THE UNIVERSITY OF CALGARY

Design and Implementation of Global Virtual Laboratory - a Network-Accessible Simulation Environment

by

Pavol Federl

A THESIS
SUBMITTED TO THE FACULTY OF GRADUATE STUDIES
IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE
DEGREE OF MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE
CALGARY, ALBERTA
DECEMBER, 1997

THE UNIVERSITY OF CALGARY
FACULTY OF GRADUATE STUDIES

The undersigned certify that they have read, and recommend to the Faculty of Graduate Studies for acceptance, a thesis entitled “Design and Implementation of Global Virtual Laboratory - a Network-Accessible Simulation Environment” submitted by Pavol Federl in partial fulfillment of the requirements for the degree of Master of Science.

_____

Supervisor, Dr. Przemyslaw Prusinkiewicz

_____

Dr. Robert Kremer

_____

Dr. Brian Wyvill

_____

Dr. Claude Laflamme

_____

Date

Abstract

Many activities in computer graphics can be regarded as experiments on virtual objects or models. In the process of experimentation the existing models are gradually improved and new model categories emerge. The Virtual Laboratory (VLAB) is a software environment designed to support model development by facilitating the manipulation of models and providing mechanisms for retrieving and storing large numbers (e.g., thousands) of them. This thesis describes a number of VLAB extensions I designed and implemented as part of my master's research. As a result of these extension, the models in VLAB can be shared between many users who may work at different geographical locations. Alternate views of databases can be maintained, allowing users to access objects in different orders. A visual parameter editor was implemented, providing intuitive mechanism for external control of parameters used in experimentations through user configurable graphical user interfaces. The overall performance and portability of VLAB was improved, and various customization mechanisms for adjusting visual appearances of VLAB applications made available.

Acknowledgments

I would like to thank my supervisor, Dr. Przemyslaw Prusinkiewicz, for the advice he offered to me during my research, for the countless hours he spent with me on discussing various aspects of VLAB, and for helping me to edit and proofread this thesis many times. I could never have finished this research without him. Learning to conduct research from a world-class researcher was a pleasure. My next thank-you goes to my mother and my brother, for supporting my academic career in every possible way. Special thanks go to my girlfriend, Jennifer Walker, for proofreading this thesis. Her moral and emotional support, as well as her incredible patience while I was writing this thesis, made the whole process enjoyable. Finally, I would also like to thank all users of VLAB for their valuable comments and suggestions, most importantly to Przemyslaw Prusinkiewicz, Jim Hanan and Radomir Mech.

Table of contents

List of tables

List of figures

Chapter 1:Introduction

Chapter 2:Statement of objectives

Chapter 3:Survey of related concepts and previous work

Chapter 4:Remote access server

Chapter 5:Panel manager

Chapter 6:Browser

Chapter 7:Metatext

Chapter 8:Hyperbrowser

Chapter 9:Conclusion and future work

Appendix A:RA class

Appendix B:More on panel manager

CHAPTER 1 *Introduction*

Computers in the current technological era are becoming more powerful, easier to use, and more affordable every day. They are now in widespread use in industry, education, research and homes. In scientific research, computers are used by scientists for various purposes such as organization of work, electronic collaboration, development of algorithms, and simulation of experiments. Various software applications exist, offering adequate functionality for each of these tasks. However, a unified environment that would support all these existing applications in a consistent way is needed. Virtual Laboratory (VLAB) is a software environment designed to address this problem.

VLAB is based on an electronic analogy of a laboratory, offering scientists a coherent platform for performing and organizing computer assisted experiments [13][18][20][21]. The idea behind VLAB emerged from the need for an environment for organizing and simplifying work related to computer based simulation of biological phenomena. The large number of files representing models of plants require structured organization. Numerous parameters used in plant modeling demand a tool which provides a fast and an intuitive mechanism for their modification. Frequent operations, such as rerunning plant generating programs or invoking editors on different files, require a tool which performs such actions quickly and conveniently. VLAB's design aims to address these issues - to be an interactive environment for creating, managing and conducting experiments.

The authorship of the concept of VLAB belongs to Dr. Przemyslaw Prusinkiewicz and his graduate students Lynn Mercer and Jim Hanan [20]. In 1990, the first version of VLAB (version 1.0) was designed and implemented by Lynn Mercer as part of her Master's Programme research [21]. VLAB 1.0 provided only very limited visual information about the experiment databases and a very simple user interface, which was a direct consequence of being implemented under a limited window management system (without support for widgets, dialogs or other GUI components now readily available). In 1995, Earle Lowe created a new version of VLAB (version 2.0) by redesigning its communication mechanism and introducing fundamental changes to its database browsing capabilities [18]. Unfortunately, much of the new functionality in VLAB 2.0 was implemented as a prototype, resulting in low performance and reduced reliability in some of its components. These shortcomings interfered with testing and verifying the new concepts introduced and implemented in VLAB 2.0. Some of the deficiencies related to the new browsing capabilities of VLAB 2.0 were addressed in my reimplementation of browser - a VLAB application used for navigating and managing databases of

experiments. Browser's reimplementation was the topic of my project in an undergraduate computer science course [7]. I selected VLAB as the topic of MSc research to improve its design and to add to its functionality.

