trademark FERN Virtual Grouplespace

```
-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------
```

```
-----------------------------------------------------------
- Enter a fern space -

(fern-enter <space-entid> )
     <space-entid>       - the unsuspecting entity
     returns:       - T/NIL
```

Request to becoming a 'subling' of the named entity. You would do this to instigate an implicit and automatic awareness of other entities that may also be 'entered' in the same space. An entity entered in another entity (a space by virtue of this relationship), will 'see' other entities also in that space.

Entities that are entered in the same space are 'siblings'. Siblings 'see' each other through fern's automatic database propogation. When one sibling changes its local database (its boundary), these changes are automatically propogated to others siblings' databases (their externals).

Entities in spaces can express particular interests or filters, to limit the automatic database propogation to the necessary data streams. These data interests are called 'in-senses'. In-senses are hints to fern about what kind of data a subling wishes to receive (see fern-perceive)

When any entity enters a space, the entity will appear in its own external as a sibling. This virtual representation is filtered however through the entity's in-senses (see fern-perceive).

A space entity need not have any special behavior to function properly as a space. All fern entities are equally equipped to serve as spaces. Space entities perform very little computational legwork. Instead, they provide a conceptual meeting ground for entities having particular relationships.

- Exit a fern space -

```
(fern-exit <space-entid> )
     <space-entid>       - the unsuspecting entity
     returns:       - T/NIL
```

Relinquish all services of the named space entity. The calling entity will no longer 'see' siblings that were entered in this space. Those siblings will no longer 'see' the calling entity in their external.

