Prolix
A Text-based Participant System for VR

by

Mark Takacs

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

Master of Science in Engineering

University of Washington

1993

Approved by ______________________________
(Chairperson of Supervisory Committee)

College of Engineering (Inter-Engineering)

August 17, 1993

University of Washington

Abstract

Prolix
A Text-based Participant System for VR

by Mark Takacs

Chairperson of Supervisory Committee: Professor Thomas A. Furness III
Engineering (Inter-Engineering)

Virtual Reality has captured the imagination of many people, yet few have experienced it directly due to the currently expensive display and sensor technologies. Those who have experienced virtual worlds often comment on the world's inability to maintain interest with prolonged use. The only thing that remains consistently interesting are other human users. Yet the number of users in a multi-participant world is realistically limited by the great cost of equipping each user with motion tracking sensors, head mounted audio and visual displays, input devices or other interface gear.

This thesis presents a program called Prolix which allows access to the virtual worlds running at the Human Interface Technology Lab (HITLab) without the expensive interface equipment. Prolix assumes only a keyboard and a screen with vt100 capability. Input to the virtual world is accomplished via typed commands and cursor keys, while output appears as narrative text and a simple ASCII-character plan-view of the participant's immediate surroundings. The text interaction style uses the conventions and interface feel of internet Multi User Domains. (MUDs)

An informal pilot study shows that Prolix provides a usable, effective and enjoyable way to communicate with others using the same access interface. Prolix successfully placed a text based person in two VR worlds, one with another immersive user, the other with another Prolix user. However, in the immorsive setting, the Prolix user's graphical display was soon overwhelmed by the volume of information generated by the moving entities and needed to be frequently refreshed.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGMENTS

I would like to thank a few people I've met along the road to finishing my thesis. These people have my thanks and gratitude for making my life a little bit easier.

- to my advisors William Bricken, Judy Ramey and Tom Furness; without their help, encouragement, and assistance my thesis would have not happened.

- to Toni Emerson, HITLab cyber-librarian, net.goddess, lender-of-SciFi-books, and information gatherer supreme who helped me find relevant (and not so relevant) literature.

- to Max Minkoff & Andy MacDonald for helping with Mercury's immersive body.

- to Geoff Coco for designing and programming VEOS/FERN and his assistance in helping me create the controlling prolix-body.fent.

- to Dan Pirone for helping me with the customization of the VEOS executable code and his assistance in getting multiple users in the same space.

- to Jeff James, Paul Danset, Dan Pezely and Jerry Prothero for their assistance with LaTeX, in which this thesis is written. A special thanks to Jim Fox for his "unofficial" public LaTeX style guide for the UW Grad school thesis requirements, which passed all the bureaucracy and red tape with flying colors.

- to Dan Myers for single handedly encouraging me to pursue a master's degree in the first place.

- to Dav Lion for showing me the proper attitude with which to approach the whole thesis experience, as well as the true meaning of "cut-and-paste" – Scissors and glue – who woulda thought it...

- to Kelly & Darren, Kelly, "Kwit Larj", Pez, Duncan, "Dr. Zoom", Pete, Rey, Steve, Jerome, Zoe, Brent, Sean, Eric, Wendy and all those who kept me social, sane and enthusiastic about my work.

- to my email regulars, Dan, Damian, Mary, Lara, Kate and Jon. Thanks for keeping in touch from so far away – watch for more oddities from alt.tak.weird-stuff.

- to the players and wizards at *Ivory Tower* and other MUDs, for providing hours of enjoyment in "real" text based worlds.

- to my parents and family for their unfailing support in my 2000+ mile chase to pursue a dream, even when it didn't make sense. It's good to know I'll always be welcome back home.

♡ and finally, for Cloe Wright – thanks for everything.

Tak – August 17,1993

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). [12]

ATTRIBUTE-VALUE PAIRS: Paired data stored by FERN. Each entity stores information about itself in attribute-value pairs, with the first data element being a label and the second its value. eg: ("name", "takacs")

CURSES: a set of cursor and screen optimization functions with optimal cursor motion.

CYBERSPACE: "an infinite artificial world where humans navigate in information-based spaces" or "the ultimate human-computer interface." [4]

ENTITY: a logical entry in the virtual world database. Every object in the HITLab's virtual worlds is an entity.

FENT: A FERN entity.

FERN: Fractal Entity Relativity Node. Provides easy database maintenance and access.

FTP: File Transfer Protocol - A method of electronic document retrieval over the Internet.

HITLAB: Human Interface Technology Lab - the research lab under the Washington Technology Center at the University of Washington where this project was researched and built.

IMMERSIVE: See INCLUSIVE.

INCLUSIVE: The "full-blown" VR experience where the user is equipped with stereographic eyephones, earphones and sensors and is made to feel a part of the virtual world, rather than an observer.

IRC: Internet Relay Chat - “[a] type of Computer Mediated Communication System [...] which does not store messages but transmits one person’s typing directly to the monitor of another person or group of people.”[18]

LEX: The UNIX lex command generates programs to be used in simple lexical analysis of text. The input files contain regular expressions to be searched for, and actions written in C to be executed when expressions are found. [17]

LOCALITY: The range of field visible to the user. The clipping range.

LOCALIZED SOUND: the technique of filtering and modifying sound sources heard through earphones so that they sound like they are being produced from a fixed location in space, regardless of head position. This method requires some way to track the head’s position so the sound source can be appropriately filtered so it appears from the desired location.

LPMUD: Lars Pensjo (creator) MUD - “The LP- family of MUDs, including Diku and AberMUD, are usually based on roleplaying adventure games; the players on those MUDs tend to run around in groups or alone killing monsters, solving puzzles, and gaining experience in the quest to become a wizard.”[41]

MACRO: a single user-definable command that invokes a series (usually long) of commands that the user would otherwise have to enter by hand each time.

MERCURY: a participant system for inclusive VR users. Mercury monitors sensors and updates the database with the user’s sensor information. Mercury also displays the correct data to the user’s senses. See Chapter 2 and Chapter 7.

METHODS: “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.”[11] As an example, every entity must have a method called `enter` that defines what that entity needs to do each time it enters a space, and to finally make the actual call to `(fern-enter)`. See section 4.3.1.

MUD: "A MUD (Multiple User Dimension, Multiple User Dungeon, or Multiple User Dialogue) is a computer program which users can log into and explore. Each user takes control of a computerized persona/avatar/incarnation/character. You can walk around, chat with other characters, explore dangerous monster-infested areas, solve puzzles, and even create your very own rooms, descriptions and items."[41]

NON-BLOCKING: The process of politely waiting for a user to input some data without halting all other processes while you wait for the data. This is usually accomplished by entering a loop that performs a fast test to see if any data has accumulated, and if not, continues on with the processing loop.

PARTICIPANT SYSTEM: The main task of a participant system is moving data from the sensors and the applications system to the renderers in a tightly coupled and efficient way.[34]

PERSIST PROCEDURES: "User-defined processes associated with fern entities. An entity can install multiple persist procs which execute in that entity's context. [...] 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 2) be atomic and 3) never block. "[11]

POLHEMUS: a magnetic sensor device that reports a 6D position with regards to a magnetic field generator. Data generated is an "x", "y", and "z" location, and yaw, pitch and roll orientation data. Used to report on the position and orientation of objects in the real world.

PROLIX: adjective. 1) Unduly prolonged or drawn out — too long. 2) marked by or using an excess of words. synonyms: WORDY

PROXEMICS: the study of the nature, degree, and effect of the spatial separation individuals naturally maintain (as in various social and interpersonal situations) and of how this separation relates to environmental and cultural factors.

PS: See Participant System.

PSEUDO-CODE: A summary of code-like statements that serve to summarize what is happening within a section of the program without having to concern the reader or writer with proper syntax. The pseudo-code is often almost in plain english.

REACT PROCEDURES: "React Procs are user-defined functions associated with data-update events. An entity can install multiple react procs which execute in that entity's context when new data arrives from other entities."[11]

STEREOGRAPHIC: technique where 3D images are simulated by presenting each eye with a slightly different view of the object. A variety of techniques are used to prevent the other eye from seeing the other eye's image, ranging from simple color filters found in 3D-comic books, to shielded screens for each eye, through expensive shutter glasses which alternately open and close in synch with the image being shown on the computer screen.

VEOS: Virtual Environment Operating Shell. Provides the distributed computing environment and network communications.

VE: Virtual Environment. Some people prefer his to VR. (See VR)

VR: Virtual Reality. A computer simulation or database visualization technique, usually involving stereographic computer graphics, localized audio, and position tracking devices. There are a variety of other terms that are a matter of preference. (See VE, VW)

VW: Virtual World. (See VR)

YACC: yet another compiler compiler – The UNIX yacc command converts a context-free grammar into a set of tables for a simple automaton which executes an left recursive parsing algorithm. The grammar may be ambiguous; specified precedence rules are used to break ambiguities. The output file, y.tab.c, must be compiled by the C compiler to produce a program yyparse. This program must be loaded with the lexical analyzer program, yylex, as well as main and yyerror, an error handling routine. These routines must be supplied by the user; lex(1) is useful for creating lexical analyzers usable by yacc. [17]

"...it the real world, ... virtual reality still means playing a text-based MUD for five hours with a party of magic users, thieves, and fighters, most of whom work for Hewlett-Packard in their spare time..."[43]

"Inside & Out"
the Stranger *v2n45, 16–22 August 1993*

Prolix *adj.*

1 unduly prolonged or drawn out
too long

2 marked by or using an excess of words
see WORDY
pro.lix.i.ty *n*
pro.lix.ly *adv*

Webster's Ninth New Collegiate Dictionary and Collegiate Thesaurus

Chapter 1

INTRODUCTION

1.1 Overview

The development of virtual reality (VR) requires the skills of many people. Virtual worlds need to be designed and arranged. This requires architectural, planning, and modeling skills. The objects in VR worlds need interesting behaviors programmed into them, from simply existing in a particular space and time, to complex reactive and proactive behaviors. The user's needs and behaviors need to be studied and incorporated into an effective interface. Elaborate system software must be coded for database management, networked communications, physical sensor drivers, graphical and acoustic imagers, and high-level user interface and development packages.

The Human Interface Technology Lab (HITLab) has a fully functional virtual reality system and continues to enhance this as more needs and applications are explored. Prolix is just one part of the many systems that contribute to the VR experience.

1.2 HITLab Background

The HITLab is a relatively young lab, approaching its fourth birthday in September 1993. The HITLab was founded by Professor Thomas A. Furness, who brought 23 years of experience designing virtual reality equipment for the Air Force to the civilian sector and the state of Washington.

The HITLab is a joint state/educational facility, operating under the Washington Technology Center (WTC) and the University of Washington. The WTC provides a link to industry, while the University of Washington provides a steady flow of new ideas and access to academia. The Industrial Engineering Department, in cooperation with the HITLab, offers a Master's degree program in Inter-Engineering focusing on virtual reality technology and human factors research. This generally means the HITLab is staffed by a few permanent researchers, either from academia or industry, and many graduate students from various engineering, arts, science, and education departments.

1.3 Project History

A continuing goal in the HITLab agenda has been the desire to create a virtual common wherein multiple participants can interact in the same collective space. Although various conventional electronic methods (email mailing lists, internal newsgroups) served a purpose, but lacked a certain sense of place and community.

In April 1993, a group of Lab researchers started a project that became known as the "ante-chamber project". A goal of this group was to create the tools needed for a virtual world that would remain online continuously, and serve as meeting place and transfer station between other virtual worlds developed at the HITLab. It was suggested that the ante-chamber be available through dial-up modem, so that people could log in from home and interact with others who might be inclusive or using the dial-up.

In the ensuing discussion, the basics of what came to be known as Prolix emerged. The ante-chamber was to be available to as many people as possible. Some through a simple ASCII/vt100 interface, while others through the normal inclusive experience. A simple split-screen interface, half top-down graphics and half text was proposed. The graphics would be single letter icons representing the objects in the virtual world, with the user navigating with the cursor keys. The text would be provided in the style of MUDs or IRC, with users being able to talk with others and examine their surroundings.

Prolix was first constructed in April 1993 as a stand-alone program not connected to VEOS in any way. In this form, the text display and input was constructed and tested, using `lex` & `yacc` to form an input grammer for the parser. The next area of development was the graphic ASCII display, modeled after numerous god's eye view displays. The first model of movement was a discrete stepping from grid space to grid space as well as a teleportation specification. The resulting program was then turned into a set of library routines and included with the standard VEOS libraries. Debugging then began in earnest on the communication protocols between the participant and the rest of the VEOS world.

In June 1993, Prolix was tested in a simple world containing an ear entity and a mouth entity. When the Prolix user entered the room with the ear and the mouth, the user was able to hear what the mouth was saying, while the ear successfully reacted when the user spoke. This experiment demonstrated that the Prolix user could communicate and interact with other entities in the VE. The graphics were only roughly used at this point.

The next stage of development was to get the Prolix user into a space with an inclusive user. This finally occurred in August 1993. A more detailed account of the abilities of Prolix can be found in chapter 5, which details the informal pilot study.

1.4 Scope

This thesis project provides a working participant system (Prolix) for experiencing virtual worlds at the HITLab through a simplified interface relying on simple vt100/ASCII terminal capabilities and textual narrative. This program is not a virtual world unto itself, but an alternate interface for someone to interact with participants who may or may not be immersed in an inclusive virtual world.

1.5 Audience

This thesis is primarily intended as a technical report on the development of Prolix. Attention is given to the intended use of Prolix, as well as brief consideration of how this tool may enable a community of participants to interact in a single virtual world or place.

1.6 Conventions

As a large part of this thesis is a discussion of the development and coding of Prolix, there is a considerable amount of C code and command syntax. Any time a `command` is mentioned, it is set off from the surrounding text by being printed in this `typewriter` font, with any optional arguments being listed in *italics*. Any `source code` or `function_names()` are also printed in the `typewriter` font.

1.7 Contents

This chapter served as an overview of virtual reality and the environment in which this project was conducted, as well as outlining the scope, intended audience, and contents of the rest of the thesis.

- Chapter 2 is a review of related work and literature, concluding with a list of programming guidelines and specifications.
- Chapter 3 serves as a brief introduction to Prolix, its functionality, and the user interface it presents.
- Chapter 4 discusses the technical details of Prolix. A basic knowledge of C and LISP is assumed, as well as general familiarity with programming techniques.
- Chapter 5 discusses the informal pilot study conducted with a few subjects, and the conclusions based on their experiences and comments.
- Chapter 6 presents application areas that Prolix was designed for or could function well within.
- Chapter 7 talks about the future of Prolix as an application, further areas of expansion, and ideas worthy of consideration.
- Chapter 8 concludes the thesis.
- Appendix A is a copy of the complete Prolix User's guide.
- Appendix B contains a copy of the survey form given to subjects to fill out after the informal pilot study.
- Appendix C has information on contacting the author and acquiring source code.

Chapter 2

RELATED WORK AND LITERATURE REVIEW

2.1 *Introduction*

Prolix has several distinct areas: the input of commands, the display of textual output, the graphical display, and the user interface in general. This chapter presents related work and literature that was used to develop the Prolix program specifications, constraints and guidelines. The first four sections discuss related work and literature in these areas of concern. In addition, there is a section about social computing, electronic communities and Computer Supported Cooperative Work. Each section concludes with a summary of recommendations for its area. Finally, the distilled recommendations from all these sections are presented as program guidelines for the development of Prolix.

2.2 *Literature Review*

This section deals with literature from the fields of cognitive science, languages, psychology, human factors, education, and computer science dealing with issues concerning the choice of interface and the operational design of the program.

2.2.1 *Text Display*

One of Prolix's main functions is to display text that describes the VR world, its inhabitants, and their behaviors. However, it must be emphasized that Prolix does not dictate *what* text is shown, instead it specifies a protocol which enables world designers and the participants themselves to control the content of the messages. So although literature is available about text (linguistics, psychology, cognitive science, authorship, technical communication, creative writing, narrative and theatre), their spatial relationships (Garnham [22]), and the interpretation of the text itself (Glenberg, Meyer and Lindem [24], Franklin [37]), these are not explored in further detail because Prolix leaves control over the content of the displayed text to the designers and participants in the virtual worlds. Prolix only concerns itself with the format of the provided text.

2.2.2 Spatial Representation System

Bryant [8] proposes a Spatial Representation System (SRS) in which he describes how people form and access mental models of spaces through perceptual or linguistic means. The SRS poses a coordinate framework in which a token representation of an object is placed, with information about the object being stored elsewhere and available for lookup. The SRS predicts that a person forms mental models egocentrically, with distances and locations measured with the person as the origin. Bryant states that this SRS model accounts for how well people judge distances and positions.

Prolix attempts to facilitate the formation of a mental model of the VR world the user is inhabiting. Some of the theories of the SRS will be used as guidelines for designing the graphical display so useful information is displayed in an easily assimilated format. These are summarized in table 2.1.

Spatial Representation Guidelines
• egocentric display
• tokens/icons to represent objects
• data available via lookup
• display of coordinates

Table 2.1: Spatial Representation System guidelines

2.2.3 Graphical Display Perceptions

This section of the review focuses on perception of graphics and human limitations.

Graphical Perception

Holland and Spense conducted a study [26] using graphs to compare how subjects judged change and proportion. The experimenters hypothesized that observers are unperturbed by small deltas of graphical change as long as they are given a direct perception of the change, rather than static displays.

Illustrated text

Glenberg and Langston [23] propose that pictures assist in the construction and management of mental models in working memory. Furthermore, mental models support implied relationships in the text. This assists the creation of a representation that is "richer" or more "elaborate" than would ordinarily be available from a representation of the text itself. They designed an experiment to confirm that pictures facilitate the comprehension of text and this facilitation is greatest when the pictures adds information that is difficult to encode or is left implicit in the text. The authors caution against using pictures that focus attention on inappropriate relations and structures. Such illustrations tended to decrease retention and comprehension.

Spotlight of Attention

Adini and Sagi conducted an experiment [1] testing the ability of subjects to simultaneously identify two targets in the visual field. Subjects were successfully able to report the different orientation of two targets when they were within 4 to 8 degrees of visual field. Targets with the same orientation were identified easily over greater than 8 degrees. The results were taken as evidence reinforcing the existence of a grouping process (employing principles of proximity and similarity) operating prior to orientation identification which enabled parallel processing of input chunks.

The results suggest that similarly oriented objects can be widely spaced without impairing identification and classification.

Cluster Detection and Relational Judgement

Liu and Wickens conducted a study [30] to investigate the use of computer graphics and cluster analysis in aiding human relational judgement. They found that in cluster detection task with actual clusters, color greatly facilitated subjects' performance. However, in schemes with no clusters, colors were found to give subjects an unwarranted confidence in identifying clusters where there were none. Unsurprisingly, in a magnitude judgment task, digit displays, rather than clustered color displays, are generally superior to an analog display for precise reading of numerical values.

Derived Graphical Guidelines

Most of the literature reviewed here comes from empirical studies attempting to support or disprove various theories of physical perception limits. Other literature on the layout and design of similar graphical displays can be found in the aerospace literature concerning exocentric viewpoints (aka god's eye views). Prolix attempts to utilize the generalized conclusions and recommendations from these fields and studies in the design and layout of the graphical display. Table 2.2 lists some guidelines for Prolix from the literature in this section.