VLAB has been successfully used to support a wide variety of activities in computer science, such as simulations of biological phenomena, experiments in fractal geometry, rendering and animation of complex scenes, presenting tutorials for classes, preparation of papers for publication and maintenance of source code.

In my MSc research I made many improvements to VLAB's design and added new functionality to its applications. These will be the topic of this thesis. The rest of this chapter is organized as follows. In Section 1.1 I discuss the motivation behind VLAB. It should help the reader to understand the concepts which are implemented in the old version of VLAB, and also provide a background for the requirements of the new version of VLAB. In Section 1.2 I introduce VLAB 2.0 and outline its shortcomings, needed later in the thesis as the basis for describing my own contributions.

1.1. Motivation

Scientists often use computers on a regular basis to perform tasks analogous to experiments in their fields. Whether this involves running existing experiments, creating new ones or deriving experiments from existing ones, writing papers, designing presentations or producing animations, these tasks share many common characteristics.

Maintenance of relationships between programs and data files

All of the above tasks need a group of one or more data files and the related programs to be run on these files. For example, physically based simulations usually require large amounts of data (stored in data files) and the simulation programs. Modeling of graphical scenes is another example, where data files describe the geometry of the models, which are then used as input to a modeler or a ray-tracer. With a large number and variety of experiments it becomes difficult to keep track of which programs to run on which data files, in what order, and with which options. Therefore, relationships between data files and programs need to be effectively maintained, for example, as a set of actions that can be performed on an experiment.

User definable actions

As scientists experiment with new ideas or simply modify and improve the old experiments, the related groups of data files are repeatedly worked on. This usually involves rerunning parts, or whole experiments numerous times. A mechanism for convenient invocation of actions within an experiment is needed to simplify the work of users, and thus enhance their efficiency.

The need for such a mechanism also emerges from the fact that live presentations of computer based experiments require numerous reruns of simulations. If typing is used to invoke programs, the presenters must concern themselves with command memorization and possible typographical errors. These can lead to delays, increase the presenter's stress factor and be disruptive for the audience.

Parameter editor

For many types of experiments it is very important to thoroughly examine the parameter space of a problem. When the parameters for the simulation are stored within files, the search for optimal parameters usually requires editing the appropriate files in a text editor, and then rerunning the simulation program on the modified data. A tool is needed which will allow the users to examine the parameter spaces in a more intuitive manner - using graphical user interfaces. Using visual controls for parameter exploration will eliminate much of the tedious and non-productive work spent by typing in a text-editor, giving the user the possibility to examine the vast parameter combinations faster and in greater detail. Such a tool can also be utilized for presentation and teaching purposes, since changing parameters visually is more intuitive for audiences than using a text-editor.

Version management and organization of experiments

Scientists often need to work on different versions of experiments in parallel, while still requiring easy access to the original versions. Old experiments are constantly being retrieved, new versions created and documented, bad ideas discarded and appropriate files deleted. Such activities inherently yield a large number of files which need to be effectively maintained. Also, as the number of experiments in a scientist's database grows, it becomes increasingly important to be able to find the appropriate experiments quickly. Therefore, a tool is needed to hierarchically organize experiments, while giving the user an intuitive interface to the database's management. Such a tool has to provide the user with a visual representation of the database, where each experiment can be associated with its own depiction. This tool will also provide functions for modifying the database, such as addition, deletion, moving, copying, searching, etc.

Collaboration

Scientists often collaborate on the same set of experiments, which requires all data related to these experiments to be transferred several times between different computers. This is especially true if the group of scientists working on the same problem do not share the same geographical location. A mechanism providing the scientists with transparent access to remote databases of experiments is required. This mechanism should offer security to prevent unauthorized access.

Alternative views of object databases

An object in a database of computer experiments may be of interest in several contexts. For instance, a mass-spring simulation of a cloth could be developed as a part of a comparative study of various techniques for simulating physical behaviors of natural objects, but it may also serve as an example of an animation technique, illustration of data triangulation or simply as the source of a picture for a publication. A mechanism for creating alternative views of object databases, reflecting conceptual associations between the objects is needed.

Support for interaction

One of the main design goals of VLAB is to encourage and support user interaction. Giving the users freedom to experiment without fear of destroying the originals is a good approach toward achieving this goal. Once the user decides to experiment with an object, it should be copied to a temporary location, where modifications can be made to this temporary copy without affecting the original. After the experimentation is over, the user must have the option of saving the changes into the database, either by replacing or adding a new version to the original.

1.2. Overview of Virtual Laboratory 2.0 and its limitations

VLAB consists of experimental units called ‘objects’ organized in an object oriented filesystem, and various utility programs (tools) that operate on these objects.

1.2.1. VLAB objects and object oriented filesystem

The organization of VLAB's object databases is based on the *prototype-extension* model described by Lieberman [17]. The prototype-extension model is an object oriented approach for storing knowledge about objects. In this model, a new object (extension) is created by describing how it differs from an existing objects (prototype), inheriting all other knowledge from the prototype. Multiple inheritance is also supported.