- Subscribe to data flows -

```
(fern-perceive <attr-name>
               <:react (lambda (entid attr-val) ...)> )
     <attr-name>    - attribute of interest
     <lambda>       - optional react proc
     returns:       - T/NIL
```

```
Declare an in-sense.  By calling this function once, the
calling entity declares that whenever it is entered in a
space, it would like to 'see' entities in that space which
have the named attribute.  You can perceive many attributes
with multiple calls to (fern-perceive ... ).

Likewise, if another entity sharing a space with your
entity has declared an in-sense as above, then it will
'see' data in your entity's boundary.

Exactly how the caller 'sees' other entities is determined
by the optional :react argument.  In the regular case where
no :react argument is given, the calling entity's
'siblings' partition will be automatically updated by
fern when sibling entities make changes to their boundaries.

If a :react lambda argument is given, fern calls this
lambda function when sibling changes occur.  The lambda
function must take two arguments: the entid of the sibling
that's changing its boundary and the new attribute.

In this mode, fern does not automatically update the
siblings partition.  To achieve the normal fern
auto-grouplspace behavior and _also_ use this :react
feature, your react proc should call (fern-put.sib.attr
entid attr) before it returns.
-----------------------------------------------------------



-----------------------------------------------------------
- Unsubscribe to data flows -

(fern-unperceive <attr-name> )
     <attr-name>    - attribute of interest
     returns:       - T/NIL
```

```
Eradicate an in-sense.  The calling entity's external will
be rid of all references to this attribute.
-----------------------------------------------------------

-----------------------------------------------------------
- Produce data flows (explicitly) -

(fern-exude <attr> )
     attr      - ("attr-name" attr-val)
     returns:       - T/NIL

Create inter-entity data flows manually.  The same behavior
automatically occurs when an entity calls fern-put.attr.

Normally, after an entity calls fern-put.attr, fern passes
these boundary changes to the entity's siblings at some
later time.  With fern-exude, the changes bypass the
grouplespace and flow directly to perceiving siblings.

NOTE: entities that use this function must call
fern-put.attr to initialize the sibling-to-sibling
connections.  For example, during entity initialization,
call fern-put.attr once for each attribute you plan to
exude.
-----------------------------------------------------------


-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------

       Accessing The FERN Perception Partition

-----------------------------------------------------------
```

```
||||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------


-----------------------------------------------------------
- Put, Get, Copy to the entity groueplspace -

Here are the most useful fern grouplespace pattern matches.
To see what the grouplespace looks like in FernII, load up
an entity or two with the works (boundary-attrs, in-senses,
procs and methods) and then call (dump).
-----------------------------------------------------------


-----------------------------------------------------------
;; E X T E R N A L
-----------------------------------------------------------
- Source -

(fern-copy.src )
     returns:      - source entid

     if you are an entity, the entity that created you is
     your 'source'.  All entities in a program have a source
     except the spore.
-----------------------------------------------------------
- Space -

(fern-copy.sps <:test-time test-time> )
     test-time     - optional nancy timestamp
     returns:      - list of current space entids
-----------------------------------------------------------
- Siblings -

(fern-copy.sibs <:test-time test-time> )
     test-time     - optional nancy timestamp
     returns:       - list of perceived entities

     Each sib is of the form: (entid (attr-list)).
     (fern-copy.sibs) returns a list of these structures, one
     for each sibling which shares a fern space with the
     calling entity.  These attr-lists are comprized of only
     attributes that the calling entity has perceived via
     (fern-perceive ... ).

     This function is the bread & butter of reactive
     programming.  It is common practice during a persist
     proc to perform a (fern-copy.sibs ... ) with a
     timestamp, and parse the resulting sparse structure of
     sibling changes to compute new reactive behavior.


(fern-copy.sibs.entids <:test-time test-time> )
     test-time      - optional nancy timestamp
     returns:       - list of entids

     Returns the entids of all the calling entity's siblings.

(fern-copy.sib <entid> <:test-time test-time> )
     test-time      - optional nancy timestamp
     returns:       - the named entity's attr-list

(fern-copy.sib.attr <entid> <attr-name> )
     entid          - entid of sibling
     attr-name      - name of sibling's attribute
     returns:       - given attr value

(fern-copy.sib.attr-names <entid> <:test-time test-time> )
     entid          - entid of sibling
     test-time      - optional nancy timestamp
     returns:       - list of named entity's attribute names
```

```
(fern-copy.sib.attr-if-attr-name <must-name> <gimme-name> )
     must-name      - name of attribute which must appear
     gimme-name     - name of attribute to retrieve
     returns:       - value of gimme-name

     If there is a sibling to the calling entity which has
     the must-name attribute and the gimme-name attribute,
     return the value for the gimme-name attribute.

(fern-copy.sib.attr-if-attr <must-attr> <gimme-name> )
     must-attr      - attribute which must match
     gimme-name     - name of attribute to retrieve
     returns:       - value of gimme-name

     If there is a sibling to the calling entity which
     matches the must-attr (name and value) and the
     has gimme-name attribute, return the value for the
     gimme-name attribute.

(fern-copy.sib.entid-if-attr <must-attr> )
     must-attr      - attribute which must match
     returns:       - entid of matching entity

     If there is a sibling to the calling entity which
     matches the must-attr (name and value) ,return its
     entid.
-----------------------------------------------------------


-----------------------------------------------------------
;; B O U N D A R Y
-----------------------------------------------------------
- Entire boundary -
```

```
(fern-copy.bndry <:test-time test-time> )
     test-time      - optional nancy timestamp
     returns:       - list of calling entity's attributes
-----------------------------------------------------------
- Boundary attributes -

(fern-put.attr <attr> )
     <attr>         - ("attr-name" attr-val)
     returns:       - old attr, if any; or T/NIL

        This function inserts and replaces.  No need to
     get-then-put.

(fern-copy.attr <attr-name> )
     <attr-name>    - name of desired attribute
     returns:       - attribute value

(fern-get.attr <attr-name> )
     <attr-name>    - name of desired attribute
     returns:       - old attr val, if any

     Instead of deleting the attribute from the boundary
     completely, this function actually replaces the current
     value of the attribute with "%", the
     'skull-and-crossbones' symbol.

     The reason for keeping 'dead' data in this way is fairly
     obscure.  The "%" is like an ordinary attribute value,
     so fern will propogate it to siblings as such.  It has
     become customary to reserve the "%" attribute value to
     signify data that was but is no more.

(fern-copy.attr.names )
     returns:       - list of the entity's attr-names
-----------------------------------------------------------
```

```
-----------------------------------------------------------
;; I N T E R N A L
-----------------------------------------------------------
- Subs -

Sublings are the entities for whom your entity is acting as
a space.

(fern-copy.sub.entids <:test-time test-time> )
     test-time      - optional nancy timestamp
     returns:       - list of subling entids

-----------------------------------------------------------
- Entire Local -

The local partition is an entity's private grouplspace
area.  It is only accessed by the user's code.  The local
is the user defined grouplespace partition.

This is your chance to write your own nancy grouplespace
patterns.  Do some experimenting with these functions to
make sure you understand how to use the local partition.

(fern-put.locl <data> <pat> <:freq frequency> )
     data      - any veos compatible expr
     pat       - any single element pattern
     frequency      - "one"/"all" default is "one"
     returns:       - old data; or T/NIL

(fern-copy.locl <pat> <:test-time test-time>
                     <:freq frequency> )
     pat       - any single element pattern
     test-time      - optional nancy timestamp
     frequency      - "one"/"all" default is "one"
     returns:       - matched data

(fern-get.locl <pat> <:freq frequency> )
     pat       - any single element pattern
     frequency      - "one"/"all" default is "one"
     returns:       - removed data
-----------------------------------------------------------
```