Derived Graphical Guidelines
• Use motion instead of static images
• Use illustrations and graphics to supplement text
• Similarly orientated objects can be widely spaced without risking identification
• Use digits, rather than graphics, when precise reading required
• correctly used, color greatly enhances identification

Table 2.2: Derived graphical guidelines

2.2.4 User Interface Principles

Shneiderman's book, *Designing the User Interface*, [40] offers a wealth of information and guidelines on designing and programming effective user interfaces. Of special note is a whole chapter on command languages. Distilled in table 2.3 is a list of pertinent guidelines for Prolix.

2.3 Other Participant Systems

This section focuses on past and present work on participant systems. Detailed below are surveys of existing participant systems (text-based and graphical), implementation details, designs decisions, philosophies, future plans, and empirical studies. Included are a thesis explaining the concept of participant systems, reviews of Habitat

Shneiderman's User Interface Guidelines
• hierarchical command structure
• consistent command syntax
• command confirmation
• informative error messages and status displays
• online help and a complete printed User's Guide
• separate windows for text output and command input
• icon representation with detail lookup
• customizable interface for expert users
• hardware independent software
• rapid response time (real-time)
• conduct a usability study

Table 2.3: Shneiderman's user interface guidelines

(perhaps the first commercially successful networked virtual society), IRC, MUDs, graphical systems, and fully inclusive virtual reality participant systems.

2.3.1 Participant Systems

Minkoff introduces a concept called a participant system [34] – an interface to VR systems.

> Virtual reality systems may be considered to consist of an application database of world dynamics and a 3D interface to that database. The world dynamics are generally unique to each virtual world, but the mechanics of the interface remain constant. The separation can be exploited to not only improve the way virtual worlds are built, but to greatly enhance the quality of the participant's experience in virtual reality. In terms of the VEOS model, reading sensors, providing the semantics for data read from them, managing the virtual body, and writing to the renderers is all handled by one component - the Participant System. [34]

The main task of a participant system is moving data from the sensors and the applications system to the renderers in a tightly coupled and efficient way. After fully describing what a participant system should entail, Minkoff explains Mercury, the HITLab's implementation of a participant system.

2.3.2 Habitat

Farmer and Morningstar worked at AMIX when Habitat [21] was built. Habitat was a "many-player on-line virtual environment" whose purpose was the entertainment of its users, or players. Players access the world database through a Commodore 64 front-end that generates a simple animated display and sends commands through to the database. The authors distill their experiences and present a series of assertions surrounded by supporting reasoning and illustrative anecdotes. Among the more pertinent ones are that Cyberspace is necessarily a many-participant system and that the presentation medium is of relatively peripheral concern.

2.3.3 Nethack

Nethack [31] is a dungeon exploration game employing a graphical display as well as a text command input line.

Games are often presented as the ultimate and least forgiving usability test. No one *has* to learn how a game works, or put up with features or quirks they don't like. If it's not usable, it won't be used. Period. Whereas some users of business software will endure a bad interface to use a particular feature.

Since the game Nethack offers the closest approximation to what Prolix is intended to represent (without the multiplayer and communication aspects found in MUDs), many of Nethack's better features and solutions can be used as guidelines for Prolix.

In Nethack, the graphical display is strictly ASCII based, though ports to more graphical environments do exist. The user looks down from above (ie: plan view), with each item of interest represented as a single character icon. Players can identify unfamiliar icons by issuing an `identify character` command, which gives the name of that object. Fuller descriptions exist only in the documentation. Movement is controlled by cursor keys (or by alternate characters on the QWERTY keyboard) and follows a discrete movement model, where each keypress moves you a single space in that direction. The graphics display is removed when the player issues commands that produce large text displays like inventory or spell lists. Small status displays are built into the main display, providing constant access and updates on commonly used information. All of these features contribute to the usability and success of Nethack.

2.3.4 IRC

The following comes from Internet Relay Chat (IRC) documentation, Reid's paper [18], experiences with IRC, and guides. IRC provides a system where world-wide computer users can engage in real-time communication with each other via typing at their keyboards. Users select nicknames and join a channel, where everything they type is displayed to everyone else joined to that channel. Up to 1400 on-line users are typical, talking on one or more of 500 daily changing channels.

IRC, as a text based chat system, offers many features that enable the user to control with whom they converse. Prolix would like to offer its users similar conversational capabilities, and could use the better features of IRC as guidelines. Since talking is what IRC is all about, anything typed is interpreted as a message you would like others to see and is displayed to them, prefaced by name. Interface and system commands are all prefaced with a leading "\" to set them apart from conversation. Users are also given a command that allows them to send messages that appear as actions, rather than spoken words. These are commonly used to lend an atmosphere (ie: "Tak smiles") to the conversation. Users can also see a list of current conversants and information about them.

2.3.5 LPMud-VR

Andersson [15] discusses the design and implementation of the LPMud driver and language, a popular system for creating (and interfacing with) virtual worlds on the Internet. In this interface, as many as 100 simultaneous users interact with each other and the world which they help build through a "simple" text-based interface. At the conclusion of the paper, Andersson discusses four conclusions for VR technologies based on his experiences:

1) that for multiple independent users, the only possible timeflow system is a linear time system, preferably real-time;

2) the user must feel as if they are controlling a character or vehicle, as striving to place the user inside the world shatters the believability of the world, at least with today's sensor and display technology;

3) the locality, or clipping range, should be a function of the world or application, not hard coded into the interface;

4) the users should be able to add to the world by creating new items.

2.3.6 MUD Clients

A "bare bones" way of connecting to any text-based virtual reality is through the Internet protocol known as `telnet`, which provides the basic communication between remote computers. However, `telnet` offers limited user features. Alternative clients have been placed in the public domain and provide friendlier interfaces and specific Mud features. Some of the features often incorporated into client services include a history buffer, separate input/output areas, local editing of files, text processing (highlighting and filtering), auto-login, macros for frequently performed actions, logging, and a small degree of programmability. Smith [42] reviews a large collection of mud clients for every conceivable type of computer. Following is a list of current (as of August 1993) publicly available clients: Tinytalk, TinyFugue, TclTT, LPTalk, SayWhat, PMF, TinyView, TINTIN, MUD.el, TinyTalk.el, LPmud.el, CLPmud.el, MyMud.el, TINT, DINK, FooTalk, RispTalk, MUDDweller, MUDCaller, BSXMUD Clients.

The author has tried (and uses) many of these, studying the interface and code from the most successful to avoid reinventing the wheel for text-based communication interfaces. The client features listed above have been used as guidelines for the design of Prolix.

2.3.7 MIRE

Kay has taken a TMI LPMud driver (a popular alternative driver developed by The Mud Institute) and used it as the basis for a multi-user news and information retrieval system. The MIRE [28] project combines a USENET feed with an MIT Media Lab filtering program called DoppleGanger to produce customized newspapers. MIRE also supports internal mail and conference rooms, all through the modified TMI-driver. Kay's paper gives an overview of Muds and News in general, as well as the DoppleGanger system before discussing the MIRE system. The rest of the document discusses design decisions and compromises, and then goes on to explain some of the modifications to the driver necessary to create MIRE.

2.3.8 Interactive Gopher

Masinter and Ostom [33] have used the multi-user aspects of a MOO (a variety of MUD) to allow several people to collaborate on a information retrieval run over the Internet service known as "gopher".

2.3.9 MUDs Grow Up

After a brief definition of Muds as text-based VR, internally extensible, and inherently multi-user, Curtis and Nichols describe two research projects that stem directly from experimentation with a MUD known as LambdaMOO. (See Section 2.4.3: Social Phenomena in Text-Based Virtual Realities)

The first project is known as Astro-VR [14], and serves as a teleconferencing system for NASA/JPL astronomers who wish to use the MUD's real-time communication and self-contained email. The Astro-VR contains several custom features, including an Image library, a presentation editor and viewer, and access to standard astronomer's tools.

The second project, the Jupiter System [14], is a teleconferencing system for internal use by Xerox-PARC and Euro-PARC. The basic MUD driver has been enhanced to support audio, video and window-based graphical user interfaces on Xerox-PARC's internal network. Xerox-PARC's workstations come equipped with sound by default and video cameras can be added. The internal geography of Jupiter reflects the actual office and conference areas. The Jupiter System hopes to enable Xerox-PARC workers to telecommute while maintaining the casual social interactions in the workplace via the video feeds placed throughout the building.

2.3.10 BSX

The following comes from the author's own limited experiences with the BSX mud, as well as text provided with the BSX-mud client. BSX was developed to be an X-windows graphical interface for a version of LPMud that was extended to support graphics. A player has a front-end client which maintains stock bitmaps and a connection to the main database server. The displays are animated, but only to the extent that objects appear and disappear as they leave the area and/or sight. Commands are completely texted based, as well as output. The graphics serve only to supplement mental imagery.

2.3.11 MIVE

Anderson, a summer 1991 HITLab intern from Sweden, was interested in supplying virtual world users with a variety of interfaces to the world, from a full fledged inclusive experience complete with as many senses as possible, down to a text-based interface. At that time, VEOS wasn't mature enough to support these interfaces. MIVE [16] was an addition that added two new interfaces to the VR worlds. The text-based system operated on a version of the LPMud world building specifications, modified to support a connection to VEOS. The second was a protocol standard and client that used a screen based renderer to produce images for these worlds.

MIVE was successful in proving that VEOS was able to support different types of clients. Currently VEOS/FERN handles many of the concerns addressed by the MIVE prototype as a integral part of its world building functionality.

2.3.12 Workstation body

The following comes from the author's own experiences with the mouse body and spaceball system while at the HITLab. Each of these systems provides a workstation-based graphical display window of the virtual world. Movement through the world is driven by more traditional and common interface devices such as a mouse or spaceball.

2.3.13 Mercury

MacDonald and Minkoff, researchers at the HITLab, explain that a virtual world requires three major components: the database containing information about the elements in the world, the system to support the database, and an interface for the user to interact with the database. Mercury provides the interface by monitoring the sensors that track the user's physical state, sending this information to the database, and displaying the data in the database appropriately. Mercury [32] [34] doesn't need a specific computer system on which to run, thus freeing valuable resources for what they do best: the interface running as fast as possible while communicating with the database, sensors updating the database whenever possible and display renderers running unencumbered with participant software.

2.3.14 Guidelines Derived from Related Work

The literature reviewed for this section was rich in technical detail. Most of the articles were written by the designers or maintainers of the given participant systems. I've gathered the best features, implementations and the designers' recommendations and listed them as table 2.4.

Guidelines Derived from Related Work
• tightly coupled data movement
• constant status displays
• discrete movement model
• aliases and macros
• atmosphere commands to enhance conversation
• allow creation from within the world
• real-time reactions
• locality should be adjustable
• separate input and output windows
• history buffer
• input filtering
• session logging
• programmability
• use graphics to supplement textual narrative

Table 2.4: Guidelines derived from related work

2.4 Virtual Communities and Communication

The reviews in this section focus on the communities and social aspects found on various virtual communities. The literature includes several papers on the evolution of communication without traditionally important physical cues, demographics on the users, discussions of unique phenomenon appearing through the communities, and a few papers drawing relationships between recreational use of these facilities and commercially viable opportunities. The literature for this section, as well as the previous section on other participant systems, is available almost exclusively through electronic means. This is not surprising for a culture rooted so deeply in cyberspace.

2.4.1 Electropolis

Reid [18] opens with a short introduction to the internet and description of an elaborate CB-style chat program known as IRC. Reid discusses how a text-based chat medium lacking physical cues affects communication between its users, followed by a discussion of the community and social norms that have arisen to compensate for the lack of these important cues.

2.4.2 Conversational

Serpentelli's two part paper [39] first discusses some issues pertinent to computer users, including the hacker mentality, gender differences, video games, email, bulletin boards, and multi-user systems, including IRC and MUDs. In the second part of the paper, Serpentelli conducts a pilot study logging conversations in different mediums (MUD, IRC and Notes - a mail/bbs system) and categorizing the utterances. According to these data, there were significant effects for the type of statements occurring based on the purpose of the room. For example, a higher number of greetings, physical actions, techno-babble and interactions with code occur on the MUD.

2.4.3 Social Phenomenon in Text-based Virtual Reality

In his often cited paper Curtis [13] discusses his experiences creating and maintaining a mud called LambdaMOO for over a year while at Xerox PARC. A fundamental part of MUDs is its multi-user and extensible aspects, which requires players to somehow distinguish themselves from other players in this anonymous medium. Choice of name, gender and textual description serve to identify a player. Curtis observes that players spend their time socializing, exploring the rooms and objects created by other players, and coding their own contributions. Other MUD activities include large gatherings for parties, weddings, games and puzzles. The typical player is considered to be a male college undergraduate from a wide range of disciplines. The anonymity and physical safety of the MUD frees some people from their conversational shyness, while inspiring others to be rude and obnoxious without fear of physical retaliation. Others become so taken with this form of communication that they often spend large amounts of time connected. There are reports[35] of players being continuously connected for 35-48 hours, neglecting their work and physical lives.

Curtis notices a distinct social hierarchy, with advantages for those who have the power and talent to create, as well as those with distinct personalities. Those in charge of creation are frequently called upon to enforce the rules of the community.

Curtis also notes that it's quite easy to strike up a conversation, because simply being there is an implicit invitation. He coins a term "social gravity" to identify the tendency of people to move towards clusters of people with the assumption that interesting conversations are occurring there. Some tactics different from physical social interaction are related to the peculiarities of the medium. "Pretending not to hear", is based upon the commonly understood fact that a person may not be in front of the terminal and so cannot respond to a message. This can be used to avoid talking without offending people. "Gagging" is to process of suppressing text that contains certain patterns and can be effectively used to censor text and whole personalities. "Cognitive time-sharing" is a skill that most players soon develop during conversations. Because response times can be slow due to typing speed and net-lag it is often possible to conduct several different simultaneous conversations. It is not uncommon to be involved in one conversation face-to-face and one or two more conducted via remote communications commands like `tell` or `page`.

Curtis concludes that MUDs seem to provide a necessary social outlet and hold potential for new audiences in professional settings.

2.4.4 Communicative

Carlstrom [9] approaches the study of the culture on a MUD from an ethnological standpoint, treating the MUD as a distinct and different culture. Brief mention is made to some differences in communication techniques beyond those discussed in other papers. First, there is no sense of proxemics on MUDs, everyone is simply there, though occasionally some spatial relationships are implied by wordings. Secondly, silence carries comparatively little meaning in an on-line dialog because of network delays, typing speed and even people not being at their terminal to respond to a comment. Finally, there is almost no concept of turn-taking.

2.4.5 Identity

Bruckman [2] proposes that MUDs can be used by their players as a form of "Identity Workshop", where the person experiments with different aspects of their personality in a safe environment that encourages this expression. Issues of gender swapping, addiction, net-romance and real romances are also discussed.

2.4.6 MicroMUSE

Brown [10] reports on the history of a MUD called MicroMuse, from its inception to its present day incarnation at MIT as a research project. Its stated goals are education, recreation, communication and transfer of ideas, as well as a visionary model of a future society.

2.4.7 Ethnography

Rosenberg [36] relates his experiences on one specific MUD, WolfMOO. The paper consists of a description of typical type of characters and areas, as well as a geography of the virtual terrain and a look at the social and political structure of the MUD and its administrators.

2.4.8 Group-Ware

Takacs [45] draws a parallel between the definition of Computer Supported Cooperative Work (CSCW), or Group-Ware, and MUDs. Group-Ware is defined from the literature as supporting multiple users, providing support for a common goal or task, and providing an interface to a shared environment. A brief explanation of MUDs is then given, and the parallels examined, revealing remarkable resemblances.

2.4.9 MudReport

Bartle [3] conducts a methodological review of many MUDs then available in the UK on a commercial or public basis, as well as some Internet Muds. The study was commissioned by British Telecomm plc to serve as a first step on assessing the commercial possibilities of MUDs. In addition to the somewhat dated reviews, Bartle addresses some issues like playability, motivation for play/dropout, addiction, and financial prospects. He notes that the average player is not willing to pay very much for current systems.

2.4.10 Guidelines Derived from Social Interaction Studies

The literature presented in this section covers a variety of domains, including papers from an ethnographic, historical, social, psychological, commercial and academic standpoints. Though coming from many different standpoints, the authors all acknowledge that there is a strong sense of community present in these systems, much more than might be expected from such a limited transmission medium. It is hoped that Prolix will provide an interface to today's developing virtual worlds that will foster such a sense of community and remove the sterile, museum-like atmosphere found in most virtual worlds today. In this regard I have gathered the best features, implementations and the designers' recommendations and listed them as table 2.5.

Guidelines from Social Interaction Studies
• atmosphere of a location influences behavior there
• establish and customize the identity of users
• cognative time-sharing occurs
• different social norms appear and shape themselves
• silence has a different meaning
• a social space has commercial appeal

Table 2.5: Guidelines derived from social interaction studies

2.5 Program Guidelines

The research and literature reviewed in this chapter cannot be considered definitive, but it does provide a strong sampling of the work, especially in the area of previous text based communication systems. This review has served to derive guidelines for constructing a usable and effective text based interface to a traditionally graphic VR environment. Table 2.6 and table 2.7 summarize the key features needed to be embedded in Prolix given the background principles, theory and recommendations reviewed above.

Prolix Design Issues
• text content left to designers and participants
• atmosphere commands to enhance conversation
• conduct a task analysis study
• conduct a usability study
• atmosphere of a location influenuces behavior there
• establish and customize the identity of users
• cognative time-sharing occurs
• different social norms appear and shape themselves
• silence has a different meaning
• a social space has commercial appeal

Table 2.6: Prolix design guidelines

2.6 Summary

The goal of this literature review was to develop a consistent and usable interface for Prolix, following the recommendations and experience of other researchers. No system exists in a vacuum, and the review of previous work gave a grounding for what worked in the past, what was needed in the future, as well as providing a guide to common pitfalls. Through reading about the described systems (and in the case

Prolix Programming Guidelines
• use motion instead of static images
• use illustrations and graphics to supplement text
• separate windows for text output and command input
• display of coordinates
• icon representation with detail lookup
• constant status displays
• separate input and output windows
• egocentric display
• use digits, rather than graphics, when precise reading required
• similarly oriented objects can be widely spaced without impairing identification
• hierarchical command structure and consistent syntax
• command confirmation
• online help and a complete printed User's Guide
• informative error messages and status displays
• customizable interface for expert users
• locality should be adjustable
• aliases, macros and pragrammability
• input filtering
• history buffer and session logging
• real-time reactions
• hardware independent software
• allow creation from within the world
• tightly coupled data movement
• discrete movement model

Table 2.7: Prolix programming guidelines

of clients, the actual use and study of provided source code) , I hope to place many of the best features of each in Prolix.

Chapter 3

PROLIX USER OVERVIEW

3.1 Overview

This chapter presents the basics of Prolix. The first section explains the general concepts; the second, driving influences for the selection of the interface and its stylistic choices. The next section illustrates the screen layout of Prolix, with a final word on the operational guidelines pertaining to navigation, communication, database manipulation and the interface itself. A complete description of Prolix and its features is described later in Appendix A:Prolix User Guide.

3.2 General Concept

Prolix is intended to be a participant system that allows entry into virtual worlds traditionally only accessible with fully inclusive participant systems. The users of Prolix interact with the virtual world through the keyboard and cursor keys, and receive display information through the screen, using only ASCII/vt100 capabilities. It is hoped that a wider range of people will be able to experience some semblance of VR if the interface avoids the higher end graphics systems such as X-Windows, or the complications of various specialized home computer platforms.