Each VLAB object represents an experiment by encompassing all of its related information in files. These files can be divided into two groups: files with data, and a file containing information about actions which can be performed on the data files. Every VLAB object is assigned a separate directory, where the object's data files are stored. VLAB objects are associated with textual names, as well as graphical icons (for easy visual inspection while browsing the database).

VLAB objects are organized into hierarchical databases, also referred to as object oriented file systems (or oofs). In these databases, any VLAB object can have extensions (children), stored as subdirectories in the directory of the prototype (parent object). This mechanism for storing object extensions as subdirectories is well supported by the UNIX hierarchical filesystem.

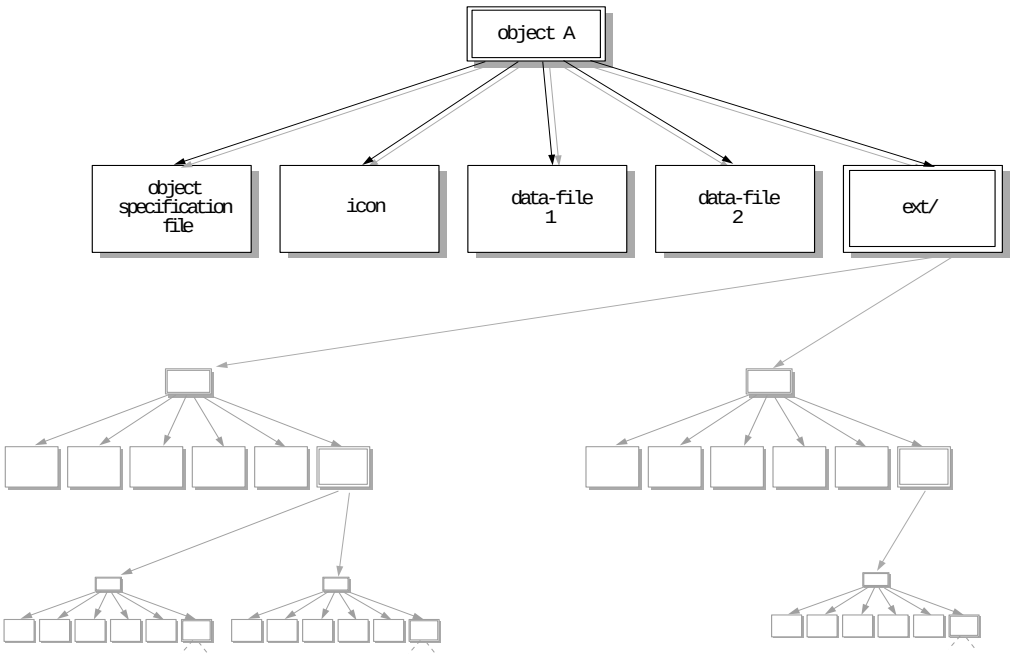


FIGURE 1-1: *Example of VLAB object's directory organization*

Figure 1-1 demonstrates an example of an object with two data files. Every VLAB object consists of any number† of data files, and three special-purpose files:

specifications (a required file for all VLAB objects) This file contains two types of information: list of data files included in the object and a description of actions which can be performed on these data files. This list of actions also defines the layout of the popup menus from which the user can invoke these actions.

icon (an optional file) This file stores an image in the SGI's RGB file format. The icon is user defined, and is used for visual representation of an experiment in the global view of the database. Icons are usually chosen by taking snapshots of simulation results. If an object does not have an icon, a default icon defined for the whole database is used.

ext (an optional directory) In the `ext` directory the object stores its extensions. Since each object is stored as a directory, the `ext` directory will only contain sub-directory entries.

VLAB's object oriented filesystem supports single inheritance on objects, which means a user can derive new objects from old ones, inheriting selected data/actions and defining new ones. Changes in the parent object are reflected in all of its children objects which inherit the data files to which these changes were applied. Single inheritance in VLAB is achieved by the following mechanism: whenever an object B is inserted into a database as a child of object A, all files of object B are compared against A's files, and only the ones that differ are stored in B's directory. Files that match are stored as symbolic links relative to A's files. Therefore, if an object was to be created as an extension of a prototype without making any changes to this extension, all files in the new extension's directory would be relative symbolic links to the files in the prototype's directory.

To demonstrate the inheritance mechanism in VLAB, consider the example in Figure 1-2. Object `model_1` has 6 entries in its directory. The first two are the `specifications` and `icon` files. The last one is the directory for extensions. The remaining entries, `data1`, `data2` and `data3`, are user defined data files needed for the experiment. Object `model_1.1` is an extension of `model_1`. It has 7 entries in its directory: it inherits `icon`, `data2` and `data3` from its prototype (`model_1`), redefines `specifications` and `data1`, and also adds an extra file `data4`. `Model_1.1.1` contains modified `specifications` and `icon`, inherited `data1` and `data2` from `model_1.1` and an extra file `data5`. Files `data3` and `data4` are not part of `model_1.1.1`.