- Local Attributes -

Although the local partition is user-defined, user's may use the standard 'attribute' regime for organizing their entities' local partitions.

```
(fern-put.locl.attr <attr> )
     attr      - ("attr-name" attr-val)
     returns:       - old attr, if any; or T/NIL

(fern-copy.locl.attr <attr-name> )
     attr-name      - name of desired attribute
     returns:       - attribute value, if any

(fern-get.locl.attr <attr-name> )
     attr-name      - name of desired attribute
     returns:       - old attr, if any
```

To understand the purpose of 'local attributes', think of each entity as a program which may need its own global variables. Since many entities may inhabit the same lisp environment (i.e. node), lisp global variables are not suitable for this purpose.

Imagine what happens when two entities in the same lisp environment use a global named cur_pos. Each entity will use the cur_pos variable in its own way, thus altering each

other's state unpredictably. Using local attributes
ensures that entities have exclusive use of their memories.

|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

  Additional Features of the FERN Distributed Envionment.

|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||

- Read a file, from anywhere in the fern pool -

```
(fern-read-file <file-name> <:host hostname> )
     <file-name>    - name of file to load
     <hostname>     - name of host from which to load file
     returns:       - unevaluated list of all exprs in file
```

Load the given file from somewhere in the pool. If no
hostname is given, fern attempts to load the file from the
console node.