3.3 Interface Style

A goal of the HITLab is to create a virtual community with its own personality and flavor. Yet most of the worlds created, while interesting, are sterile and museum-like. None of these worlds yet created have the style and feel of a community such as a comfortable local cafe, bar, or even the sense of electronic community found on many MUDs or IRC [2], [18], [39], [44]. What's missing are the *people*. Because Prolix's hardware requirements are so commonly available, I hope that its widespread use will encourage many people to enter the current VR worlds. Having many people present and communicating within the same virtual space should allow the space to acquire an identity associated with the people who frequent that space.

3.4 Screen Layout

The typical Prolix screen is split into two major sections, the graphical display and the text display. Each of which is further divided in two: the graphics display has a plan and side view, while the text display has a separate input and output window. (See figure 3.1)

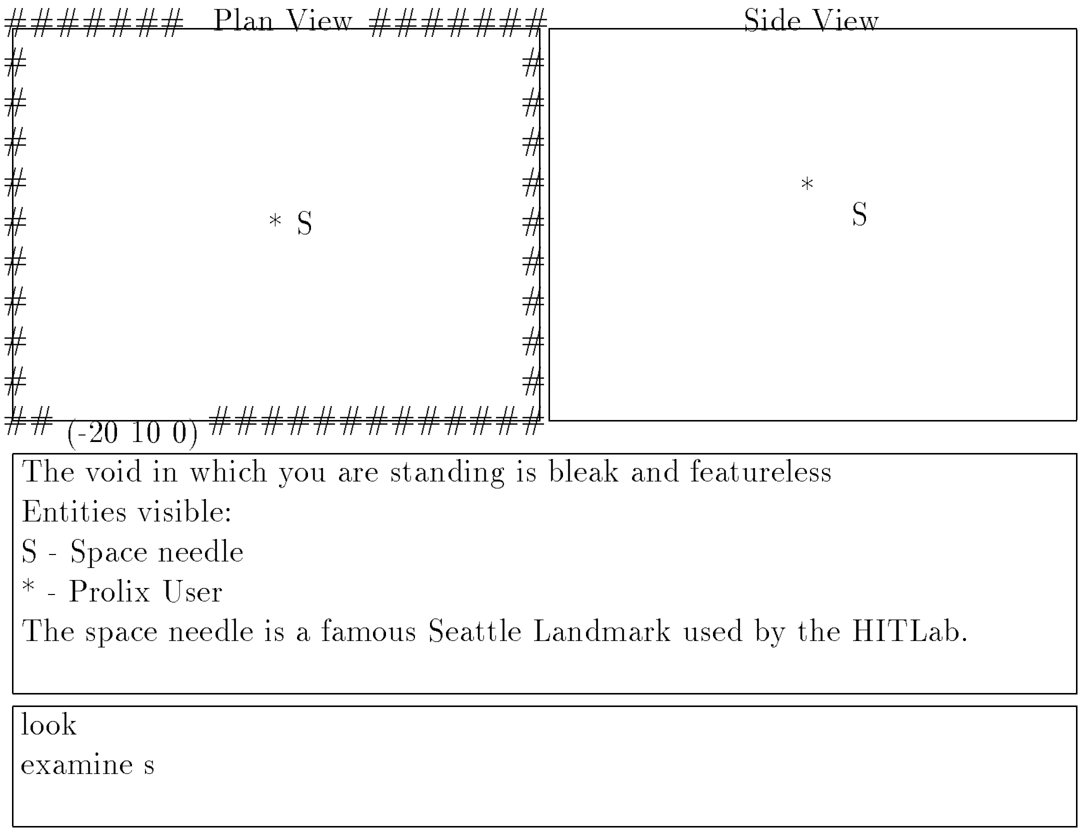


Figure 3.1: The text display

3.4.1 Text Display

In its initial configuration, the text display takes up the bottom half the screen, with three to five lines (adjustable at compile time) at the bottom of the screen allocated to echo the user's typing input. (See figure 3.1) The text output appears in the output windows and scrolls upward until it reaches the top of the output window. The default output window size ends at the beginning of the graphics window. It is possible to eliminate the graphics window entirely, in which case the output window expands to encompass the entire window. In this mode, the up cursor key recalls the last typed line of input.

The results of user issued commands, text output from events in the VR world, and behaviors and communications from other participants and entities appear in the output window. To avoid cluttering the display, Prolix doesn't echo the user's commands to the output window, just their results or a feedback message.

3.4.2 Graphic Display

In its initial configuration the graphics window takes up the top half of the screen. (See figure 3.2) The left half of the display shows the plan view, while the right side displays an orthographic side view. In the default configuration, each discrete character-wide space on the display grid represents a 1x1x1 meter volume. A single character icon is placed on the screen according to the coordinates of the object in the VR world. The plan view window shows the entities' `x` and `z` coordinates, while the side view window shows the missing `y` coordinate as well as the `x` coordinate. The HITLab's VR worlds follow a protocol where the center of the world is defined te be (0,0,0), with `x` as left and right, `y` as up and down, and `z` as depth into the scene. Mentally combining the information provided by the two views allows you to place the object in the world, as well as its elevation or altitude. If an object moves, its icon moves in both windows simultaneously.

In figure 3.2 the "*" represents the user, who is located slightly to the left of and above the "S", which is the base of the space needle at the center of the VR world coordinates(0,0,0). The user and the space needle are separated by at least 20 meters horizontally in this example.

In the default configuration, your icon is free to move about the display until it reaches a configurable radius from the center of the display, at which point the display is re-centered about the user. The other mode fixes your icon in the center of the display and any movement commands cause the world to go scrolling by.

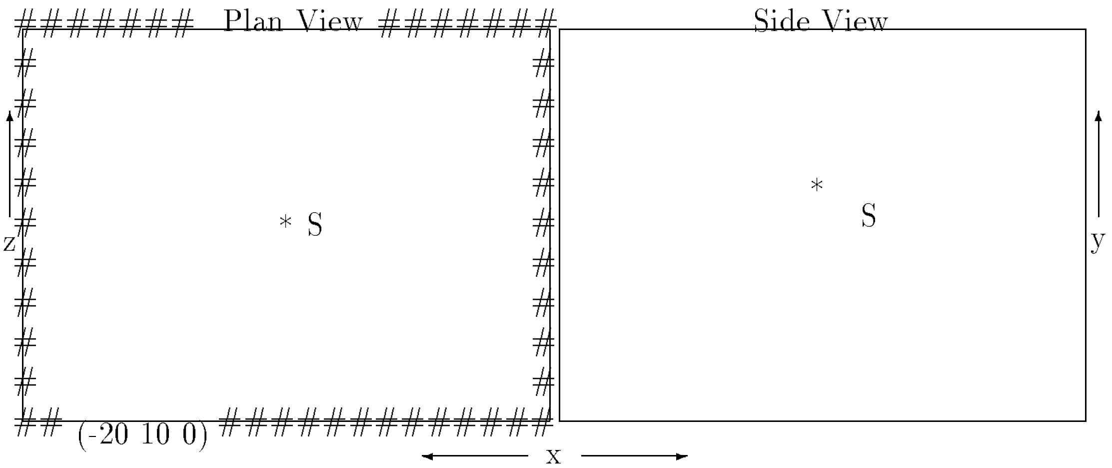


Figure 3.2: The graphics display

3.5 Operation Guidelines

This section gives a brief description of some of the more common commands and their intended uses.

3.5.1 Navigation

Navigation through the VR world with Prolix is accomplished by the use of the cursor keys. Prolix starts with the plan view graphical display as the active window. The active graphical display window is outlined with "#" marks, rather than dashes.

For the sake of clarity, I will assume you are in a world where the plan view represents looking down on a space where the top of the screen is north, bottom is south, and left and right are west and east. The side view has the top of the screen as up, the bottom of the screen as down, and the left and right sides as west and east. See figure 3.3.

When you hit the cursor-right key, your icon moves one space east in each of the windows. If you use the up-cursor or down-cursor while the plan view is active you will move north or south, respectively. However, if the side view is active, pressing the up-cursor or down-cursor moves your icon up or down in elevation. In figure 3.3 pressing the up-cursor will move you north. When your icon comes near the edge of the screen, you are repositioned in the center of the display and the screen is redrawn. The percentage of the screen on which you can move around is a user preference and

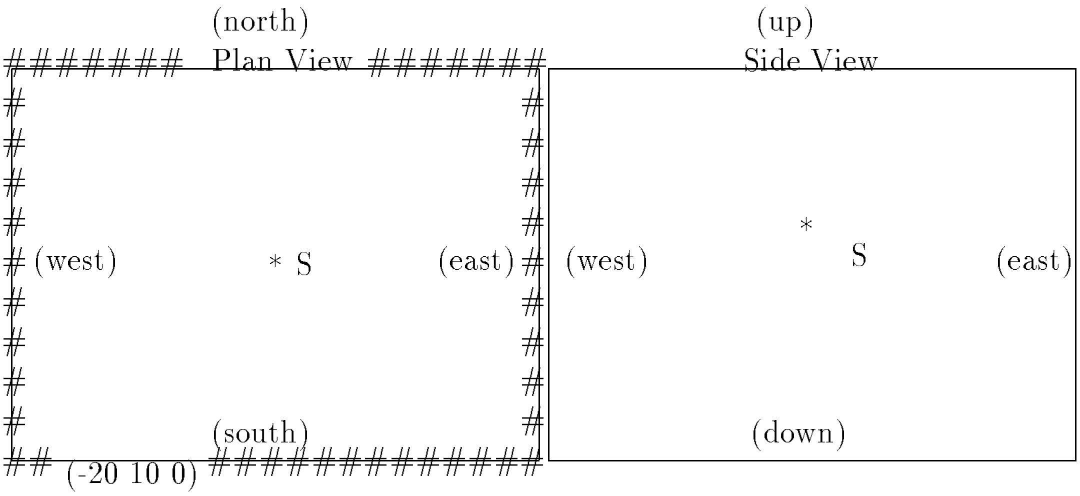


Figure 3.3: Movement orientation

is set with a command called `sweet` and defaults to 100%, which determines how close your icon can get to the edge of the screen before being repositioned. Setting this value at 0% causes the user's position to be fixed at the center of the screen, where subsequent use of the cursor keys causes the screen to scroll past in that direction.

A teleportation command called `goto` is provided to allow you to instantly jump to some known location. Providing the name or icon of another entity as arguments to this command moves you directly to that entity. If no argument is provided, you are moved to the absolute world coordinates (0,0,0).

3.5.2 Communication

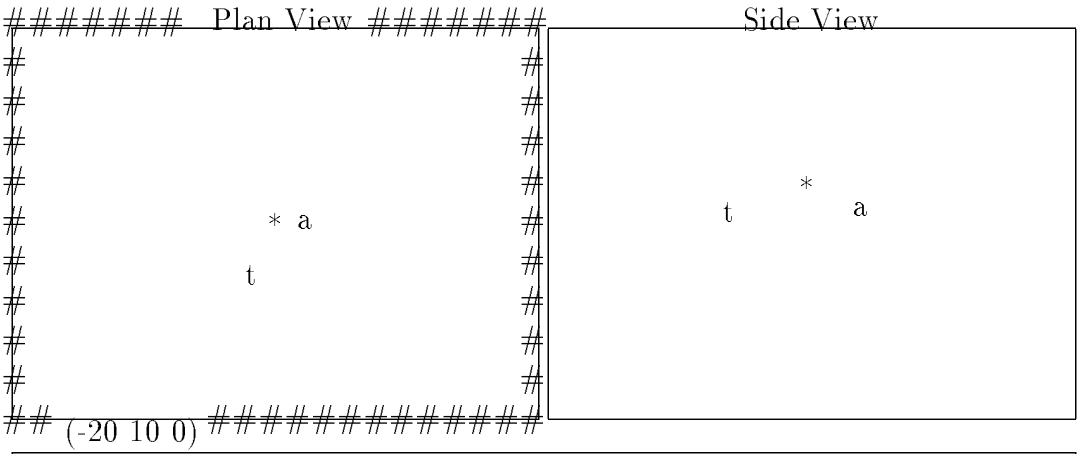
Communication in the VR world with Prolix is accomplished mainly through the `say` and `tell` commands. The `say` command causes anything you type to be passed out into the VR world for other entities to see and/or process. Other entities receive information on who sent the message and the message's content. It is up to the particular entity to decide how to process this information. Any Prolix interface will automatically display incoming text messages.

An example session will serve to illustrate the essential points. In figure 3.4, the user has issued the `who` command, (line (a) in figure 3.4) which gives a listing of all the participant systems in the VR world (the line marked with (1)).

Then the user decides to ask a question using the `say` command (b) so that it is distributed to everyone. Appropriate feedback (2) is given in the output window.

The user known as Teklon replies with a behavior (3) known as an `emote`. The `emote` command prepends whatever text you type with your name. In this example, Teklon has typed `emote shrugs` to produce the output marked with a (3). In this manner, your character can appear to do many things that lend atmosphere and style to the conversation.

Reading Ari's reply (4), the user decides to reply directly to Ari with the `tell` command (b)(5), which is only received by the person to whom it is addressed (Ari). Other users (like Teklon) remain unaware that anything is even happening. Technically, everyone receives all the messages, but each individual Prolix entity performs filtering functions (see chapter 4 for more details). Ari replies in kind (6).



(1) 3 Users online: Ari (a), Teklon (t), Tak (*)
(2) You say: Anyone know when the meteor shower starts?
(3) Teklon shrugs.
(4) Ari says: I saw a posting about it.
(5) You tell Ari: I think it starts at midnight..
(6) Ari tells you: I think you're right.

(a) who
(b) say Anyone know when the meteor shower starts?
(c) tell ari I think it starts at midnight.

Figure 3.4: Communication in Prolix

There are several other ways that Prolix allows you to shape your communication, including `shouts`, `echos`, and `aliases`. These are discussed in appendix A.

3.5.3 Database Manipulation

Database manipulation is the term for altering the database that you keep about other entities and the information you disseminate to them.

Each VR entity has a collection of attribute-value pairs that can be modified by that entity. Figure 3.5 shows a typical entity. Prolix provides a way to change the traditional values (visibility, name, etc), as well as specific fields recommended by the Prolix protocol.

```
('name' 'space needle')
('visible' t)
('wireframe' nil)
('teleportable' t)
('picture-desc' 'needle')
('alias' 'needle')
('icon' 'S')
('short' 'The Space Needle')
('long'
 'The Space needle is a Seattle icon and a defacto VR
  object as well.  It rises many meters overhead and
  is built from hopefully sturdy supports. Two elevators
  provide access to the observation deck at the top.')
```

Figure 3.5: Example entity

The `icon` command allows you to change the icon that is displayed on the graphics display of other Prolix users. For the benefit of any inclusive users that may be in the world, Prolix provides the `pict` command which takes as an argument the filename of your new graphical description file.

The `name`, `title` and `short` commands allow you to change the information shown to other Prolix users when they ask for information about you. The `name` command allows you to change your name.

Normally when other users `look` or issue a `who`, they are provided with the short description of each entity. The short description is composed from two subfields, the name and title field. For example, the user "tak" with the title "- the lowly peasant" have a short description of "tak - the lowly peasant" which would be displayed as a result of a `who`, `look`, or `examine`. The `short` command enables complete control of your short description. Using the short description, you could appear as "a rock." when people `look` in the room or view the `who` list.

The current version of Prolix allows you to alter anything in your database, including your name, icon and other identifying data. As with any great flexibility, there is usually a compromise somewhere. This great flexibility assumes the users trust other users not to maliciously masquerade as someone else. If prolonged usage of Prolix shows this to be a problem area, the command set can be restricted so that key identification data is unable to be modified.

3.5.4 Interface Configuration

Prolix offers its users the option to customize the interface and get status information about the current state of the program.

As mentioned in section 3.5.1, Prolix has several variables governing how movement in the graphical display works, including a command to control the amount of VR world space each screen grid space represents. This command is called `grid`, and takes a numerical argument that changes the space represented by each grid square. The grid size defaults to one.

Another variable that the user can set is the `sweet` percentage. This is the percentage of the screen (measured from the center) that the user can move about on without causing a screen refresh. The default sweet percentage is 100%, enabling the user to advance to the edge of the screen.

Other commands of interest are the `help` command, which prints a brief (but complete) listing of all commands. The `prolix` command shows a list of various interface command variables such as the `sweet` percentage, the grid size, the active window, your screen and world coordinates, and other miscellaneous information about the displays.

There are several control sequences that are useful during operation. The most useful one is perhaps the refresh (`<control-l>`), which discards all optimization information and draws the screen over again from scratch. There are a number of other control sequences, but the only other one that needs to be mentioned here is the `quit` command, which is triggered by typing `quit` or by the `<control-c>` sequence.

3.6 Summary

This chapter has served to explain Prolix. The interface was designed to allow the maximum number of people into the VR worlds at the HITLab. The interface was specifically constructed to encourage communication among people by providing a way to exchange real-time dialog for anyone with a vt100 display and a keyboard. The screen layout and command structure were explained and illustrated, with examples of navigation, communication and a brief discussion of the more important beginning database manipulation and interface commands.

Chapter 4

PROLIX INTERNALS

4.1 Overview

This chapter explains the inner workings of Prolix at the code level. After outlining the system level requirements, I discuss the levels of control, ranging from the topmost level of FERN control of the entity, down to the C code level of the Prolix library routines. Another section discusses the parser and lexer that accepts and processes the user's text command input. Finally, an example illustrates the process from user input all the way to another user's screen.

4.2 System Requirements

Prolix uses the `curses` screen and cursor optimization routines provided with UNIX systems to manage the screen and text. Prolix has been simultaneously developed on an Ultrix V4.2 (Rev. 96) system and on SunOS v4 systems as a library extension to VEOS and should compile on those systems without modifications. Because Prolix is a library addition to VEOS, its use requires VEOS which, at present, only exists for Ultrix, Sun4 and IRIX.

4.3 FERN control

The participant system's representation in the VEOS's VR database is just another FERN entity, like any other entity. Since it is very costly for the LISP-coded prolix-body.fent to do all the processing, the fent makes calls to custom Prolix C-functions that are included in the VEOS library.

When the prolix-body.fent is first started, it calls the Prolix library routine called `prolix-init()` which sets up the user's screen and initializes the user's keyboard. A corresponding function, `prolix-close()`, is called when requested to close the screen and restore the keyboard.

In general a fent is controlled by "methods", "react procedures", and "persist procedures". Each time a fent is given some processing time, it runs its persist procs. The methods are called upon request and the react procs are called in response to a change in a specific attribute the fent is perceiving. Sections 4.3.1-4.3.3 briefly discuss Prolix's methods, reacts, and persists. A more general description of these FERN protocols can be found in *fern.doc*. [11]

4.3.1 Methods

Methods are normally called by other fents when they want you to do something. For example, when you collide with a portal, you don't call the code to enter that portal yourself, you request that the portal perform the action for you by perhaps calling a method in the portal called something like `setup-entrance`. This method might perform some world initialization tasks and then call `enter` in the participant.

The Prolix body has no specific methods aside from the standard methods all VEOS entities have. As applications are developed using the Prolix body and people want to make requests to the body, methods can be added.