In this scenario, if a user made a change to file `data3` in `model_1`, such a change would propagate to `model_1.1`, but `model_1.1.1` would be unaffected. If the change was made to `data2` in `model_1`, such a change would be reflected in both `model_1.1` and

† only limited by the UNIX operating system

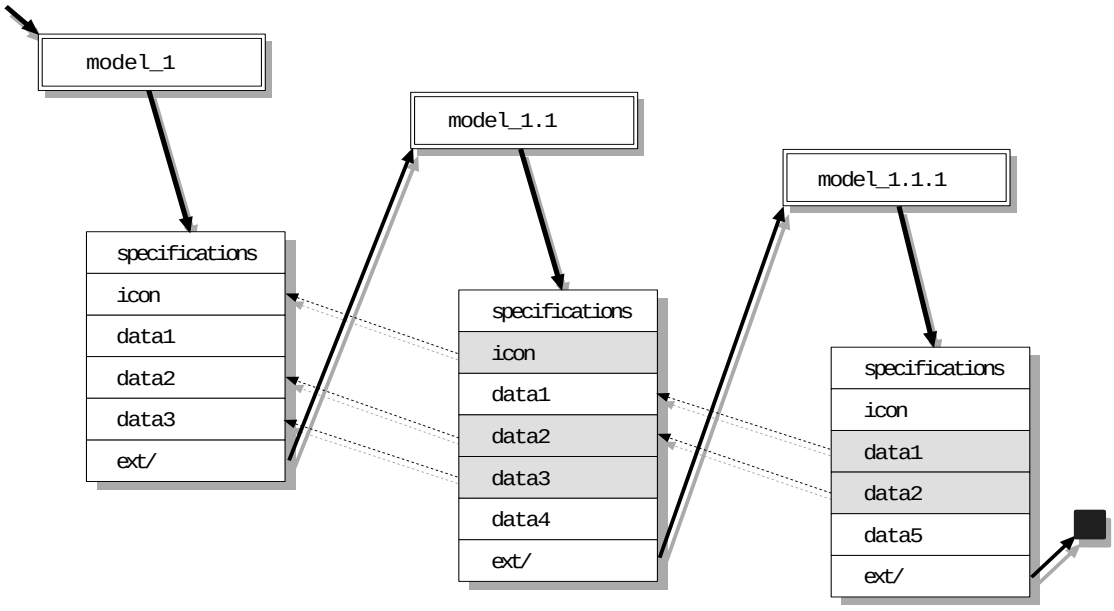


FIGURE 1-2: *Structure of VLAB's objects*

`model_1.1.1` objects. Naturally, any modification to `data1` in `model_1` would not affect `model_1.1`, nor `model_1.1.1`. Any change applied to `data1` of `model_1.1` would propagate to `model_1.1.1` while changes made to `data3` or `data4` would not.

It should be noted that VLAB objects do not inherit extensions.

1.2.2. VLAB 2.0 components

VLAB offers various tools to help access the databases of VLAB objects, and assist the users in experimenting with these objects. The most important tools are: browser, object manager, VLAB daemon and panel manager. The remainder of this section describes each of these components. Their shortcomings are analyzed where appropriate.

Browser

Browser is a VLAB component that provides the user with a visual interface for navigating through the hierarchy of objects. It can be invoked on any VLAB object database, displaying its hierarchy as a two dimensional tree graph (Figure 1-3). The user can expand specific parts of the hierarchy tree by showing or hiding children, show or hide

icons for individual objects or entire sub-trees, and textually search for experiments. Browser also makes it simple to modify the object hierarchy by supporting operations for cutting, copying, pasting, renaming, dragging & dropping of objects. Browser can be also used to invoke object manager (described in the text section) on selected objects.

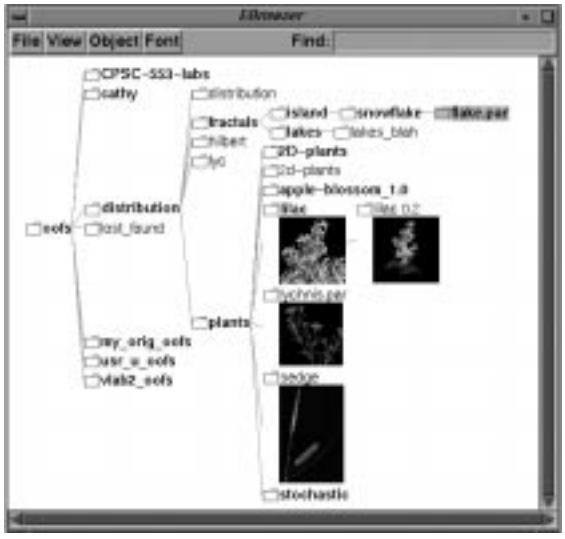


FIGURE 1-3: *Snapshot of the VLAB 2.0 browser*

The main limitations and shortcomings† of VLAB 2.0 browser can be summarized into three categories: lack of support of collaboration, low performance and user customization. VLAB 2.0 browser's lack of support for collaboration can be attributed to the fact that the entire VLAB up to and including version 2.0 was designed and implemented as a single user system. Browser's most important limitation is that it cannot be used to access databases stored on remote computers, making the collaboration between scientists at different geographic locations difficult.