- Read and evaluate a file, from anywhere in the fern pool -

```
(fern-eval-file <file-name> <:host hostname> )
     <file-name>    - name of file to load
     <hostname>     - name of host from which to load file
     returns:       - T/NIL

Retrieve the named file and evaluate each expression
contained within.  Uses (fern-read-file ... ) to offer
extension of the standard lisp (load ... ) function.

NOTE: Always use full pathnames so that the calling code
will behave the same on different hosts.
-----------------------------------------------------------



-----------------------------------------------------------
- Write a file, to anywhere in the fern pool -

(fern-write-file <file-name> <expr-list> <:host hostname> )
     <file-name>    - name of file to write
     <expr-list>    - list of data
     <hostname>     - name of host on which to write file
     returns:       - T/NIL

Write the given expressions to the named file.  This file
can later be accessed though (fern-read-file ... ).
-----------------------------------------------------------



-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------

          Attention to Node Operation

-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------
```

```
- Fern Node Global Variables:

-----------------------------------------------------------
home           <read-only>

The host name where symbol is evaluated.
-----------------------------------------------------------
self      <read-only>

The entity-id of the evaluating entity.
-----------------------------------------------------------
fern-display   <read-only>

The hostname where all node xterms display.
-----------------------------------------------------------
fern-debug     <set-at-will>

Enables verbose general fern operations.
-----------------------------------------------------------
as-debug       <set-as-will>

Enables tracing of entity context switching.
-----------------------------------------------------------
frame-debug    <set-at-will>

Enables display of frame rate statistics.
-----------------------------------------------------------
flow-debug     <set-at-will>

Enables tracing of automatic data flow.
-----------------------------------------------------------
enter-debug    <set-at-will>

Enables tracing of space entering.
```

```
-----------------------------------------------------------
str-debug     <set-at-will>

Enables tracing of stream message operations.
-----------------------------------------------------------
merge-debug   <set-at-will>

Enables tracing of pool merge operations.
-----------------------------------------------------------
file-debug    <set-at-will>

Enables tracing of file operations.

-----------------------------------------------------------
fmaxclog      <set-with-caution>

The maximum msg width of inter-host streams from this node.
Use these heuristics to tune for specific application:

high values (4-50) yield
    best thruput (best processor utilization and parallism)
    worst latency (lots of msgs are getting queued)
    very clumpy flow (msgs tend to caravan through system)

low values (2-3) yield
    better thruput (some parallism, good utilization)
    not-so-good latency (some msgs are getting queued)
    clumpy flow (msgs tend to caravan through system)

minimum value (1) yields
    worst thruput (little parallism, lots of idle waiting)
    best latency (msgs get serviced right away)
    most consistent transmission (msgs flow at even pace)

the default setting is: 1
```

```
-----------------------------------------------------------


-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------

               Major Confusions

-----------------------------------------------------------
|||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
-----------------------------------------------------------
```

1. Idea of entities and methods.

Entities are smart little optimizers. They do their jobs, you do yours. Share the responsibility, and the abstraction. Let the entity model encourage a high degree of modularity, mobility and correctness in your programs.

Everything happens from within _some_ entity's context, except the first call to (fern-run ...). I mean everything!!

When matches don't work, take the perspective of the entity. What does _it_ see? A quick (dump) will reveal each entity's grouplespace perspective. It helps to become familiar with object-oriented thinking.

```
-----------------------------------------------------------
```

2. The entity concept has new meaning.

Users of FernI will find that the old notion of an entity has been rightly reshaped in FernII into the concepts of the entity and node.

The FernI entity was a heavyweight unix process usually programmed for a discrete task. A FernI program could run several entities on a host machine calling on unix to simulate parallelism by context switching between them. Entities could pass messages to each other via heavyweight unix network sockets.

In FernII, each participating host machine runs exactly one unix process called a FERN node. The need for costly unix context switching is diminished because each node works within one unix process.

The FernII entity is still programmed for a discrete task; but the entity has become lightweight. Many entities run on a single node sharing the unix process. FernII's nodes simulates paralellism by performing lightweight context switching between entities.

----

3. The entity definition.

An entity definition is an unevaluated list of lisp expressions which when evaluated by fern-new-ent will define a fern entity. The lisp expressions will normally consist of calls to: (defun ..) (fern-put.attr ..) (fern-perceive ..) (fern-persist ..) (fern-def-meth) (fern-put.locl ..) and any other setup your entity needs to do.

----

4. Fern spaces are an entity grouping facility.

The FERN virtual grouplspace features are designed to define a crude world 'perspective' for an entity. Furthermore, spaces provide a mechanism for grouping entities into relational sets. Consider the following canonical example:

EntityA is earmarked as the 'gravity' space. Currently entered in EntityA are Entity1, Entity2, Entity3, and EntityGrav. EntityGrav is entered in this space only (by design), and so can easily perceive all entities in the gravity space. The EntityGrav entity is thus equipped to affect its sibling entities with gravity.

Note that in this example, Entity1, Entity2 and Entity3 would need to exude the proper attributes (like "mass" and "6D") to be acted upon by EntityGrav. However, they would not need to perceive anything special to enjoy gravity space influence - they are acted upon by virtue of their membership in that space.