4.3.2 React Procedures

React procedures, known as react procs, are called in response to a change in the VR database. When an entity starts up, it registers which attributes it is interested in hearing about with VEOS via the `fern-perceive` command.

```
(fern-perceive emote :react (lambda (ent text)
                        (prolix-incoming-emote ent text)))
```

Figure 4.1: A prolix-body.fent "react proc"

The react proc in Figure 4.1 is from the prolix-body.fent, which is interested in hearing about an attribute called "emote". Whenever any attribute-value pairs with an attribute of "emote" are posted to the VR database, the prolix-body.fent reacts by calling the function `prolix-incoming-emote()` and supplying the entity that posted the attribute-value pair (ent) and the value (text). By convention, any function calls prefaced with the name of a library addition ("hg" for Mercury, "im" for the imager, and "prolix" for Prolix, etc) are assumed to be calls to C library routines. In this case, the prolix-body.fent does very little. It simply passes the needed information down to the Prolix code.

A discussion of how the prolix code processes the various `prolix-incoming-` calls can be found in Section 4.4.6.

4.3.3 Persist Procedures

Persist procedures, known as persist procs, are called whenever it becomes an entity's turn to process. Because the persist procs are part of the entity's turn, they must be designed to finish as quickly as possible to avoid deadlocking that entity with a long loop. With this in mind, the prolix-body.fent has one persist proc, which does three things, all of which are designed to quickly return a value with as little processing as possible. This persist proc gets called whenever VEOS devotes some processing time to this entity.

```
(defun prolix-body-frame ()                     ;; persist proc
  (let ()
    (if prolix-quit (progn                      ;; exit test
                      (prolix-close)            ;; exit sequence
                      (fern-close)))
    (prolix-keyboard)                           ;; keyboard input
    (if (setq attrs (prolix-comm))              ;; check changes
          (mapcar 'fern-put.attr attrs))        ;; then post 'em
    (prolix-render)))                           ;; render changes

(fern-persist '(prolix-body-frame))             ;; persist proc
```

Figure 4.2: A prolix-body "persist proc"

In Figure 4.2 the function is defined and then entered in the prolix-body.fent's list of persist procs.

The function definition for `prolix-body-frame` starts by testing the variable `prolix-quit` which is set to `TRUE` by another function if the user wants to quit the program. If the user has triggered a quit, then `prolix-close` is called to close down the text and graphical display, after which `fern-close` removes the prolix-body.fent from the virtual world.

If the user has not yet quit, three function are called in sequential order, starting with `prolix-keyboard()`, then `prolix-comm()`, and ending with `prolix-render()`.

The `prolix-keyboard`() function is a non-blocking keyboard fetch. The function doesn't wait for the user to hit any keys, but checks to see if any keys have been hit since the last time it was called. If the user has not hit any keys, the function immediately returns. If there was keyboard activity, `prolix-keyboard()` updates the internal database and returns. For more details, see Section 4.4.3.

The `prolix-comm()` function controls communication between Prolix's internally maintained database and the VEOS virtual world database. When the function executes, it picks up all the attribute-value pairs that have been prepaired for exporting and returns them in a list. If there is anything in this list, the `fern-put.attr` call posts these new attribute-value pairs to the rest of the VEOS VR database. For more details see Section 4.4.4.

The `prolix-render()` function is responsible for rendering the graphical display. The function consists of a single pass through Prolix's internal entity database, checking each entity to see if it should be drawn. Since Prolix uses the `curses` screen optimization library functions, only changes in the graphical display are drawn unless the user requests otherwise. For more details see Section 4.4.5.

4.4 Prolix control

Most of the Prolix library routines are called from the prolix-body.fent. These include the five functions `prolix-init()`, `prolix-close()`, `prolix-keyboard()`, `prolix-comm()`, and `prolix-render()` as well as the various `prolix-incoming-` functions from the react procs. Section 4.3 discusses when these functions are called and the following sections discuss what happens in each of these functions.

The remaining Prolix functions are triggered by the user's commands, and include a set of various routines called `outgoing_()` functions. These functions are discussed in section 4.6.1.

4.4.1 prolix-init() function

The `prolix-init()` function is called once when the prolix-body.fent is first initialized. Its responsibility is to setup the user's screen and keyboard, and initialize the internal database.

```
initscr();                   /* init stuff               */
noecho();                    /* curses call              */
nonl();                      /* don't xlate newlines     */
cbreak();                    /* don't process input      */
raw();                       /* don't interpret ANYTHING */
echo_off();                  /* custom call (no_block.c) */

init_prolix_db_terminal();   /* fill in the dbase blanks */
open_windows();              /* open all the windows     */
```

Figure 4.3: prolix-init() interface & terminal setup

In Figure 4.3 the `curses` routine `initscr()` is called first, which initializes default `curses` values and determines what capabilities the user's terminal has.

This call is followed by a set of `curses` routines. The first two turn off echoing and newlines because the standard screen isn't used by Prolix, input and output are each redirected into their own window. The call to `cbreak()` allows Prolix to handle how the backspace and delete characters are processed. Once again, this is necessary because of the custom windowing. The call to `raw()` places the user's keyboard into raw mode, which allows Prolix complete freedom in how it interprets the key hits, including signals like interrupt, quit, and suspend.

The call to `echo_off()` places the user's terminal into a state known as "no_block" or "no_delay" mode. In the normal mode, when the program requests a character from the keyboard, the program waits until a character is actually typed. The design of Prolix requires that each function should finish quickly. Entering "no_delay" mode makes character request calls return a `NULL` if nothing has been typed.

The function `init_prolix_db_terminal()` allocates space and assigns default values for the prolix interface structure, which contains information about the status of the interface – screen size, display mode, etc.

```
in = newwin(INPUT_LINES,COLS,LINES-INPUT_LINES,0);
scrollok(in,TRUE);
if (!in) {
    fprintf(stderr,Unable to create input window. Exiting.\n);
    closedown();
}
```

Figure 4.4: Detail on open_windows()

Figure 4.4 shows the `curses` calls that creates one of the custom windows. Prolix opens four windows (technically more are opened, but the extras serve only as borders or backdrops), an input and output window, as well as a top and side view window. Only the code for the input window is shown, the others being very similar.

The call to `newwin()` opens a window and stores a pointer to that window in the variable `in`. The function needs four arguments, the first is the number of lines, followed by the number of columns, the screen line on which to begin, and the column on which to begin. The values in CAPS are #`define`d by either `curses` or Prolix to reflect the dimensions of the user's terminal. The call to `scrollok()` tells `curses` that it can scroll the `in` window when printing a newline at the bottom of the window requires a scroll. The `if` statement checks to see that the window has been created successfully. If not, it prints a message and closes down Prolix.

After the input, output, and two graphics windows are opened, `prolix-init()` makes a series of `curses` calls that actually draw the windows onto the user's screen.

4.4.2 prolix-close() function

The function `prolix-close()` is called whenever the users requests it via the `quit` command or the key sequence `<control-c>`, or when a non-recoverable error occurs. Its responsibility is to close up all the windows, and restore the user's keyboard and terminal to their original state.

In figure 4.5 the call to `close_windows()` leads off the function by cleanly closing

```
close_windows();                   /* close windows politely      */
mvcur(0, COLS-1, LINES-1,0);       /* put the cursor at the end   */
(void) endwin();                   /* close down Curses           */
echo();                            /* curses call to reinit       */
echo_on();                         /* re-enable normal key stuff  */
```

Figure 4.5: The prolix-close() function

all the windows. Three `curses` calls follow: `mvcur()`, `endwin()`, and then `echo()`. The `mvcur()` function forces the cursor to be placed at the bottom of the screen, no matter where it was last left by `curses`. The `endwin()` function closes down `curses` and frees memory associated with the windows. Finally, the `echo()` restores the terminal's normal echo capabilities, while the `echo_on()` is a custom function that restores the blocking capabilities of the terminal.

You may note that `exit()` is not called. This is because prolix is controlled from within VEOS and a call to `exit()` would bring the user all the way back out to a system prompt instead of just back to the VEOS prompt.

4.4.3 prolix-keyboard() function

The `prolix-keyboard()` function is a part of the prolix-body's persist proc and is called repeatedly, as often as VEOS devotes processing time to this entity. Its responsibility is to process every single keypress the user makes and perform the associated command.

The code for `prolix-keyboard()` is a single `switch()` statement, but with a lengthy list of `cases`, so pseudo-code is used for readability.

The `switch()` statement calls the function `getch_no_hang()` which checks the keyboard for key hits and assigns the return value to `ch`. If there were no keys hit, the value of `ch` is set to the #`defined` value of `ERR`. The rest of the code is a long series of `case` statements which compare the value of `ch` to the specified value after the case.

For convenience, I have collapsed the middle case statements into categories as seen in Figure 4.6. The first set of case statements handles various interface and edit-

```
switch (ch=getch_no_hang())
{
    case ERR:                   /* no keys hit yet */
        break;
    if (editing-or-control-sequence)  handle-controls;
    if (cursor-keys)                  handle-movement;
    default:
        oldkey=ch;              /* save character              */
        text_buffer[c++] = ch; /* enter char into input stream */
        wprintw(in,'%c',ch);   /* manually echo char to input  */
        wrefresh(in);           /* redraw the input window      */
        break;
}
return;
```

Figure 4.6: prolix-keyboard() pseudo-code

ing control sequences. The second set of cases handles cursor keys, which serve as the primary means of moving around if the display is enabled, or as a basic history buffer if not. The next few sections detail the four basic parts of the `prolix-keyboard()` function.

no-keyhit

The first `case` is important and is reproduced in actual code (Figure 4.7), rather than pseudo-code. This simple case statement checks if `ch` has a value equal to `ERR`, which in most cases it will, since `prolix-keyboard()` is called far more often than a person can type, even holding down a key. Checking this case first assures that the function will finish as rapidly as possible.

default

The last case is the default handling of keyboard input for typed commands and is shown in its entirity in Figure 4.8. First the current character is saved for use

```
case ERR:           /* no keys hit yet */
   break;
```

Figure 4.7: prolix-keyboard() no-keyhit

by other parts of the function. The current character is then placed into a string which will eventually be passed to the parser (see Section 4.4.3). The `wprintw()` call to `curses` manually echos the typed character to the input window, while the `wrefresh()` actually displays the window.

```
default:
    oldkey=ch               /* save character        */
    text_buffer[c++] = ch; /* enter char into input */
    wprintw(in,'%c',ch);   /* manually echo char    */
    wrefresh(in);          /* redraw input window   */
    break;
```

Figure 4.8: prolix-keyboard() default

Cursor Movement

Operating the cursor keys is the most common form of movement. Figure 4.9 displays the code for just the cursor-right `case`. The other three cursor keys have similar code, with the exception of a small `if` statement for the up/down keys that takes into account the active window. This extra code simply controls which variable, the `y` or `z`, gets updated according to the graphics window that is currently active.

The first `if` statement checks to see if the graphics display is even activated. If not, no actual work needs to be done, so the key press is logged and control returned.

Otherwise, processing continues by advancing the "x" position stored in `Pos.x` by the value of `prolix.grid`, which stores how much space one grid represents. A counter is advanced that keeps track of how close the user is to the edge of the screen. If it's over the established threshold stored in `prolix.sweetx`, `center()` is called to place the participant in the center of the graphics display, repaint the screen, and reset the counter.

```
case CURSOR_RIGHT:
   if (prolix.display) {
     Pos.x += prolix.grid;
     prolix.sweet_count_h++;
     if (ABS(prolix.sweet_count_h) >= prolix.sweetx) {
          center();
     } else {
          prolix.me->draw = TRUE;
          prolix.me->oldx = prolix.me->x;
          prolix.me->oldy = prolix.me->y;
          prolix.me->oldz = prolix.me->z;
          (prolix.me->x)++;
     }
     update_borders();
     outgoing_6D(Pos.x, Pos.y, Pos.z);
   }
oldkey = ch;
break;
```

Figure 4.9: prolix-keyboard() cursor-right

If no centering is required, the current position becomes the old position and the new coordinates are assigned. Finally, the borders are updated so the coordinate status display is refreshed. The new position is then sent to `outgoing_6D()`, which is responsible for preparing the new 6D in a format that the `prolix-comm()` will recognize.

Interface control & editing

The handling of control and editing characters is another long list of `case` statements. Figure 4.10 lists the control and editing cases. All of these cases perform simple tasks described below.

```
case CTRL_C:                /* not caught in raw() mode */
case CTRL_C:                /* not caught in raw() mode */

case CTRL_L:                /* refresh screen           */
case CTRL_D:                /* toggle display on/off    */
case CTRL_B:                /* toggle big/small window  */
case CTRL_O:                /* toggle active window     */

case CTRL_M:                /* newline/return handling  */
case BACKSPACE:             /* line editing            */
case DELETE:                /* line editing            */
case CTRL_U:                /* delete the input line    */
```

Figure 4.10: prolix-keyboard() control & editing

Signal Capturing

Since Prolix operates in "raw" mode, the `<control-c>` and `<control-z>` cases serve to capture the break and suspend signals. In the case of `<control-c>`, Prolix wants to clean up nicely before exiting, and when it does exit, it doesn't want to call `exit()`, but simple return to the VEOS prompt. This is handled by first calling `prolix-close()`, and then issuing its own `<control-c>` signal interrupt so that VEOS can capture it and leave the system in a VEOS prompt. In the case of capturing a `<control-z>`, no special processing is required, so the `case` simply issues a `<control-z>` interrupt for VEOS to capture.

Interface commands

Some interface commands are triggered by control sequences. In choosing which characters to represent various functions, I tried to choose similar control functions from popular applications and utilities.

The `<control-l>` sequence (used by Emacs, `vi` and some UNIX shells) triggers a screen refresh. The `case` statement simply calls a series of `curses` routines that are specially designed for refreshes.

The `<control-d>` sequence ("d" for display) toggles the graphical display on or off. The `case` statement toggles the value of a boolean variable `prolix.display` and then calls the function `display_off()` or `display_on()`. The function `display_on()` is a collection of calls similar to those in figure 4.4 where a window is opened. The `display_off()` function uses the `curses delwin()` routine to remove the graphics windows.

The `<control-b>` sequence ("b" for big) toggles the text display so that it either takes up the whole screen (scrolling under the graphics display if that is enabled) or the default bottom half. The `case` statement works similarly to the `<control-d>`, except that the functions called consist of closing the old text window and re-opening it with its new configuration.

The `<control-o>` sequence ("o" for other, similar to the EMACS command `C-x o`, which switches to the other window) toggles the graphics display so that the other window is active for movement. This is accomplished by a setting an interface variable which controls how the cursor keys work in the `prolix-keyboard()` function. See section 4.4.3 for details.

Text Editing

Some basic text editing commands are provided with Prolix. The current version of Prolix supports delete and backspace processing, a limited history buffer, and a clear-input-line command.

The most basic command editing is signaling when you've completed a command. This is done by hitting `RETURN`. The newline is placed in the input string, followed by a `NULL`. The resulting string is then placed in the history buffer and then passed to the lexer and parser, which evaluates the command for validity, takes the proper action and returns. (See Section 4.5.1 and 4.5.2) After this, the current input string is erased by placing the index counter at zero, so the next characters typed will form a new string.

The backspace and delete control sequences must be manually interpreted because Prolix operates in "raw" mode. As mentioned above, the input command is constructed by individually adding characters to a string. When a backspace or delete is caught, the index counter is decremented by one. The result of this is that the next character typed is stored in the input string over the "deleted" character. A `curses` routine is called to first get the current cursor position in the input window, remove that character from the screen, and reset the cursor. Some formatting and boundary checking is done to handle deletes or backspaces beyond the edge of the screen.

A `<control-u>` (used by UNIX shells) acts as a shortcut and deletes the whole command input line. This is accomplished by setting the input index counter to zero, effectively erasing the current string. A `curses` routine, `wclrteol()`, is called to clear the input window to the end of the line.

4.4.4 prolix-comm() function

The `prolix-comm()` function is a part of the prolix-body's persist proc and is called repeatedly, as often as VEOS devotes processing to this entity. See section 4.3.3 and figure 4.2 for a reminder of when this function is called.

The only responsibility of this function is to return a LISP list data structure to the prolix-body.fent. This list contains any attribute-value pairs that are to be distributed to the rest of the VW through VEOS by means of a `(fern-put.attr)`.

As a design decision, Prolix maintains a global data structure called `prolix.XPort` that conceptually acts as a "mailbox". Whenever the Prolix user does something that the rest of the VR world needs to know about, it sets a flag in `prolix.XPort` according to the type of the data to be communicated (text, 6D, or boolean attr-vals) and stores the data in the mailbox.

The vast majority of the time, the "mailbox" will be empty. And when there are issued commands, there will never be more than one piece of data in the box, as this function is called much more frequently than it is possible to enter commands (even single key commands like the `<cursor-keys>`). This observation eliminates the need for costly list processing code in the function.

Figure 4.11 shows `prolix-comm()` as pseudo-code. In keeping with the FERN protocol of being atomic and non-blocking, the first thing this function does is determine whether or not there is anything for it to do. If not, it returns a `NIL`, which signals the controlling prolix_body.fent (figure 4.2) that there are no attr-val pairs to be put into the VR world.

```
if ((prolix.XPort_comm_flag) ||      /* Check 'mailbox' for comm? */
    (prolix.XPort_bool_flag) ||      /* any boolean toggles?      */
    (prolix.XPort_6D_flag))          /* any movement?             */
    {
        create-a-LISP-data-structure;
        if (prolix.XPort_comm_flag)   Process-comm-data;
        if (prolix.XPort_bool_flag)   Process-boolean-data;
        if (prolix.XPort_6D_flag)     Process-6D-data;
        return (pResult);
     } else {
        return NIL;                    /* nothing to report       */
     }
```

Figure 4.11: prolix-comm() pseudo-code

Otherwise, the function creates a LISP data structure to store the value it is about to construct. It then performs a series of `if` statements that determine what type of data is in its storage space and processes that data into the LISP data structure and returns it.

Each data type requires a slightly different method of processing, though they are all eventually stuffed into a LISP list. The 6D data deals with the movement and placement of the user's body or other 6D attributes. The boolean data processing deals with preparing `t` or `nil` values. The communication processing handles any other situation by converting the data in the mailbox into the two text strings of an attr-val pair.

Figure 4.12 shows the actual code for what occurs in what figure 4.11 referred to as "process-comm-data". The code for "process-bool-data" and "process-6D-data" is similar in concept.

Firstly, this chunk of code resets the `comm_flag`, and then initializes a temporary LISP list to `NIL`. The actual data, perhaps a "say" and a "hello", is stored in `prolix.XPort_attr` and `prolix.XPort_value`, respectively. The two calls to the function `cvstring()` convert the "say" and "hello" variables of type `*char` to the LISP string type. These new values are added onto the temporary list `pList` with the

```
if (prolix.XPort_comm_flag) {
    prolix.XPort_comm_flag = FALSE;     /* communications sent  */

    pList = NIL;                        /* () in LISP           */
    pAttr = cvstring(prolix.XPort_attr);
    pValue= cvstring(prolix.XPort_value);
    pList = cons(pValue, NIL);          /* (<value>) in LISP    */
    pList = cons(pAttr, pList);         /* (<attr> <value>)     */
    pResult = cons(pList, pResult);     /* ( (<attr><val>) )    */
}
```

Figure 4.12: prolix-comm() processing communications detail

`cons()` to form a proper FERN attr-val pair. This temporary result is added onto the `pResult` LISP variable, where it will eventually be returned and sent out into the VR world. Section 4.3.3 details how the prolix-body.fent's "persist proc" broadcasts this string as a `(fern-put.attr ('say' 'hello'))`.