Being a prototype implementation, having large parts implemented in Tcl/Tk and relying on external programs to supplement its internal functionality, browser 2.0 does not provide the speed needed for browsing in practice. For example, the user has to wait minutes to recursively display icons in a reasonably sized tree.

As VLAB became a popular environment for experimentation by many researchers, the drawbacks of its inability to customize its appearance were identified. The default settings (colors, fonts, tree layout, icon sizes, etc.) were not appropriate for some activities, such as

† I use the word *limitation* when describing missing concepts or design flaws, and the word *shortcoming* when describing flaws in the implementation.

interactive demonstrations, where the font and icon sizes must be increased in order for the audiences to be able to recognize them. Also, different users have different preferences.

Object manager

Object manager (Figure 1-4) is a tool used to manipulate the internals of a specific object. When object manager is invoked on a VLAB object, all files constituting this object are copied to a temporary space, called the “lab table” [21], where the user can safely modify them without worrying about destroying the original. The set of possible actions on an object is read from its `specifications` file, displayed on the screen as a pull-down menu. The user can then perform actions on an object by choosing items from this menu, without any detailed knowledge of the programs and data files involved in these actions. The changes made to an object while it is on the lab table can be saved back to the database, added to the database as a new extension, or ignored altogether.

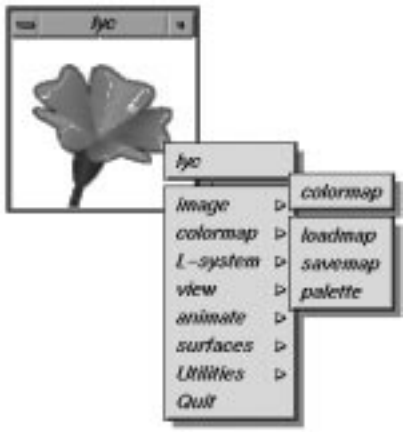


FIGURE 1-4: *Snapshot of VLAB 2.0’s object manager*

VLAB daemon

The Daemon is a VLAB component which provides a communication mechanism for the rest of the VLAB components (Figure 1-5). Almost all components of VLAB need to maintain some sort of inter-client communication in order to preserve consistency of displayed information. For example, changes to the object database made by one tool need to be reflected by all other tools displaying the same information. VLAB Daemon is transparent to the user, as it is invoked automatically by the first VLAB tool that needs it, and is shut down by the last one to quit.

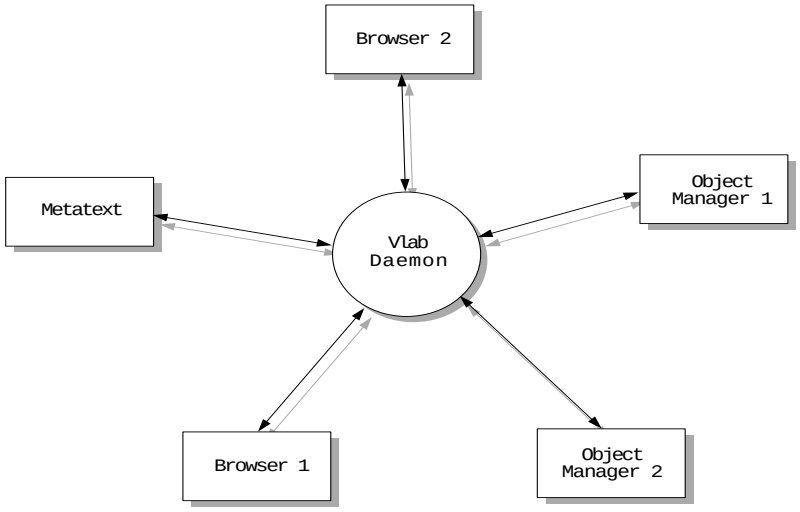


FIGURE 1-5: *Inter-client communication in VLAB 2.0*

Panel manager

Panel manager (Figure 1-6) is a VLAB program used to control simulation parameters during experiments. The parameters which panel manager can control are usually stored in text data files, and their locations are specified either by line and column numbers, or by a prefix string after which the parameter follows. Panel manager therefore acts as an editor of text data files. The creator of an experiment decides which parameters should be editable, and constructs a file with the description of a graphical user interface for each of the editable parameters. This file is used by object manager to display the set of controls on the screen, presenting the user with a visual interface for parameter modification. In VLAB 2.0 there are two GUI components available: buttons and sliders. Each of the GUI components is associated with a message which will be sent by object manager to the standard output as soon as the component is activated. This message is then translated by a special-purpose filter program into an appropriate editing action on the specified data file.

Panel manager in VLAB 2.0 has the following shortcomings: the lack of a tool for building graphical interfaces and one-way communication. The descriptions of panel's user interfaces for panel manager are written in textual form. Designing a graphical interface for a panel therefore involves editing the textual description and then re-running object manager on the new description to see the result. Often, this process has to be repeated until a satisfactory result is obtained. This process is rather inconvenient, particularly when specifying the layout.

FIGURE 1-6: *Snapshot of VLAB 2.0's panel manager*

Panel manager 2.0 only supports one way communication - sending messages to the filter program which performs the editing operations on the actual data files (Figure 1-7). There is no mechanism to find out the current values of the parameters from the data file, therefore the values in a newly displayed panel often disagree with the actual values in the data files. There is only one way to avoid this asynchronism in VLAB 2.0 - the most recent parameter values have to be manually encoded into the panel's definition file as default values for the associated GUI components.