4.4.5 prolix-render() function

The `prolix-render()` function is a part of the prolix-body's persist proc and is called repeatedly, as often as VEOS devotes processing time to this entity. Its responsibility is to determine if any parts of the graphical display need updating and, if so, to draw them in a fast and efficient manner.

Of the five main control functions of the entity, the code for `prolix-render()` is the most complex. It is represented entirely in pseudo-code as figure 4.13.

Before the code in figure 4.13 is even called, two quick checks are made to see if a call to the renderer is even necessary. If the graphics display has been turned off, or there were no renderable changes (logged by the commands and the `prolix-incoming-()` functions) since the last time, the loop is not even entered.

Once inside the main loop, Prolix checks each entity to see if its position has changed, moving to the next entity as soon as it is determined that the current entity is not to be drawn. If an entity needs to be redrawn, its old image on the

```
while(entities) {
   draw_me = FALSE;             /* don't draw any entity by default */
   if ((refresh) || (draw(ent)) ||        /* check reasons to draw  */
       (!clipped(ent)))  draw_me = TRUE;
   if (!visible(entity)) draw_me = FALSE; /* never draw invisible   */
   if (clipped(ent)) erase(ent);          /* erase clipped entities */
   if (draw_me) {
      erase(ent);                         /* erase old spot         */
      get_screen_coords(ent);             /* where is it now        */
      draw(ent);                          /* draw in new spot       */
   }
}
```

Figure 4.13: prolix-render() pseudo-code

screen is erased (replaced with whatever was there before), the new screen coordinates computed, and the entity's new position is drawn.

The `erase()` and `draw()` are implemented as calls to `curses` primitives, and are simple and uninteresting.

The `clipped()` function returns whether or not an entity is within the bounds of the graphical display. This is accomplished by comparing the entity's current VR 6D with the 6D of the screen's current center. If the entity's coordinates are too far from the center of the screen, the object is marked as clipped. There is no check for occlusion by other objects or partial obscurement, as the Prolix interface handles entities with an iconic representation that assumes all objects are "point sources".

The `get_screen_coordinates()` function computes an entity's screen coordinates based on its present 6D in the VR world. In order to accomplish this, Prolix maintains the 6D coordinates (even though this version of Prolix does not use the orientation information) of the center of the screen, as well as how many units of VR space each grid space represents. The equation used to convert the `x` component of an entity's 6D to the proper screen coordinates is shown in figure 4.14. The function first computes a delta from the center of the screen using the 6D data for each of the entity's `x`, `y` and `z` coordinates separately. The offset from the screen center is then computed by dividing this distance by the grid size and adding the result to the screen coordinates of the screen's center.

```
e->x = (( ((e->sixD.x - ScreenOrigin_6D.X)/grid)) +
        ScreeOrigin.displayX);
```

Figure 4.14: Conversion from 6D to screen coordinates

As an example, I'll assume the screen is centered on (10,10,10) in VR space and the entity in question is at (20,-10,20), with each grid representing 10 units of virtual space. The screen's center in screen coordinates is (15,15), indicating that it's 15 lines up and 15 characters out from the bottom left corner. Looking at the `x` component first, the entity's 6D minus the screen origin's 6D ($20 - 10 = 10$) divided by the grid size (10) is 1. If the screen's origin is (15,15), the entity's new screen `x` component is 16 ($1 + 15 = 16$). The `y` and `z` values are computed in a similar manner, which has been left as an exercise for the reader.

4.4.6 prolix-incoming-() functions

The `prolix-incoming-()` functions provide the communication bridge from the FERN VR world to the screen based interface. They are responsible for conversions from LISP to C data structures and then calling appropriate display and/or database update functions.

Each attribute that the Prolix user is interested in perceiving from the VR world has a corresponding `prolix-incoming-()` function that is called whenever the controlling prolix-body.fent perceives it. For example, when the `emote` attribute is perceived by the fent (figure 4.1), the function `prolix-incoming-emote()` is triggered.

Currently, Prolix has `prolix-incoming-()` functions for the following attributes: `name`, `alias`, `say`, `tell`, `shout`, `echo`, `emote`, `short`, `long`, `visible`, `6D`, and `icon`. All the functions are very similar, so a specific example will serve to illustrate the others.

For the example, I will examine the `prolix-incoming-emote()`. Suppose someone emotes the message that "tak smiles". This message would be received by the prolix-body.fent's react procedure as illustrated in figure 4.1 as the FERN/LISP data structure (`'emote' 'tak smiles'`) along with the id of the sending entity.

LISP passes this code through an extraction and conversion routine that extracts the ent-id and converts the LISP structures to their corresponding strings. These values are then passed to a function called `incoming_emote()` with the ent-id and string “tak smiles” as arguments.

Figure 4.15 shows the complete code for `incoming_emote()`. The first thing done is a lookup of the given Ent-id in Prolix's local database. If the entity is not found, then `FetchEntity()` calls another function to create a new entry in the local database. If the entity issuing the “tak smiles” happens to be me, then repaint the screen (*not* a `refresh`, but a `repaint`, which actually prints the output) and exit. Otherwise print the message and leave (`word_wrap()` calls `repaint` itself).

```
e = FetchEntity(EntId);   /* find entity in Prolix's database  */
if (e==prolix.me) {       /* if we sent this message           */
   repaint_screen();      /* redraw the screen and DON'T print */
   return TRUE;           /* it again. Its been done.          */
}
word_wrap(NULL,str,NULL,COLS-2); /* Otherwise print it out     */
return TRUE;
```

Figure 4.15: incoming-emote() function

The other type of command received is a database update, like a `name`, `alias`, `short`, `long`, `visible`, `6D` or `icon` message. These functions also consist of a call to `FetchEntity()` to look up and possibly create a new entry. After getting the entity's Id, the function proceeds to update it and possibly set a flag noting that a graphical change needs to take place during the next render loop.

Figure 4.16 shows the `incoming_icon()` code and illustrates a database update command.

4.5 Command Interpretation

> In programs with structured input, two tasks that occur over and over again are dividing the input into meaningful units, and discovering the

```
e = FetchEntity(EntId);    /* Look up/or create local db entry */
e->icon = str[0];          /* assign icon to entity            */
prolix.redraw = TRUE;      /* force a redraw next time through */
renderable_change(e);      /* note that a change happened here */
return TRUE;
```

Figure 4.16: incoming-icon() function

> relationship between the units. This division into units (which are usually called tokens) is known as *lexical analysis*. [...] As input is divided into tokens, a program needs to establish a relationship among the tokens. This task is known as *parsing* and the list of rules that define the relationships that the program understands is a *grammar*. [27]

The input expressions accepted by Prolix follow a simple "verb object" or "command argument" syntax. Prolix accepts abbreviations and supports the command conventions developed by the Internet community over the last six years.

A large portion of a Prolix user's time will be spent entering text commands. The processing and interpretation of these commands is therefore an important part of Prolix. The following sections describe the lexer and parser that I generated using the UNIX `lex` and `yacc` utilities.

4.5.1 Lexical Analyzer

As noted above, the lexer's main task is to take an arbitrary stream of characters and break it up into meaningful chunks, called tokens. The UNIX programming utility `lex` allows you to specify a series of regular expressions that define what pattern of characters has meaning and assign that string to a token. When a string matches a regular expression, the lexer is said to have recognized the pattern, otherwise it is considered rejected. The programmer need only provide the regular expressions and their corresponding tokens, and `lex` will generate the code to recognize those tokens.

To invoke the generated parser, the programmer makes calls to `yylex()` until the entire input stream is recognized or rejected. After all the input has been accepted, the lexer will have returned a string of tokens that represent the input. This string of tokens is input to the parser.

4.5.2 Parser

The parser's job is to recognize a syntactically valid command or reject a series of tokens that represent the user's input. The UNIX programming utility `yacc` allows you to specify a grammar that defines what patterns of tokens constitute a valid command. The programmer need only provide a specification for the grammar and the parser will be generated.

The parser is called when Prolix has received a complete input string, which the user signals by pressing `RETURN`. The pertinent parts of this part of the input processing are shown in figure 4.17.

```
case CTRL_M:                        /* HIT RETURN                 */
   t_buffer[c++] = '\n';            /* put the newline in t_buffer */
   t_buffer[c] = '\0';              /* end the string politely    */
   strcpy(h_buffer,t_buffer);
   tbi = 0;                         /* changed by parser at input()*/
   first = TRUE;                    /* set CMD state 4 parser     */
   while(t_buffer[tbi]) yyparse();
   c=0;                             /* restart buffer             */
break;
```

Figure 4.17: Accepting an input stream

The first few lines in figure 4.17 end the input string (collected a character at a time) and copy the input (via `strcpy()`) into the history buffer. The next two lines initialize some global variables needed by the parser.

The `while()` command is deceptively short for all the work it invokes. This command feeds each character into the parser, which in turn feeds these characters to the lexer in the hopes that they will form recognizable tokens that the parser can act upon. After the lexer returns a set of tokens, the parser attempts to recognizes a valid command and trigger the appropriate action.

In the case of the communications or database update commands, the parser passes the message to the `outgoing_()` routines described in section 4.6.1. The interface commands are handled by their own set of routines which consist exclusively of information displays (program status queries) or internal database updates (interface customization commands).

4.6 Example Command Processing

Now that I have described all the parts, it would be useful to assemble the pieces into a whole and see how a sample command would work its way through the system.

4.6.1 Outgoing Commands

As an example of a user issued command, I will trace the actions of an `emote` command through a somewhat simplified trace, starting from the character input, all the way to its outcome of being broadcast into the VR space.

Assume I have typed "emote smiles happily.", intending to produce the result "Tak smiles happily" on other user's screens. The `prolix-keyboard()` routine described in section 4.4.3 accepts and assembles the input string character by character. When I type `RETURN`, the string is passed down and eventually reaches the parser.

Here the "emote " string matchs the first regular expression rule listed in figure 4.18. Note also that the rules for recognizing an `emote` permit it to be abbreviated by typing a ":". I could have produced the same result with ":smiles happily.".

The rest of the text ends up being recognized as the token `WORDS`. The complex regular expression that recognizes a string of text as `WORDS` is just an enumeration of acceptable character input separated by spaces.

Once the lexer finishes, the string "emote smiles happily" has been reduced to the two tokens `EMOTE` and `WORDS`. The parser then examines the grammar specification to see if the tokens it has just received represent a valid input.

In figure 4.19, the parser starts at the `sentence` and examines the list of things that make up a sentence. One of these is an `emote_phrase`. An `emote_phrase` consists of the token `EMOTE`, perhaps followed by some `text`, which is subdefined as being a `WORD` or `WORDS` token. Since the tokens we have are `EMOTE WORDS`, we recognize this input as a valid command and perform the C function `outgoing_emote()`, shown

```
<CMD>emote[ \t]* |
<CMD>:[ \t]* {
               strcpy(yylval.sval,yytext);
               BEGIN ARG;
               return EMOTE; }

<ARG>[ \t]*[-_=+`~!@#$%^&*()}{:;'\\|/?.,a-zA-Z0-9 ]+ {
                          strcpy(yylval.sval,yytext);
                          BEGIN CMD;
                          return WORDS;         }
```

Figure 4.18: lexer recognizing an "emote"

```
sentence        : say_phrase
                | emote_phrase
    [...long list deleted....]
                ;
emote_phrase    : EMOTE        { outgoing_emote(NULL); }
                | EMOTE text   { outgoing_emote($2);   }
                ;
text            : WORD
                | WORDS
                ;
```

Figure 4.19: parser recognizing an "emote"

in figure 4.20.

```
int outgoing_emote(char *str) {
   if (!str) {
        wprintw(out,USAGE: emote|: <text>\n); /* usage error    */
        return FALSE;
    } else {
        strcpy(temp_string, prolix.me->name);   /* massage string */
        strcat(temp_string,  );
        strcat(temp_string, str);
        word_wrap(You ,str,NULL,COLS-2);      /* output self    */
        export(emote,temp_string);            /* output others  */
return TRUE;
    }
}
```

Figure 4.20: outgoing-emote() function

The `outgoing_emote()` performs several duties. If an incomplete command has been issued (if I had typed just "emote") I would be shown an error message showing the expected usage. Otherwise, the outgoing message of "Tak smiles happily" is constructed from the given string "smiles happily." by prepending my chosen name and a space. A confirmation message is echoed to me with the `word_wrap()` function. The constructed string is then sent to `export()`.

`export()`'s only duty (not shown here) is to place the two strings given as arguments in the mailbox and set a flag so that the `prolix-comm()` function discussed in section 4.4.4 can broadcast the message to the rest of the VR world.

4.6.2 Incoming Commands

As an example of an incoming command, I will trace the actions of an incoming data item through a somewhat simplified trace, starting from the FERN data structure all the way to its outcome of being shown on each user's screen. As most of this was detailed in section 4.4.6, the rest of this section is a summarized account of the other section.

I will consider the same command "emote smiles happily." that I traced in section 4.6.1. Currently, this command has been issued to the VR world by FERN. Now we will look at what another Prolix entity does to receive and display the message.

During the prolix-body.fent's react proc, it receives a LISP data structure item `('emote' 'tak smiles happily.')` from FERN. Section 4.3.2 details this process. After identifying the entity that sent the message, the fent passes the information to a C call as detailed in section 4.4.6. This process is responsible for the conversion and manipulation of the data and finally the display on the user's screen.

4.7 Summary

This rather lengthy chapter has described the functioning of Prolix in detail, starting from the original control by the prolix-body.fent and proceeding to the lexical analysis of the input stream. At the end, I have attempted to provide a holistic view of the program's operation by tracing some sample input.

Chapter 5

INFORMAL USABILITY STUDY

5.1 Overview

This chapter describes an informal usability study conducted using the Prolix interface to determine its utility and performance. The sections below present a description of the study, the subjects and their comments, and a discussion of the results of the study.

5.2 Description

This section describes the two phases of the informal study, as well as the study's goals and criteria for success.

5.2.1 Tag Space

In the first phase of the study, an inclusive subject and a Prolix subject are placed in a VR world where they can play "tag". One subject is designated as "it" and chases down and touchs the other person, which will switch the "it" status and the game continues. The immediately recognizable game allows the subjects to spend time doing something in the intrinsically multi-player environment, without any explanation of what they're supposed to be doing.

The immersive subject tags the other subject by flying through the Prolix subject's mannequin-like graphical representation. The immersive subject has visual cues to determine when they are "it" – in this study the sky turns yellow when the immersive subject is "it" and blue when the immersive subject is not "it". To further illustrate who is "it", there is a "hat" placed on "it". The Prolix subject could see the tag take place on the graphical display, as well as being informed with a text message – "Prolix user is IT!". At any time, the subject could `look` at their environment and see who was it because their name had an "IT!" appended to it.

5.2.2 Chat Space

In phase two, the inclusive subject switches to a Prolix interface for the purpose of testing the interaction when multiple people use the text interface in the same VR world.

The prolix subjects sit at two different remote terminals and joined the same space. (Until additional software is written, a limitation in VEOS prevents more than two participants in a space.) They are in a common space with a graphical description and a few objects that they can examine. The subjects are also be given the chance to communicate through a real time text exchange, using Prolix's communication features.

For this study, the simple "chat" world did not contain many world dynamics or objects and may at first have seemed like a fancy implementation of the UNIX `talk` utility. The "chat" world contains manipulatable objects and descriptions of them and the space that the participants can query. This concept of a pre-existing space qualifies even this simple "chat" world as a VR world.

The subjects were not given an explicit task. They were told to explore the VR space in which they found themselves, and to experiment with the communication commands.

5.2.3 Study Goals

The informal usability study was conducted with four goals in mind.

1) to verify that the interface was usable.

2) to see if the interface was effective at conveying the substance of what went on in the VR world.

3) to see if the text interface uniquely offered anything that the immersive interface didn't.

4) to receive comments from subjects who had not been a part of the development effort.

Usability

In order to test the usability of Prolix, subjects were observed as they used the interface. Of particular note were signs of confusion, number of errors committed, number of times the help function or quick reference sheet was used, and their questions to the experimenter. The interface's usability was graded poorly if the subjects exhibited obvious signs of frustration with the interface, or it failed them.

Effectiveness

The effectiveness of Prolix in this informal study was assessed subjectively, primarily relying on the comments and observed reactions of the subjects. The interface was judged as effective if the subjects were able to play an enjoyable game of tag and/or communicate effectively with another Prolix subject.

Uniqueness

Uniqueness of the interface (or what the subjects thought was unique) was ascertained from the subjects' comments during their use of Prolix or from the comment sheet.

Comments

When subjects are introduced into a new system, unexpected things tend to happen. A comment space was provided for the subjects to express thoughts and ideas. Perhaps the most important observations are gathered in this section, particularly the comments on the conversational ability provided by Prolix from those subjects who haven't used real-time internet text-based interfaces (MUDs, IRC, etc) before.

5.3 Subjects

The subjects for the informal usability study were chosen either for their availability, past experience with text-based interfaces or interest expressed in the project. These selection criteria are in no way intended to represent the population in general or any other usability testing standards. In fact, the study was conducted to assess quickly the usefulness and point out successful features and potential problem areas of Prolix.

5.4 Subject Comments

Subject comments were gathered in two ways. As the subjects were using the interface, the experimenter recorded their comments, what they were doing at the time, while noting what difficulty they seemed to be having or enjoying. At the conclusion of the session the subjects filled out a short form (included as Appendix B) which included an area for their general comments.

5.5 Results

This section describes the results of each phase of the study and discusses the results in terms of the previously stated goals. Results were drawn from the subjects' video taped verbal comments during the study, as well other spoken comments. The comment forms were helpful and prompted some other comments that may have otherwise been missed, although they rarely transcribed all their verbal comments onto the comment sheets.

5.5.1 Tag Results

A total of eleven subjects participated in the "tag" world. When there were multiple subjects present, one of them operated each end of the interface. When there was only one subject, I (or another experienced subject) ran the other end of the interface. The subjects were given a chance to play the game using either interface.

The tag game worked well for the immersive subject. The Prolix subject's movements were successfully communicated to the immersive environment, where the mannequin-like figure moved according to the key-presses. The immersive subject was easily able to tell who was "it" by noting the visual cues of sky color and who had the "it" hat.

The tag game was more difficult for the Prolix subject. The subject was able to tell who was "it" by the text messages when they were tagged and by looking for the "IT!" text appended to one of their names, but the graphical display was difficult to interpret. The graphical display became rapidly cluttered with graphical glitches and needed to be refreshed often. As a result, the inexperienced Prolix subjects had considerable difficulty locating and avoiding the immersive subject.

5.5.2 Chat Results

A total of thirteen subjects participated in the "chat" world. When it was possible for two subjects to attend at the same time, they talked to each other. When there was only one subject, the experimenter ran the other end of the interface.

The subjects were uniformly pleased at this aspect of Prolix. The graphical display behaved much better in this world, with each subject's movements being accurately and speedily drawn an the other subject's screen. The subjects who were experienced with text-based worlds typically spent a few moments looking at the objects, and then began to chat. Subjects who had never used text-chat systems before were quite hesitant about what was going on at first, but rapidly became fully engaged. Both subject types completely ignored the graphics display after some initial exploration. Perhaps since the subjects started within sight of each other, they felt as if they were in the same "place" already and felt no need to move about on the display. In future studies, it would be informative to start subjects in different "places" and observe if they moved within sight of each other. Most of the conversation was held through the use of `say` commands, with use of various other atmosphere commands (`emote`, `echo`) in rough relationship to their experience at using other text-based worlds.

5.5.3 Discussion

This section evaluates the usability, effectiveness and apparent uniqueness of Prolix.

Usability

The interface to the tag world was only partially successful. The text based notifications and information look-ups worked as planned. However, the graphical display presented to the subjects was overwhelmed with data from the immersive environment and became too cluttered with graphical "ghost" images, though it worked well with a single subject, when one subject remained stationary, or when the other subject was a Prolix subject as well. This "ghosting" may have been caused by Prolix's internal handling of each participant's view of the world. An incorrect partition or sorting of this data could account for the double images if one participant's data was stored in the other's data partition.

Observations of the subjects' use and their comments indicate that the Prolix interface to a chat world was extremely usable. Subjects had no difficulty remembering the commands and referred to the guide sheet and on-line help infrequently. The subjects with previous text based communication experienced felt especially familiar with the system and had few problems. One frequent mistake by subjects with no previous text experience was the omission of the `say` command, causing the rest of the typed text to trigger an error message.

Effectiveness

The tag world was effective and enjoyable for the immersive and the Prolix subject when the experimenter or another experienced subject operated the Prolix end of the interface. When an inexperienced subject operated the Prolix end, they always became confused by the graphical display problems and could not locate (and therefore evade) the immersive subject. The "ghosting" on the plan and side views probably made it difficult for subjects to construct a 3D model of the space.

The immersive subject's experience was considerably better than the Prolix experience, partly as a result of being immersed via the expensive hardware. But the immersive subjects consistently reported that playing tag with another subject was great fun. Several subjects reported that it was great to be chased by the other subject and have such a personalized reaction. Typically programmed "smart" VR entities are unable to mimic human reactions and behaviors. Even when the Prolix subject was unable to chase or avoid the immersive subject effectively, the immersive subjects still had fun, in one case taking great lengths to stay as far away as possible from the blindly moving Prolix subject.

The chat world was throughly successful. The subjects' verbal and written comments, as well as the conversational threads themselves, indicated that they were effectively and successfully communicating.

Uniqueness

Table 5.1 lists what the subjects thought was the most unique feature of Prolix. Several subjects put several of these comments. The most frequent remarks from experienced VR people was the inclusion of a text subject with an immersive subject. While subjects from outside the lab often commented more frequently about their enjoyment of the chat environment.

Subjects' remarks on unique features
• the inclusion of two subjects in a world through the keyboard
• having a text subject interacting with an immersive subject
• ability to chat with another person in real time

Table 5.1: Subjects' remarks on unique features

Comments

Subjects commented that the immersive game of tag was fun to play against another participant, even if the other participant was handicapped by display problems.

Subjects who had little computer experience often wrote comments that they enjoyed talking to other subjects via their terminals in real time. Those subjects who were familiar with text based communication systems commented that it was good to see this in a VR system.

Experienced VR subjects at the HITLab were most impressed by the inclusion of another participant controlled from the keyboard.

I overheard one group of subjects remarking to each other:

> "I understand what's going on when he's doing it, but I don't think I could operate it."
>
> "Yeah, I don't think any normal human could be expected to use that graphical display."

The situation they were observing was an experienced subject operating the Prolix interface against another tag subject. He was continually hitting the "refresh" function because the Prolix graphical display was often cluttered with extra characters, obscuring the location of the immersive subject.

Another subject very eloquently stated his thoughts: "That was great, but you have to do something about that refresh thing."

One subject, who had indicated little prior computer experience, asked if Prolix would work over a modem. She was informed that Prolix had been designed with this in mind.

Other subjects who were unfamiliar with VR technology wondered that there weren't more systems available to put multiple participants in the same world.

5.6 Summary of Results

The subjects played a game of "tag" with one inclusive subject and a Prolix subject. The immersive subjects had fun chasing and avoiding the Prolix subject, who was often handicapped by Prolix's graphical display problems. When asked, the subjects agreed that the VR worlds were more interesting with people controlling other objects in the world.

The inclusive subject was then put on a Prolix terminal with another Prolix subject at another terminal and placed in a VR world where conversation was encouraged. Experienced and novice subjects of text based communication systems both enjoyed the conversation atmosphere provided by Prolix.

From the observation of subjects and their comments, it seems that Prolix's graphical display is overwhelmed by data from an immersive environment. However, it was usable, effective and enjoyable with other Prolix-only environments. The text-based part of Prolix was successful in providing a usable, effective, and enjoyable way for people to communicate with each other.

Chapter 6

APPLICATIONS

6.1 Overview

There are a variety of possible uses for Prolix. Prolix was inspired by a combination of VR technology and MUDs so there are obvious applications as an alternative to expensive graphical interfaces as well as potential uses in entertainment and communications. Prolix may be able to find a place for itself in the field of Computer Supported Cooperative Work or in the current HITLab system as a debugging tool. As with other enabling technologies, Prolix may be put to unimagined uses.

6.2 Alternative Interface

Prolix was designed to provide an alternative means to experience VR. Prolix should be an accessible alternative to traditional expensive VR interfaces since it requiring nothing more than a keyboard for input, and the venerable vt100 emulation for simple ASCII only graphic and text output. While inclusive interfaces provide a much richer experience that currently encompasses the visual and auditory domains, this alternate interface could provide a low order access for those unable to access the high-end interface equipment.

Typical immersive interface equipment (ignoring the additional computational resources required to produce spatially localized sound sources and generate a 12-30 Hz update rate in a shifting point-of-view polygon intensive world) is expensive. First, and most expensive of all, are the eyephones. The VPL eyephones at the HITLab date from 1989 with a retail price of $8,000. The HITLab originally used a VPL dataglove, but have since adopted a $10 Toys-R-Us joystick as a discrete input device. Position tracking is accomplished by a $7,000 FastTrack magnetic sensor with a sensor located atop the eyephones and one in the joystick. Auditory feedback is provide through a pair of quality headphones, retailing for about $100. These immersive interface components total approximately $20,000. The additional computational components (e.g. high end graphics workstations and localized sound processor) costs an additional $100,000 to $200,000.

Compare this price to the ever plummeting cost of a PC with telecommunications software and a modem. Free public access to terminals with dial-up capabilities already exists. At a very minimum, Prolix provides a much cheaper, though impoverished, experience of VR.

6.3 Communications

Once a participant has connected to a virtual world, through any interface, the most interesting entities in that VR are likely to be other participants with whom one will undoubtedly want to converse with. Some fully inclusive VR system provide a microphone for each participant, providing a natural way for participants to communicate within the VR world.

If an immersive virtual reality system does not provide speech capability, such as a workstation-based graphics interface, communication with the world may be extremely limited or awkward. With the text based interface, communication is provided by typing messages which are propagated out to all other interested parties. Although not as easy as speaking, typing thoughts and words to another person in real-time has been proven to be an effective means of communicating. A whole culture of internet users have developed informal protocols and techniques for surmounting and exploiting conversation without the assistance of human body language. [2], [9], [13], [15], [18], [21], [29], [36], [39], [44]

6.4 Entertainment

Forthcoming VR worlds can be expected to exist solely for their entertainment value. The entertainment value comes from the intrinsic behaviors encouraged within the VR world or the social interplay among the participants.

The literature review contains several papers [2], [9], [10], [13], [29] that discuss the entertainment based world of MUDs, where players interact with each other or database entities created by other players. Often, the game aspect of these systems (killing monsters for experience points) is completely ignored in favor of conversing with other players and friends as a form of entertainment.

6.5 Computer Supported Cooperative Work

Computer Supported Cooperative Work (CSCW) systems are defined as providing support for a common goal or task, and providing an interface to a shared environment. VR is by its very nature a shared environment. The current goals and tasks commonly include exploration and visualization, with a start being made toward a truly interactive artwork. Prolix provides explicit support for common VR goals and tasks at the HITLab. Currently this includes navigational support, and graphics that provide iconic spatial location.

6.6 Debugging

Since Prolix is text based, there is the potential for online access to an editor, giving a participant the power to shape the world from within. This idea has already been implemented in various internet community games [13] [41], where a certain subset of users have the power to access the entity datafiles from within the environment. This provides world designers with the ability to rapidly iterate development. They can create an object, load it, test it, make modifications, and repeat until the entity behaves as desired.

6.7 Summary

VR possesses the potential to bring many geographically separated people together in a common space. Until the interface hardware becomes cheaper, it is hoped that Prolix will serve as an alternative interface, allowing people to gather in VR for entertainment, communication, CSCW, debugging or other forthcoming applications.

Chapter 7

FUTURE DIRECTIONS

7.1 Overview

This chapter outlines some future directions for Prolix expansion, including a section discussing the conversion of Prolix from a set of VEOS library calls to a set of Mercury 2.0 library calls and actual improvements or additions to the functionality of the interface, rather than a restructuring of the code.

7.2 Mercury 2.0

Currently, Prolix exists as a set of library routines linked into the VEOS executable at compile time. The interface is used by starting a controlling FERN entity that makes the proper calls to the library routines. As Prolix's primary use is as an alternative interface, the program must function as a fully operational participant system, dealing with renderers, sensors and data management.

At the time Prolix was started, the HITLab's participant system was in its early stages of development and designed to exclusively handle fully immersive VR systems. In Mercury's upcoming version 2.0, the software supports generalized sensor and renderer drivers.

A conversion to Mercury 2.0 protocols would vastly reduce the amount of code required for a text based system. The same functionality of Prolix could be reproduced by writing two drivers, a keyboard sensor to gather and process key hits and a vt100 renderer to show the output. These two drivers would then be added to the library of available devices for the Mercury user to choose from. Using the Mercury 2.0 system, Prolix would gain the robustness of a more mature system, including subtle data transfer issues, timing and flow control, multiple participant data partitions, as well as filtering and processing front-ends.

7.3 Graphical Display Ghosts

Prolix's graphical display suffered from "ghost" images of entities under certain situations. During the usability study, the subjects often had to refresh the screen to remove the ghost images. The ghosts appeared when many entities were moving, and were especially prevalent in the immersive environment. This cluttered the graphical display and obscured any real motion. Some subjects dealt with the problem by frequent screen refreshes, but a more permanent solution must be addressed in any future releases of Prolix.

7.4 Alternate Movement Models

The current release of Prolix supports a discrete movement model. A single cursor press changes one's position from the current grid space to the next. If the grid is set small enough, observers of the text body in the immersive VR world perceive this motion as continuous. If the grid is set at a larger setting, immersive users see the Prolix user "popping" from spot to spot. While this movement model has its appeal, some form of smoothing could be performed so the illusion of continuous motion is maintained.

The discrete movement model also means that one is only moving when actively pressing the keys. Each keypress directly adds the appropriate value to the 6D position. (Example: the left key subtracts one grid value from the current position.) An alternate model is a velocity model, as used by the HITLab's immersive users, where pressing the fly trigger on the input wand sends a preset velocity to the body. The body then takes care of updating the user's 6D by computing the distance traveled from the elapsed time and last known position. Since the body constantly updates its 6D as rapidly as possible when the trigger is held down, one gets the illusion of smooth flight. This movement model could be incorporated into Prolix to provide smooth movement at high grid spacings to any immersive users watching.

Yet another movement model that could be implemented is a "thrust-and-rotate" system. In this model, certain keys rotate the body, while another provides the thrust. In simple 2D spaces, the right and left cursor keys rotate you clockwise and counterclockwise while the up-cursor key provides the thrust. Moving a "thrust-and-rotation" system into 3D presents some cursor-key mapping problems. For example, how do you rotate up? Should you use a 5-button system and try to represent a pilot's joystick? If properly implemented, the "thrust-and-rotate" model would be a worthwhile addition to Prolix.

7.5 Bounding Box Graphics

Currently, Prolix represents any object as a uniformly sized character icon placed at the object's origin, despite the object's real size. As an example, the participant's icon is as big as the space needle's icon. A more accurate way to represent objects would be based on some measure of their shape. It could be as simple as representing the space needle as a stack of eight "S's", instead of a single "S" at the base. This would involve introducing some other set of required attributes to the FERN protocol or, more elegantly, an interpretation of the polygon-graphical description file.

7.6 LISP Interface

The HITLab's current world development is driven from a LISP front-end to take advantage of the interpreted nature of LISP in order to prototype worlds rapidly. This is done by temporarily pausing execution, issuing LISP commands, and resuming execution. Having this same flexibility to put the world on "pause" and issue LISP commands would be beneficial within Prolix as well.

Issuing LISP commands would be easiest if there was a command that removed the user from Prolix's control loop and left the user at the natural VEOS prompt. Users could then issue LISP commands and then resume their Prolix session, greatly simplifying the debugging process.

7.7 Editor Access

Allowing the user access to the datafiles that represent the entities allows anyone to create entities from within the interface itself. In MUDs, a small subset of players have access to a special set of UNIX-based commands to navigate the file system and manipulate the data there [15]. Prolix could adopt a similar protocol that would allow for the creation and modification of FERN entities and the controlling LISP code. Allowing access to an underlying file system presents inherent security risks unless appropriate precautions are taken to prevent access to unauthorized areas.

7.8 Aliases and Macros

As users become more experienced at operating the interface and performing tasks, they often desire to change a long series of repetitive actions into a macro that can be executed with a single command. This command can then be executed so the user doesn't have to type the whole series each time.

Aliases provide the user with a similar flexibility to rename a command to a shorter or more familiar name. For example, a DOS user may feel more comfortable renaming a cryptic UNIX command like `rm` to the more familiar `delete`.

The ability to customize the command structure is appreciated by the expert user, just as the logically grouped command hierarchy is appreciated as the user learns. With this in mind, Prolix could be expanded to include macros and aliases.

7.9 Prolix as a Client

Currently, the use of Prolix requires the participants to run the VEOS executable on an account with access to the VR world they wish to join. In the case of the HITLab, this limits the number of people who have access to the VR worlds to those who have accounts on the HITLab's computer subnet.

A more open solution would be to establish a constantly running world that would be able to accept participants through some established protocol. The HITLab has taken the first steps towards this goal by proposing a constantly running "antechamber" that serves as a portal/server to other HITLab worlds. However, this still presupposes an account at the HITLab.

Perhaps a better approach might be borrowed from MUDs, where the world accepts telnet connections over a specific port. This allows anyone with telnet access to participate in any MUD that advertises its internet address and port number. If Prolix were to be structured as a client that connected to a remote VR world through some protocol, many more people would have access to VR.

7.10 Summary

Any further releases of Prolix must address the graphical display ghost difficulties. Other future directions for the expansion of Prolix include fattening the functionality of the interface through the addition of alternate movement models, bounding box graphics, prototyping, debugging and world building commands. Although transparent to the users, a conversion to Mercury 2.0 would make the interface more robust. Finally, adopting some sort of client/server relationship between the participant system and the VR world would increase the number of VR users.

Chapter 8

CONCLUSION

Prolix was conceived and built to fulfill the need of bringing many people together in a common virtual space. Today's virtual reality holds great potential, especially in providing a venue for facilitating the interactions of people in a common space. Yet the technology to put many people in a VR space is currently expensive. This suggests that an available, cheap, but effective, alterative interface to VR would be helpful.

As an inexpensive alternative, Prolix provides access through one of the most common terminal emulation techniques that uses only a basic screen and keyboard. No exotic sensor or display devices are required.

As an effective mean of accessing VR, text based interfaces have succeeded before in the internet community (MUDs, IRC, gopher, archie) and will continue to be used. Prolix serves as an effective text based communication system for VR worlds. It can better serve as an effective interface to other immersive VR worlds if the graphical display problems are addressed.

The construction of Prolix as a fully functioning participant system was a difficult task, requiring scrupulous attention to data communication and storage issues, graphics, and terminal handling. Prolix achieved its design goals of placing a text-based participant into existing VR worlds and allowing Prolix participants to communicate with each other, all without the use of expensive interface equipment.

If Prolix is to be truly successful as an alternate interface and tool for social interaction within VR spaces, the interface must be made available to a spectrum of individuals and organizations.

Prolix and other alternate ways of empowering large numbers of people with the chance to experience VR are beginning to be developed. As this happens, the content of the worlds will take on much more importance. VR world designers will have to concentrate on providing interesting things for people to do *together* in their worlds, rather than simply constructing visually engaging worlds.

BIBLIOGRAPHY

[1] Yael Adini and Dov Sagi. Parallel processes within the “spot-light” of attention. *Spatial Vision*, 6(1):61–77, 1992.

[2] Amy Bruckman (asb@media lab.media.mit.edu). Identity workshop: Emergent social and psychological phenomena in text-based virtual reality. Technical report, MIT Media Laboratory, April 1992. Electronic Text available via anonymous ftp from beta.xerox.com:/pub/MOO/papers/identity-workshop.ps.Z.

[3] Dr. Richard Bartle. Interactive multi-user computer games. Technical report, British Telcom plc, December 1990. Electronic text available via anonymous ftp from beta.xerox.com:/pub/MOO/papers/mudreport.ps.Z.

[4] Micael Benedikt, editor. *CyberSpace: First Steps*. MIT Press., 1992.

[5] Benford and Fahlen. A spatial model of interaction in large virtual environments. In *Proceedings of 3rd European Conference on CSCW*. milan, September 1993. (at press).

[6] William Bricken and Geoffrey Coco. The veos project. Technical report, Human Interface Technology Lab, Seattle, WA, 1993.

[7] Claudia Brugman (brugman@crl.ucsd.edu). Spatial cognition: The prespective from theoretical semantics. *PSYCOLOQUY*, 3(45), Tues June 23 1992. Electronic full-text available via anonymous ftp from princeton.edu:/pub/harnad/psyc.92.3.45.space.6.brugman.

[8] David J. Bryant (bryant@northeastern.edu). A spatial representation system in humans. *PSCOLOQUY*, 3(16), Sat May 23 1992. Electronic full-text available via anonymous ftp from princeton.edu:/pub/harnad/psyc.92.3.16.space.n.bryant.

[9] Eva-Lise Carlstrom. Better living through language: The communicative implication of a text-only virtual environment, or, welcome to lambdmoo! Technical report, Grinnell College, May 1992. Electronic document available via anonymous ftp from beta.xerox.com:/pub/MOO/papers/communicative.txt.Z.

[10] Carl Brown (cbrown@netcom.com). Micromuse history. Electronic Text available via anonymous ftp from 18.43.0.102:muse/info/Muse.History, October 1992.

[11] Geoffrey Coco. fern.doc. Technical report, Human Interface Technology Lab - Seattle, Wa, 1992.

[12] Geoffrey Coco. The virtual environment operating system: Derivation, functian and form. Master's thesis, University of Washington, June 1993.

[13] Pavel Curtis. Mudding: Social phenomena in text-based virtual realities. Technical report, Xerox PARC, 1992. Electronic document available via anonymous ftp from beta.xerox.com:/pub/MOO/papers/DIAC92.ps.Z.