Metatext

Metatext is a tool which makes it possible to access VLAB objects in arbitrary order, independent of the hierarchical organization of the database [29]. Metatext is invoked on an index file, which contains a list of entries converted by metatext into a pull-down menu (Figure 1-8). When the user selects one of these entries from the pull-down menu, metatext will load and interpret a frame file whose file-name corresponds to the selected choice. For example, if the user chose the item *Phyllotaxis*, metatext would load and interpret a frame file `Phyllotaxis`.

The frame files consist of a mixture of textual information, UNIX commands and messages to the VLAB Daemon. Before interpreting, metatext separates these three types

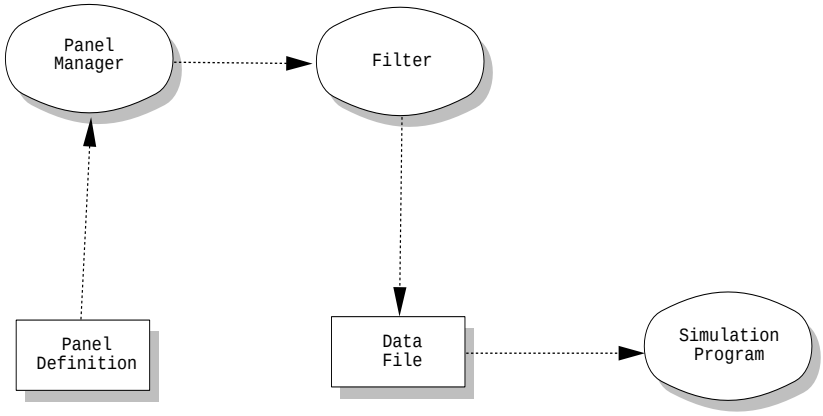


FIGURE 1-7: *Communication in VLAB 2.0's panel manager*

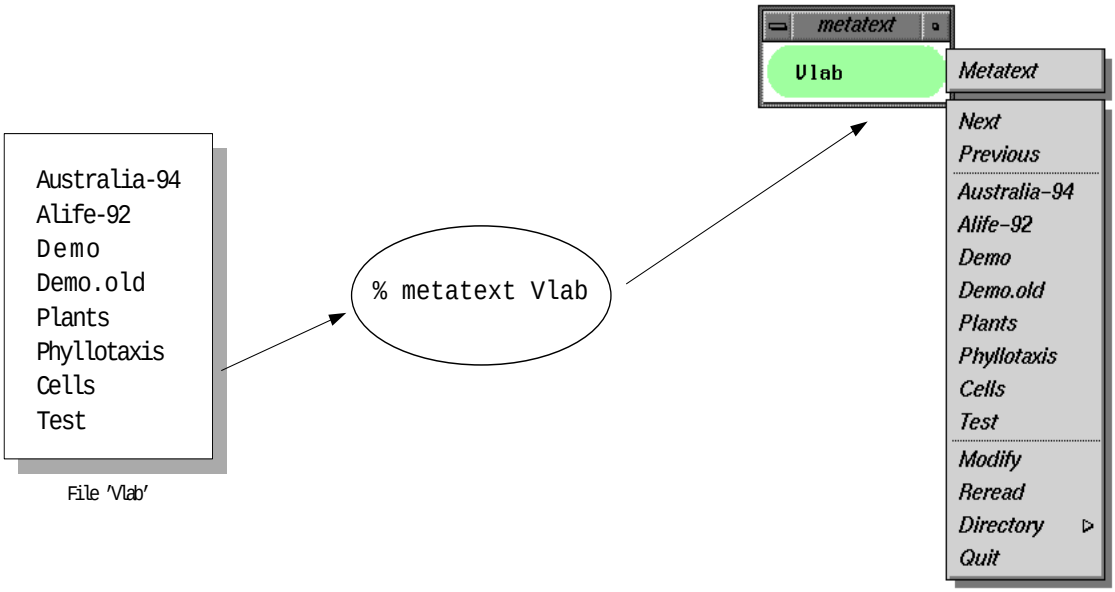


FIGURE 1-8: *Metatext in VLAB 2.0*

of information. Then the text from the frame is displayed in a separate window, UNIX commands executed in the order they are found in the frame file, and messages sent to the VLAB Daemon. UNIX commands are commonly used to spawn other copies of metatext (on different index files), and the messages to the VLAB Daemon are used to invoke object manager.

Maintaining a large number of metatext index and frame files presents a similar problem to the one of maintaining a large number of experiments. Organization of metatext nodes, such as moving, adding, renaming or modifying require the use of a UNIX shell and a text editor - not a very convenient interface for the user. The most important limitation of metatext, however, is keeping the links in metatext frames to the VLAB objects up to date while the object database keeps changing. Because there is no support offered by metatext to this end, it is the user's responsibility to manually update the affected frames whenever the referenced objects are modified (e.g. moved, renamed or deleted).