[14] Pavel Curtis and David A. Nichols. Muds grow up: Social virtual reality in the real world. Technical report, Xerox PARC, January 1993. Electronic document available via anonymous ftp from beta.xerox.com:/pub/MOO/papers/MudsGrowUp.ps.Z.

[15] Johan Andersson (d8andjo@dtek.chalmers.se). The design & implementation of lpmud: Implications for vr technology. Technical Report R-91-4, Human Interface Technology Lab - Seattle, Wa, 1991.

[16] Johan Andersson (d8andjo@dtek.chalmers.se). Mive - multi interfaced virtual environments. Technical Report R-92-7, Human Interface Technology Lab - Seattle, Wa, 1991.

[17] UNIX development team. Unix man pages, 1991.

[18] Elizabeth M. Reid (emr@munan.ee.mu.oz.au). Electropolis: Communication and community on internet relay chat. Technical report, University of Melbourne,

1991. Honor's thesis for Dept of History. Electronic document available via anonymous ftp from beta.xerox.com:/pub/MOO/papers/electropolis.ps.Z.

[19] Benford et al. From rooms to cyberspace: Models of interaction in large virtual computer spaces. *Interacting with Computers*, 1993. (a Butterworth-Heinmann journal).

[20] Fahlen et al. A space based model for user interaction in shared synthetic environments. In *InterCHI '93*, 1993.

[21] Randal F. Farmer and Chip Morningstar. The lessons of lucasfilm's habitat. In Michael Benedikt, editor, *Cyberspace: First Steps*. MIT Press, Cambridge, MA, 1992.

[22] Alan Garnham. Space: The final frontier? rely to haberlandt on garnham on reading-inference. *PSYCOLOQUY*, 30(4), April 1993. Electronic full-text available via anonymous ftp from princeton.edu:/pub/harnad/psyc.93.4.30.reading-inferenence.12.garnham.

[23] Arthur M. Glenberg and William E. Langston. Comprehension of illustrated text: Pictures help to build mental models. *Journal of Memory and Language*, 31:129–151, 1992.

[24] Arthur M. Glenberg, Marion Meyer, and Karen Lindem. Mental models contribute to foregrounding during text comprehension. *Journal of Memory and Language*, 26:69–83, 1987.

[25] Gerald M. Phillips (GMP@PSUVM.bitnet). Implicit philosophy. *PSYCOLOQUY*, 3(30), Tue June 2 1992. Electronic full-text available via anonymous ftp from princeton.edu:/pub/harnad/psyc.92.3.30.space.2.phillips.

[26] J. G. Hollands and Ian Spense. Judgement of change and proportion in graphical perception. *Human Factors: The Journal of the Human Factors Society*, 24(3):313–334, June 1992.

[27] Tony Mason John R. Levine and Doug Brown. *lex & yacc*. O'Reilly & Associates, Inc., 2nd edition, 1992.

[28] Erik A. Kay. Mire: A multi-user information retrieval environment. Technical report, Massachusets Institute of Technology, May 1992. Electronic full-text available via anonymous ftp from lysator.liu.se:/pub/lpmud/misc/thesis.ps.Z.

[29] Kevin Kelly and Howard Rheingold. The dragon ate my homework. *Wired*, 1(3):68–73, July 1993.

[30] Yili Liu and Christopher D. Wickens. Use of computer graphics and cluster analysis in aiding relational judgement. *Human Factors: The Journal of the Human Factors Society*, 34(2):165–178, April 1992.

[31] Dean Luick. *Nethack 3.1.1: vision.c*. Nethack Development Team, 1992. Electronic archival available via anonymous ftp from mrcnext.cso.uiuc.edu:/nethack.

[32] Andrew MacDonald and Max Minkoff. Mercury explained. Human Interface Technology Lab, 1993.

[33] Masinter and Ostom. Collaborative information retrieval: Gopher from moo. In *Proceedings of INET*, 1993.

[34] Max Minkoff. The participant system. Master's thesis, University of Washington, June 1993.

[35] Scott Moir. Rants & raves – muds. *Wired*, 1(4):17, September 1993.

[36] Michael S. Rosenberg (msr@casbah.acns.nwu.edu). Virtual reality: Reflections of life, dreams, and technology - an ethnography of a computer society. Electronic document available via anonymous ftp from beta.xerox.com:/pub/MOO/papers/ethnography.txt.Z, March 1992.

[37] Nancy Franklin (nfranklin@ccmail.sunysb.edu). Inquiring into the spatial representation system. *PSYCOLOQUY*, 40(3), Thur June 11 1992. Electronic full-text available via anonymous ftp from princeton.edu:/pub/harnad/psyc.92.3.40.space.4.franklin.

[38] C. C. Presson and B. R. Roepnack. Multiple mental models. *PSYCOLOQUY*, 3(65), Wed December 16 1992. Electronic full-text available via anonymous ftp from princeton.edu:/pub/harnad/psyc.92.3.65.space.12.presson.

[39] Jill Serpentelli. Conversational structure and personality correlates of electronic communication. Technical report, Haverford College, 1992. Electronic document available via anonymous ftp from beta.xerox.com:/pub/MOO/papers/conv-structure.txt.Z.

[40] Ben Shneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Addison-Wesley Publishing Company, 2nd edition, 1992.

[41] Jennifer Smith. Frequently asked questions 1/3: Muds and mudding. Electronic text distributed via USENET group rec.games.mud.announce, July 1993.

[42] Jennifer Smith. Frequently asked questions 2/3: Mud clients and servers. Electronic text distributed via USENET group rec.games.mud.announce, July 1993.

[43] staff. Inside & out. *the Stranger*, 2(45):2, 1993.

[44] Steve Steinberg, editor. *InterTek*, volume 3(3). Steve Steinberg Press, Winter 1992.

[45] Mark Takacs (tak@hitl.washington.edu). Muds as groupware. Class paper, December 1991.

Appendix A

PROLIX USER GUIDE

This appendix serves as program documentationa and a user's guide. It is divided into several sections, each one addressing a specific aspect of Prolix. The first explains the general concept behind Prolix, while the second explains the choice of interface style. The screen layout is explained, followed by a section on the basic use of Prolix. Section A.5.1 contains an alphabetical list of the commands, as well as a topical sorting of the commands. Section A.6 lists and explains all the commands.

A.1 General Concept

Prolix is intended to be a participant system that allows entry into virtual worlds traditionally only accessible with fully inclusive participant systems. The users of Prolix interact with the virtual world through the keyboard and cursor keys, and receive display information through the screen, using only ASCII/vt100 capabilities. It is hoped that a wider range of people will be able to experience some semblance of VR if the interface avoids the higher end graphics systems such as X-Windows, or the complications of various specialized home computer platforms.

A.2 Interface Style

A goal of the HITLab is to create a virtual community with its own personality and flavor. Yet most of the worlds created, while interesting, are sterile and museum-like. None of these worlds yet created have the style and feel of a community such as a comfortable local cafe, bar, or even the sense of electronic community found on many MUDs or IRC [2], [18], [39], [44]. What's missing are the *people*. Because Prolix's hardware requirements are so commonly available, I hope that its widespread use will encourage many people to enter the current VR worlds. Having many people present and communicating within the same virtual space should allow the space to acquire an identity associated with the people who frequent that space.

A.3 Screen Layout

The typical Prolix screen is split into two major sections, the graphical display and the text display. Each of which is further divided in two: the graphics display has a plan and side view, while the text display has a separate input and output window. (See figure A.1)

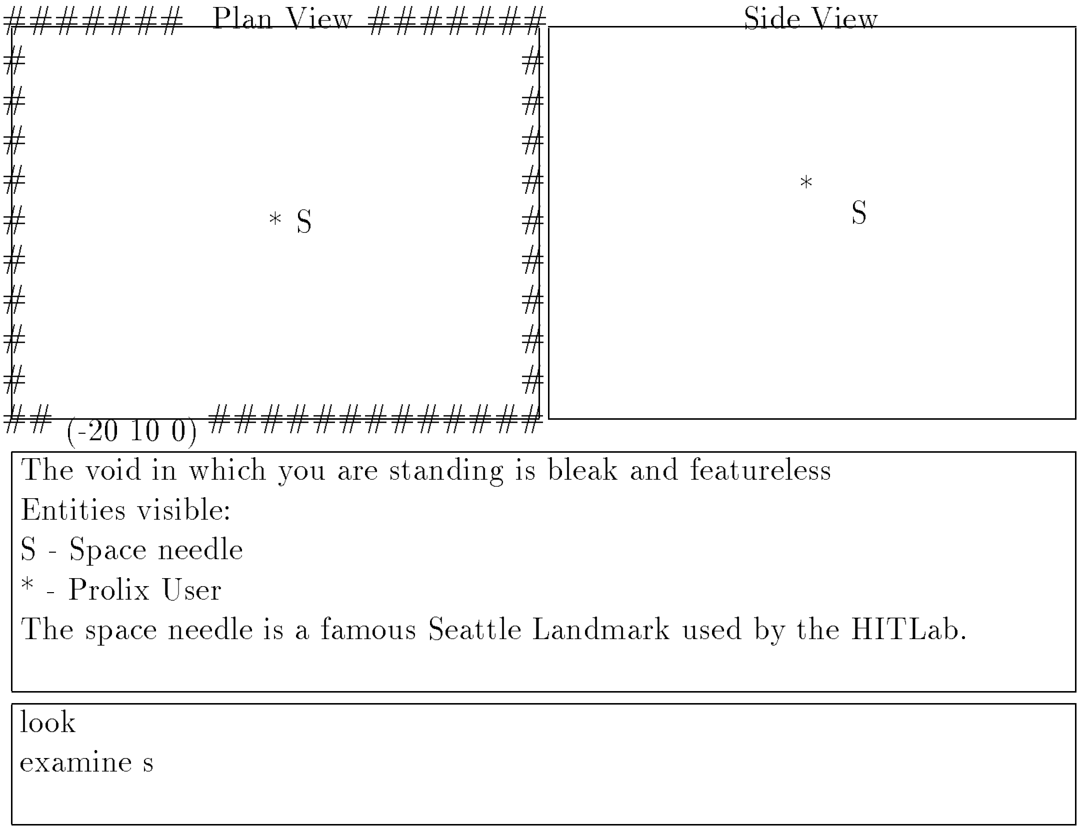


Figure A.1: The text display

A.3.1 Text Display

In its initial configuration, the text display takes up the bottom half the screen, with three to five lines (adjustable at compile time) at the bottom of the screen allocated to echo the user's typing input. (See figure A.1) The text output appears in the output windows and scrolls upward until it reaches the top of the output window.

The default output window size ends at the beginning of the graphics window. It is possible to eliminate the graphics window entirely, in which case the output window expands to encompass the entire window. In this mode, the up cursor key recalls the last typed line of input.

The results of user issued commands, text output from events in the VR world, and behaviors and communications from other participants and entities appear in the output window. To avoid cluttering the display, Prolix doesn't echo the user's commands to the output window, just their results or a feedback message.

A.3.2 Graphic Display

In its initial configuration the graphics window takes up the top half of the screen. (See figure A.2) The left half of the display shows the plan view, while the right side displays an orthographic side view. In the default configuration, each discrete character-wide space on the display grid represents a 1x1x1 meter volume. A single character icon is placed on the screen according to the coordinates of the object in the VR world. The plan view window shows the entities' `x` and `z` coordinates, while the side view window shows the missing `y` coordinate as well as the `x` coordinate. The HITLab's VR worlds follow a protocol where the center of the world is defined te be (0,0,0), with `x` as left and right, `y` as up and down, and `z` as depth into the scene. Mentally combining the information provided by the two views allows you to place the object in the world, as well as its elevation or altitude. If an object moves, its icon moves in both windows simultaneously.

In figure A.2 the "*" represents the user, who is located slightly to the left of and above the "S", which is the base of the space needle at the center of the VR world coordinates(0,0,0). The user and the space needle are separated by at least 20 meters horizontally in this example.

In the default configuration, your icon is free to move about the display until it reaches a configurable radius from the center of the display, at which point the display is re-centered about the user. The other mode fixes your icon in the center of the display and any movement commands cause the world to go scrolling by.

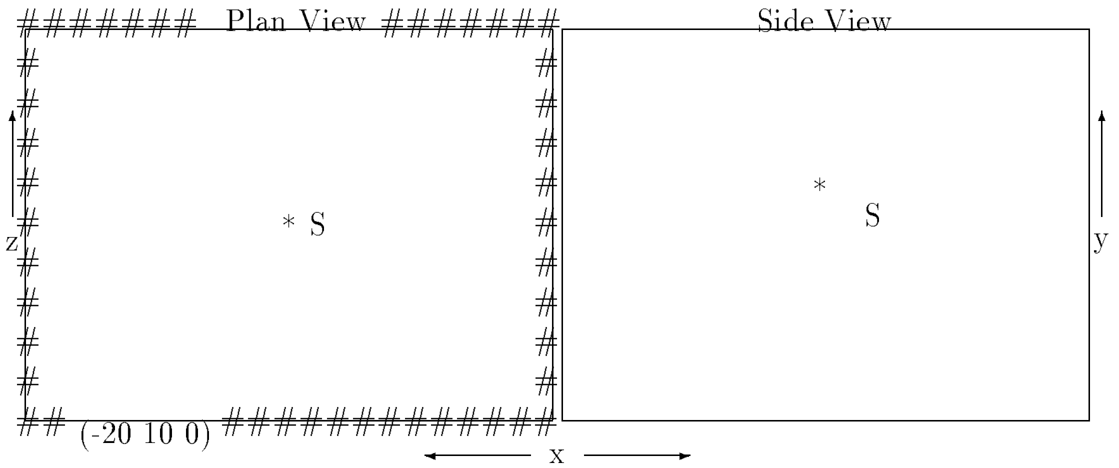


Figure A.2: The graphics display

A.4 Operation Guidelines

This section gives a brief description of some of the more common commands and their intended uses.

A.4.1 Navigation

Navigation through the VR world with Prolix is accomplished by the use of the cursor keys. Prolix starts with the plan view graphical display as the active window. The active graphical display window is outlined with "#" marks, rather than dashes.

For the sake of clarity, I will assume you are in a world where the plan view represents looking down on a space where the top of the screen is north, bottom is south, and left and right are west and east. The side view has the top of the screen as up, the bottom of the screen as down, and the left and right sides as west and east. See figure A.3.

When you hit the cursor-right key, your icon moves one space east in each of the windows. If you use the up-cursor or down-cursor while the plan view is active you will move north or south, respectively. However, if the side view is active, pressing the up-cursor or down-cursor moves your icon up or down in elevation. In figure A.3 pressing the up-cursor will move you north. When your icon comes near the edge of the screen, you are repositioned in the center of the display and the screen is redrawn. The percentage of the screen on which you can move around is a user preference and

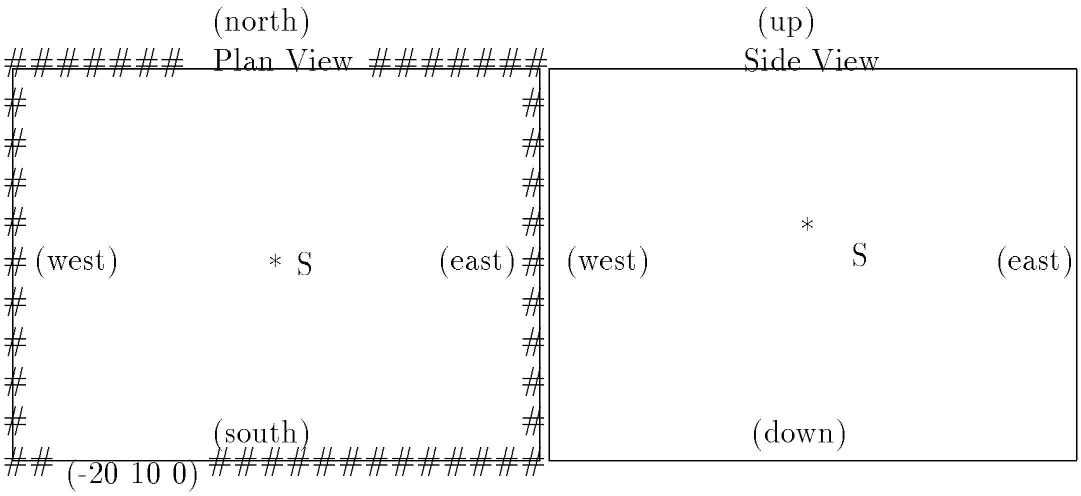


Figure A.3: Movement orientation

is set with a command called `sweet` and defaults to 100%, which determines how close your icon can get to the edge of the screen before being repositioned. Setting this value at 0% causes the user's position to be fixed at the center of the screen, where subsequent use of the cursor keys causes the screen to scroll past in that direction.

A teleportation command called `goto` is provided to allow you to instantly jump to some known location. Providing the name or icon of another entity as arguments to this command moves you directly to that entity. If no argument is provided, you are moved to the absolute world coordinates (0,0,0).

A.4.2 Communication

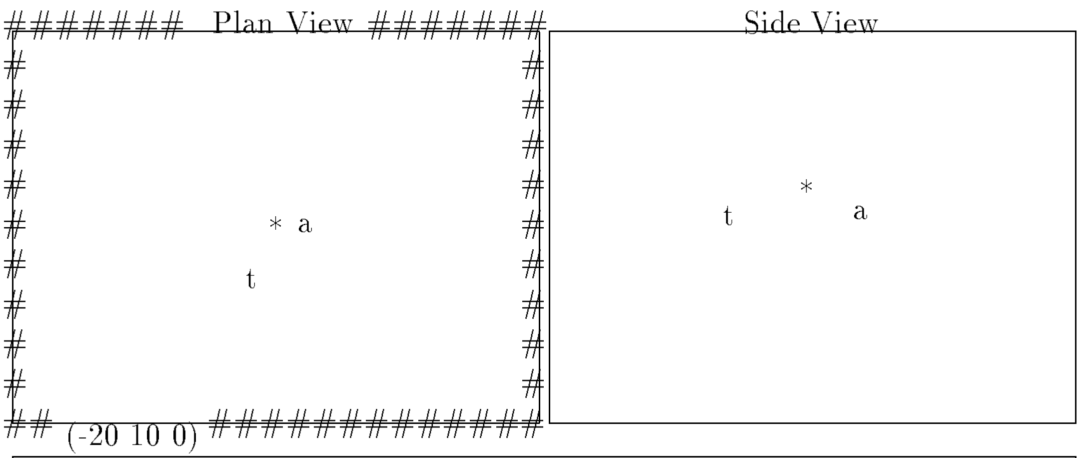
Communication in the VR world with Prolix is accomplished mainly through the `say` and `tell` commands. The `say` command causes anything you type to be passed out into the VR world for other entities to see and/or process. Other entities receive information on who sent the message and the message's content. It is up to the particular entity to decide how to process this information. Any Prolix interface will automatically display incoming text messages.

An example session will serve to illustrate the essential points. In figure A.4, the user has issued the `who` command, (line (a) in figure A.4) which gives a listing of all the participant systems in the VR world (the line marked with (1)).

Then the user decides to ask a question using the `say` command (b) so that it is distributed to everyone. Appropriate feedback (2) is given in the output window.

The user known as Teklon replies with a behavior (3) known as an `emote`. The `emote` command prepends whatever text you type with your name. In this example, Teklon has typed `emote shrugs` to produce the output marked with a (3). In this manner, your character can appear to do many things that lend atmosphere and style to the conversation.

Reading Ari's reply (4), the user decides to reply directly to Ari with the `tell` command (b)(5), which is only received by the person to whom it is addressed (Ari). Other users (like Teklon) remain unaware that anything is even happening. Technically, everyone receives all the messages, but each individual Prolix entity performs filtering functions (see chapter 4 for more details). Ari replies in kind (6).



(1) 3 Users online: Ari (a), Teklon (t), Tak (*)
(2) You say: Anyone know when the meteor shower starts?
(3) Teklon shrugs.
(4) Ari says: I saw a posting about it.
(5) You tell Ari: I think it starts at midnight..
(6) Ari tells you: I think you're right.

(a) who
(b) say Anyone know when the meteor shower starts?
(c) tell ari I think it starts at midnight.

Figure A.4: Communication in Prolix

There are several other ways that Prolix allows you to shape your communication, including `shouts`, `echos`, and `aliases`. These are discussed in section A.6.

A.4.3 Database Manipulation

Database manipulation is the term for altering the database that you keep about other entities and the information you disseminate to them.

Each VR entity has a collection of attribute-value pairs that can be modified by that entity. Figure A.5 shows a typical entity. Prolix provides a way to change the traditional values (visibility, name, etc), as well as specific fields recommended by the Prolix protocol.

```
('name' 'space needle')
('visible' t)
('wireframe' nil)
('teleportable' t)
('picture-desc' 'needle')
('alias' 'needle')
('icon' 'S')
('short' 'The Space Needle')
('long'
 'The Space needle is a Seattle icon and a defacto VR
  object as well.  It rises many meters overhead and
  is built from hopefully sturdy supports. Two elevators
  provide access to the observation deck at the top.')
```

Figure A.5: Example entity

The `icon` command allows you to change the icon that is displayed on the graphics display of other Prolix users. For the benefit of any inclusive users that may be in the world, Prolix provides the `pict` command which takes as an argument the filename of your new graphical description file.

The `name`, `title` and `short` commands allow you to change the information shown to other Prolix users when they ask for information about you. The `name` command allows you to change your name.

Normally when other users `look` or issue a `who`, they are provided with the short description of each entity. The short description is composed from two subfields, the name and title field. For example, the user "tak" with the title "- the lowly peasant" have a short description of "tak - the lowly peasant" which would be displayed as a result of a `who`, `look`, or `examine`. The `short` command enables complete control of your short description. Using the short description, you could appear as "a rock." when people `look` in the room or view the `who` list.

The current version of Prolix allows you to alter anything in your database, including your name, icon and other identifying data. As with any great flexibility, there is usually a compromise somewhere. This great flexibility assumes the users trust other users not to maliciously masquerade as someone else. If prolonged usage of Prolix shows this to be a problem area, the command set can be restricted so that key identification data is unable to be modified.

A.4.4 Interface Configuration

Prolix offers its users the option to customize the interface and get status information about the current state of the program.

As mentioned in section A.4.1, Prolix has several variables governing how movement in the graphical display works, including a command to control the amount of VR world space each screen grid space represents. This command is called `grid`, and takes a numerical argument that changes the space represented by each grid square. The grid size defaults to one.

Another variable that the user can set is the `sweet` percentage. This is the percentage of the screen (measured from the center) that the user can move about on without causing a screen refresh. The default sweet percentage is 100%, enabling the user to advance to the edge of the screen.

Other commands of interest are the `help` command, which prints a brief (but complete) listing of all commands. The `prolix` command shows a list of various interface command variables such as the `sweet` percentage, the grid size, the active window, your screen and world coordinates, and other miscellaneous information about the displays.

There are several control sequences that are useful during operation. The most useful one is perhaps the refresh (`<control-l>`), which discards all optimization information and draws the screen over again from scratch. There are a number of other control sequences, but the only other one that needs to be mentioned here is the `quit` command, which is triggered by typing `quit` or by the `<control-c>` sequence.

A.5 Command Reference

This section presents several brief lists of the commands grouped in different ways. Section A.5.1 is an alphabetical list of all possible typed and control sequence commands. Section A.5.2 is a functional grouping of the commands into the four categories of communication, movement, database manipulation and interface control.

A.5.1 Alphabetical List of Commands

Figure A.5.1 alphabetically lists thhe complete set of commands that Prolix accepts, including control-key sequences.

Alphabetical List of Commands		
Control-b	emote	quit
Control-c	examine	say
Control-d	goto	shout
Control-l	grid	short
Control-o	help	sweet
Control-u	home	tell
Control-z	icon	title
Cursor Keys	long	visibility
clear	look	who
describe	prolix	wireframe
echo	pict	

Figure A.6: Alphabetical list of commands

A.5.2 Commands by Function

Figure A.5.2 contains a functionally grouped list of all the command and control-sequences that Prolix accepts.

Commands by Function			
Communication:	say	tell	shout
	emote	echo	
Movement:	<cursor keys>	goto	home
Database:	who	look	examine
	icon	name|short	pict
	long|describe	title	
	visibility	wireframe	
Interface:	help	brief	sweet
	grid	quit	prolix
	ctrl-l	ctrl-c	ctrl-d
	ctl-o	ctrl-u	ctrl-z
	ctrl-b		

Figure A.7: Commands by Function

A.6 Description of Commands

This section gives complete specifications of all the Prolix commands. The listings include command syntax, description, an example, and a list of related commands. The commands are logically grouped into subsections for communication, movement, database manipulation and interface control.

A.6.1 Communication

say

Syntax: `say` <*text*>

Description: The `say` command is used to send a message to everyone in the immediate area. The message is automatically prepended with your name. This is the most common form of communication.

Example: you type: `say The metro demo is fun. Let's try it.`
you see: You say: The metro demo is fun. Let's try it.
others see: Tak says: The metro demo is fun. Let's try it.

Abbreviation: '

See Also: tell, shout

tell

Syntax: `tell` <*name*> <*text*>

Description: The `tell` command is used to send a person-to-person message which is automatically prefaced with your name. The message is only seen by the person to whom it is addressed. The recipient will recieve the message even if they are in another area.

Example: you type: `tell ari Meet me at the metro demo.`
you see: You tell Ari: Meet me at the metro demo.
ari sees: Tak tells you: Meet me at the metro demo.
others see: (others don't see anything)

See Also: say, shout

shout

Syntax: `shout` <*text*>

Description: The `shout` command is used to send a message to everyone logged in at the time. The text message you provide is automatically prepended with your name.

Example: you type: `shout Tell me if you want a metro tour.`
you see: You shout: Tell me if you want a metro tour.
others see: Tak shouts: Tell me if you want a metro tour.

See Also: say, tell

echo

Syntax: `echo` <*text*>

Description: The `echo` command causes the text provided by the user to be shown to all other users in your area.

Example: you type: `echo The Metro train pulls into the station.`
you see: The Metro train pulls into the station.
others see: The Metro train pulls into the station.

Abbreviations: ;

See Also: emote

emote

Syntax: `emote` <*text*>

Description: The `emote` command is used to provide atmosphere by sending some text that describes what you're doing. Your name is automatically prepended.

Example: you type: `emote smiles.`
you see: You smiles.
others see: Tak smiles.

Abbreviations: :

See Also: echo

A.6.2 Movement

This section describes the commands that allow you to navigate through the VR world.

Cursor Keys

Syntax: <*cursor keys*>

Description: The cursor keys provide the primary way to move yourself through the virtual world. Each key hit moves you a discreet distance (as set by the `grid` command) in that particular direction.

See Also: goto, home

goto

Syntax: `goto` <*x,y,z coordinates*>

Description: The `goto` command instantly changes your location to the specified coordinates.

Example: you type: `goto 10 20 0`
you see: You move to position (x=10 y=20 z=0)
others see: Tak teleports to a new position.

See Also: home, cursor keys

home

Syntax: `home`

Description: The `home` command instantly changes your position to a predetermined set of cooriditates, typically x=0, y=0, z=0.

Example: `home`

See Also: goto, cursor keys

A.6.3 Database Commands

This section contains a description of all the commands that allow you to alter the data stored about you, as well as query for data about the VR world.

look

Syntax: `look` {`at` <*icon*> | <*text*> }

Description: This command has two uses. The first, with no arguement, gives you a list of all the entities in the space with you. This includes other participants as well as inanimate VR world objects. The information provided consists of the entity's icon and short description, which in the case of other participants, will usually consist of their name and title. The second use, given as `look at` is the same as using the `examine` command.

Example: `look at tak`

See Also: examine

examine

Syntax: `examine` <*icon*> | <*text*>

Description: This command displays the "long" description of the entity you request. The argument can be the representative icon, or the entity's name.

Example: `examine tak`

Abbreviations: exa

See Also: look

who

Syntax: `who`

Description: This command shows a listing of all the other participant systems in the current VR world with you, including immersive participants as well as other Prolix users. The information provided is a single line consisting of the character's icon and short description, which usually consists of a name and title.

Example: `who`

See Also: look, examine

icon

Syntax: `icon` <*character*>

Description: This command changes the graphical icon that represents your position in the VR world on other Prolix graphical displays.

Example: `icon T`

See Also: name, title, short, pict

short

Syntax: `short` <*text*>

Description: The `short` command is included primarily for total database control. Text entities have three basic things that identify them: an icon, a short description, and a long description. As a default, the short description is the user's name, followed by their choosen title. If this command is used, the given short description overrides the default combination.

Example: `short This is tak's new short description.`

See Also: icon, name, title, long

title

Syntax: `title` <*text*>

Description: Your title, combined with your name, produces the information shown to people when they `look` around the room or issue a `who` command. Using this command allows you to change your title information.

Example: `title - the great and powerful wizard of Oz`

See Also: name, short

name

Syntax: `name` <*text*>

Description: Use of the `name` command changes the identifier used as your name. Issueing this command automatically changes your icon to the first letter of your name.

Example: `name ted`

See Also: icon, short, title, long.

long

Syntax: `long` <*text*>

Description: This command lets you alter the detailed description that others see when they `look` or `examine` you.

Example: `long Tak is a very tall human with big feet.`

See Also: describe

describe

Syntax: `describe` <*text*>

Description: Use of the `describe` command has the same effect as issuing the command `long`. The two different names exist for the same function to reflect the different ways the command can be pictured. `long` is the database field where the info will be stored, while `describe` is more descriptive of the function.

Example: `describe Tak is a very tall human.`

See Also: long

pict

Syntax: `pict` <*filename*>

Description: This command exists for the benefit of any inclusive users who may be with you. This command changes the polygon picture file associated with your body to the new file that you specify.

Example: `pict cube`

See Also: wireframe, icon

visibility

Syntax: `visibility` {`t` | `nil`}

Description: This command toggles your entity's VEOS and Prolix visibility status. Issued as `visibility nil`, this command removes your icon from the graphical display and indicates that your picture should not be shown to other immersive users. `visibility t` restores your icon and picture.

Example: `visibility nil`

See Also: wireframe

wireframe

Syntax: `wireframe` { `t` | `nil`}

Description: This command exists for the benefit of any inclusive users who may be with you. Issued as `wireframe t`, it changes your current graphical representation from polygon shaded to a wireframe model. `wireframe nil` has the opposite effect. This has absolutely no effect on other Prolix viewers.

Abbreviations: wire

Example: `wireframe t`

See Also: visibility

A.6.4 Interface Commands

The commands listed in this section help maintain and costomize the Prelix interface. These commands do not effect the VR world at all, with the exception of the quit command, which causes your entity to leave the VR world.

help

Syntax: `help`

Description: This command prints a list of commands ordered by functionality, much like the list in section A.5.2.

Abbreviations: ?

Example: `help`

quit

Syntax: `quit`

Description: This causes Prolix to close down and exit immediately, as well as exiting any spaces you may be in.

Example: `quit`

See Also: control-c (quit), control-z (suspend)

grid

Syntax: `grid` {<*number*>}

Description: Specifies grid length for each space on the graphical display. If issued without an argument, it reports the current grid value.

Example: `grid 10`newline Sets each space in the graphical display to a 10x10x10 volume.

See Also: sweet, movement - cursor keys

sweet

Syntax: `sweet` {<*percentage*>}

Description: The sweet percentage control how far from the center of the display you can move before a refresh is triggered and you are placed back in the center of the screen. The default value is 100% which triggers a refresh when you reach the edge of the display. A value of 50% would allow you to move halfway to the edge of the screen before recentering and redrawing the display. A value of 1% will mean that you do not move from the center of the screen. As you press the cursor keys, the world scrolls by you. Issuing the command without any value reports the current percentage rate, as well as how many grid spaces this allows you to move with the current screen size.

Example: `sweet 100`

See Also: control-l (refresh), grid

Control-b

Syntax: `<control-b>`

Description: This command toggles the output window from its normal half screen size to a full screen size and back. Note that if the graphics display has been disabled with control-d, you will be unable to regain the half-screen sized window, as that would leave half the screen unused.

Example: `<control-b>`

See Also: control-d (display toggle)

Control-c

Use of the control-c sequence has the same effect as issuing the command `quit`.

Control-d

Syntax: `<control-d>`

Description: This control sequence toggles the graphics display on and off. Note that in the default configuration, the text output window extends only to the edge of the graphics display. If you turn off the graphics display, Prolix expands the text output area to fill the whole screen, just as if the user had issued a control-b. Pressing control-d again will re-enable the graphics display. Note that the text output will remain in "big" mode, with text scrolling under the graphical display.

Example: `<control-d>`

See Also: control-b (BIG toggle)

Control-l

Syntax: `<control-l>`

Description: Throw away the `curses` screen optimization information and redraw the screen from scratch. The graphics display is recentered as well.

Example: `<control-l>`

See Also: clear, sweet

Control-u

Syntax: `<control-u>`

Description: Clears the current text input line.

Example: `<control-u>`

Control-z

Syntax: `<control-z>`

Description: Causes the execution of the program to be suspended. When proceesing is returned to the foreground via the UNIX `fg` command, you will be in FERN mode and able to issue LISP or FERN calls. To resume the Prolix interface, you will need to type `(fern-go)`.

Example: <`control-z`>

clear

Syntax: `clear`

Description: Clears the screen by printing a line of dashes "-" followed by numerous newlines. Use this comand when the refresh command fails to properly remove screen clutter.

Example: `clear`

See Also: control-l (refresh)

history

Syntax: <`cursor-up`> (only in text only mode)

Description: If the graphics display has been turned off with <control-d>, the <up-cursor> operates to recall the last typed command.

Example: <`cursor-up`>

See Also: control-d (display, display toggle)

prolix

Syntax: `prolix`

Description: This command prints a list of information about the interface itself. Including currunt screen size, sweet percentage, grid size, active screen, 6D world coordinates, screen origin in the world and output window size.

Example: `prolix`

See Also: grid, sweet

Appendix B

SUBJECT SURVEY FORM

This appendix contains the survey form the subjects were given to fill out at the completion of the informal pilot study.

```
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

       Prolix Text-based Interface to Virtual Reality
       Subject Survey

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Please feel free to fill in only the parts that you are comfortable with,
although I am especially interested in your comments.

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Name:

Have you used multiple person text based interfaces (MUDs, IRC, UNIX talk,
etc) before?

Have you used multiple person graphics based interfaces (xTrek, Rogue,
Moria, NetWar) before?

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

For the following questions, I am interested on your thoughts and opinions,
as well as observations.  These comments will be used to assess areas of
strength and weakness in the interface.  The questions are there as
guidelines for discussion.  Please feel free to elaborate on them or related
points the questions may raise.

-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
      * General Use *
```

```
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Overall reaction to the system:
            terrible         1 2 3 4 5 6 7 8 9 N/A   wonderful
            frustrating      1 2 3 4 5 6 7 8 9 N/A   satisfying
            dull             1 2 3 4 5 6 7 8 9 N/A   stimulating
            difficult        1 2 3 4 5 6 7 8 9 N/A   easy
            inadequate power 1 2 3 4 5 6 7 8 9 N/A   adequate power
            rigid            1 2 3 4 5 6 7 8 9 N/A   flexible

Did you feel you understood what was going on in the VR world:
            confusing        1 2 3 4 5 6 7 8 9 N/A   clear

Learning to operate the system:
            difficult       1 2 3 4 5 6 7 8 9 N/A   easy

Exploration of features by trial and error:
            discouraging    1 2 3 4 5 6 7 8 9 N/A   encouraging

Can tasks be performed in a straight-formard manner:
            never           1 2 3 4 5 6 7 8 9 N/A   always

Supplemental reference materials
            confusing       1 2 3 4 5 6 7 8 9 N/A   clear

What did you spend most of your time doing?

Any other comments?





-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
   * Graphical display *
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Did the graphical display give you a feeling of where you were located?
            confusing        1 2 3 4 5 6 7 8 9 N/A   clear

Did you feel you understood what was going on in the graphical display:
            confusing        1 2 3 4 5 6 7 8 9 N/A   clear
```

```
Was the display crowded, or cluttered?
          cluttered       1 2 3 4 5 6 7 8 9 N/A   clear

Was it useful to have a graphical display?
          useless         1 2 3 4 5 6 7 8 9 N/A   useful

Any other comments?


-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
      * Text display *
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Did the text output give you a good idea of what was going on in the VR
world?

Was the text that appeared on the screen:
            confusing      1 2 3 4 5 6 7 8 9 N/A   clear
         cluttered       1 2 3 4 5 6 7 8 9 N/A   clear
            too slow       1 2 3 4 5 6 7 8 9 N/A   just right
            too fast       1 2 3 4 5 6 7 8 9 N/A   just right

Did you understand the gist of your conversation with others:
            clueless       1 2 3 4 5 6 7 8 9 N/A   like being there

Remembering names and use of commands:
            difficult      1 2 3 4 5 6 7 8 9 N/A   easy

Help messages on the screen:
            confusing      1 2 3 4 5 6 7 8 9 N/A   clear

If you conversed with the other text-users, was it effective? How did it
compare to other text conversation systems you may have used?

Was it to useful speak to others?
          useless         1 2 3 4 5 6 7 8 9 N/A   useful

Any other comments?



-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
```

```
 * Movement & navigation *
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Were you able to move about the world easily?
            difficult       1 2 3 4 5 6 7 8 9 N/A   easy

Was it useful to move about?
           useless          1 2 3 4 5 6 7 8 9 N/A   useful

Did you rely more on the graphical or text display (or neither) to confirm
your movements?

Any other comments?



-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-
* Comments *
-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-

Please feel free to provide any additional comments.  The questions below
are intended to serve as guidelines.  Please feel free to write other
thoughts down that will assist us in identifying weak/strong areas.


Negative:
Was something really annoying?  Confusing?  Inaccurate or just plain wrong?


Positive:
Was there something you liked alot?  Any unique features?  Potential uses?



Please feel free to write down any additional thoughts you may have:
```

Appendix C

SOURCE CODE AVAILABILITY

As the source code for the Prolix library additions to VEOS contains more than 7000 lines of code (courtesy of `wc -l *.c *.h *.l *.y`), it would have been impractical to include it here. Tho source code will be available at the HITLab's forthcoming anonymous ftp site (probably `ftp.hitl.washington.edu`) or directly through me.

I would welcome any comments or information requests. I can be reached at the address below until at least December 1993.

Mark Takacs
University of Washington
c/o Human Interface Technology Lab FJ-15
Seattle, WA 98195
tak@hitl.washington.edu
mtakacs@fry.halcyon.com