1.2.3. Portability

In order to prove the usefulness of VLAB as a general tool for performing experimentation, it is desirable VLAB be tested by users under many different environments. In order to make VLAB accessible to wider audiences, it is essential that it is portable to many other operating systems (not all researchers have access to SGI workstations, and most biologists have primarily PC's). Implementation of VLAB 2.0 uses many features specific to SGI workstations, making it unportable to other operating systems. Since the entire VLAB is very closely tied to the UNIX operating system, it would be a major undertaking to port it to operating systems like OS/2 or Macintosh or Microsoft's Windows environment. To start addressing the issue of portability, VLAB should be portable at least to other flavors of UNIX.

1.3. Summary

In this chapter I have introduced the Virtual Laboratory 2.0 - a software environment for computer based experimentation. Various components of VLAB 2.0 have been discussed and their limitations pointed out. These limitations are further discussed in the following chapter, where I present the statement of objectives.

CHAPTER 2 *Statement of objectives*

The objective of my research is to extend VLAB 2.0 by designing and implementing more functionality. I approached this by addressing the limitations and shortcomings of VLAB 2.0, as identified in Section 1.2. This chapter defines the specific goals of my research and briefly outlines how I accomplished these goals.

2.1. Alternative views of object databases

VLAB users often wish to use their objects in several different contexts. Metatext partially addressed this need in VLAB 2.0, by providing limited support for creating alternative views of object databases. Metatext only provides support for traversal of nodes containing hyperlinks to objects. Metatext nodes have to be created, updated and organized manually, as Metatext does not offer any mechanism to this end. These limitations greatly limit Metatext's usefulness as a tool for creating alternative views of object databases. A new mechanism, providing all functionality available through metatext, with support for an automatic update of hyperlinks and a user friendly interface for management of hyperlink databases was required.

I have designed a new system for hyperlink organization - a hierarchical hyperlink database. I have also developed a new VLAB application, called hyperbrowser, which allows users to navigate and effectively maintain such databases. The automatic update of hyperlinks has been addressed in my implementation by introducing an extra level of indirection between hyperlinks and objects, using an object lookup table. Hyperlinks point to objects through this table, which is automatically updated whenever the object database is modified. Any modifications to the database of real objects are therefore automatically reflected in the database of hyperlinks.

2.2. Support for collaboration

In order to improve VLAB's support for collaboration, a mechanism for accessing remote databases and controlling permissions for modifications to collaborator's databases was needed.

To allow users to access remote databases (located on remote filesystems), a new daemon (remote access server) has been designed and implemented. Remote access server runs on the remote computer and performs actions on the remote database as requested by VLAB tools accessing the database. Browser, hyperbrowser and object manager have been modified to take advantage of this new daemon. From the user's perspective, access to remote databases is transparent, since there is almost no difference between working with objects located in local and remote databases. The objects in remote databases can be accessed, moved between databases or modified just as they are in local databases. To make it easy for an application developer to request services from remote access server, a remote access library has been written. Both remote access server and remote access library can be easily extended to offer more functionality when needed. Similar to HTTP daemons used today for WWW access, remote access server has the potential to be used by scientists to offer information to the general Internet community.

To address the issue of controlling outside access to object databases, RAserver maintains a list of accounts for each user with access to the database. Each account is protected by an encrypted password, and is assigned a level of access. Two levels of access are supported in the current implementation: read-only and full access. The owner of the remote database can decide which person gets which type of access. With read-only access, RAserver will not permit any modification to the database for the connected user. With full access (only given to the most trusted colleagues) the remote user can modify the database in any way. A facility for maintaining an account list of remote users is part of RAserver.

2.3. Panel manager

The following two goals were to be accomplished with respect to panel manager. First, a tool was needed for visual design of panels, which would simplify the process of creating and modifying panels for parameter modifications. Secondly, two-way communication had to be implemented for automatic synchronization of displayed information and the contents of the parameters at start-up.

After a detailed evaluation of the existing panel manager and the required extensions, I decided to redesign and re-implement this tool. The new panel manager allows the user to

build and modify panels through a visual interface. Eventhough the implemented set of controls in panel manager 3.0 is small, extensibility was a major issue in its design. When needed, future developers can easily extended the set of controls. A sample screenshot of the new panel manager's user interface builder is presented in Figure 2-1.

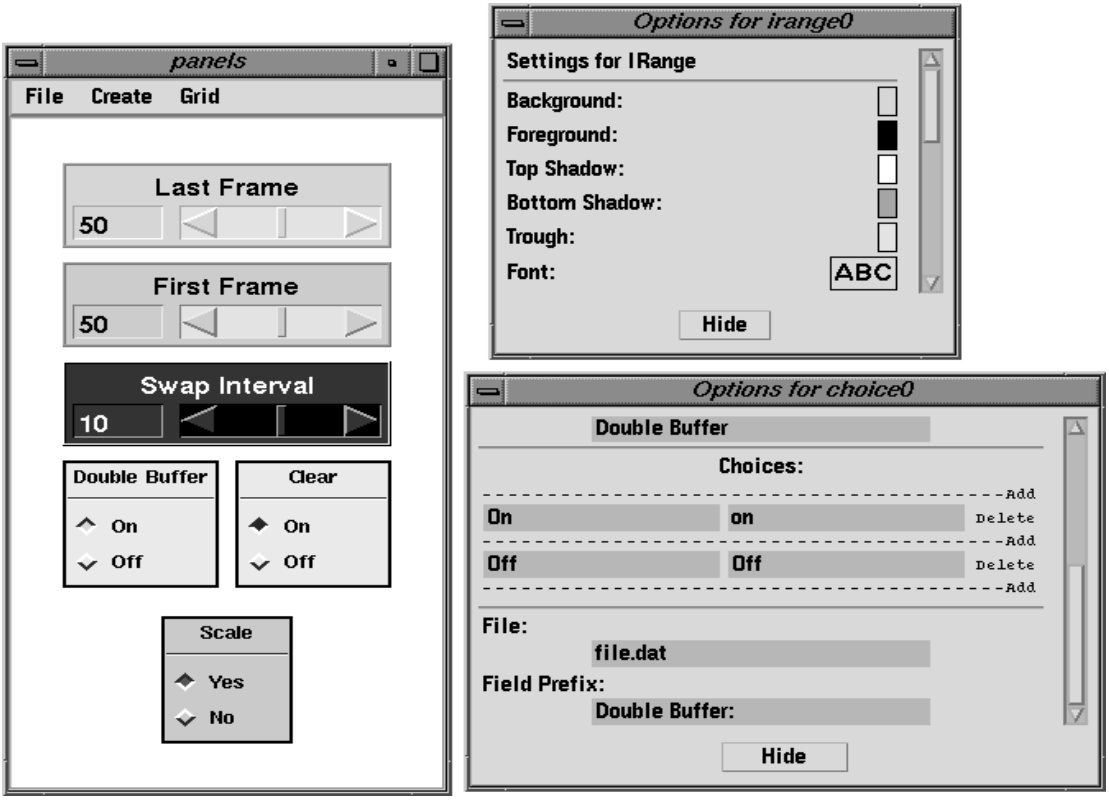


FIGURE 2-1: *Snapshot of panel manager 3.0 in graphical build mode*

The design of panel manager present in VLAB 2.0 focused on flexibility, so that it could be used for editing of parameters under many different circumstances. This was reflected in its implementation by separating the visual interface and text editing functionality into two separate components: panel and parameter editor, described in Section 1.2.2. This implementation limited the communication flow to one way, causing inconsistencies between values of parameters being displayed and values of parameters in the data files. Panel manager 3.0 combines the user interface and text editing functionality into one application, allowing two-way communication between panel manager and the data files. When panel manager 3.0 is invoked on a panel definition file, the current parameter settings are read from the appropriate data files, and the controls are then adjusted to reflect these values. Integration of user interface and text editing functionality allows panel manager 3.0 to synchronize the displayed and stored information, similar to the concept presented in [1]. Although panel manager 3.0 does not achieve the same level of

flexibility† as the old implementation, it has a wide range of editing functionality built in, sufficient for the purposes of editing parameters in data files.

The language used by panel manager to specify panels was redesigned into a more structured language, improving its readability and extensibility. Chapter 5 describes the design and implementation issues of the new panel manager in greater detail.

2.4. Portability

VLAB 2.0 was developed on the SGI platforms running IRIX (flavor of UNIX) operating system. It used many IRIX specific features, making it unportable to other operating systems. To make VLAB accessible to a wider spectrum of users, VLAB has to be portable to a number of different operating systems. I have addressed this issue in my implementation of VLAB by developing all of its components in a UNIX-portable manner [15][35]. I have tested and verified VLAB's portability by successfully porting it to two different flavors of UNIX - SunOS 4.1.4 and Linux 2.0.

2.5. Performance improvements

The main objective in the area of performance improvements was to make the new implementation of browser work faster, particularly on operations involving subtrees of objects. The main goal was to guarantee that the user would not have to wait more than a few seconds for the most common tasks to finish. To this end, browser 3.0 was completely implemented in C++ in combination with Motif/OpenGl libraries. Many inefficiencies, such as the use of external programs, were removed from both browser and panel manager. These changes have brought significant speed improvements over the old version of browser. Table 2-1 compares the timings I performed on versions 2.0 and 3.0 of browser. Most importantly, however, browser 3.0 allows its users to display icons for entire subtrees of objects in a reasonable time. This functionality was not even available in browser 2.0 - the users could only display one icon at a time.

† panel manager 3.0 can only be used for editing data files, while panel manager in VLAB 2.0 could be used for other purposes - by writing a different filter program for responding to the messages sent out by the panel component

Table 2-1: Performance comparison of VLAB 2.0 and VLAB 3.0, in seconds

Action	VLAB 2.0	VLAB 3.0
showing a sub-tree of 1000 extensions	32	20
searching in a tree of 1000 extensions	45	20
showing an icon	3	0
drag/drop a node	15	1
copy & paste 100Kb	2	2
copy & paste 1Mb	25	2

2.6. User customization

The default visual appearance of VLAB applications is not well suited for some purposes, such as presentations, because the fonts are too small and the color contrast is low. A mechanism allowing the users to change the visual appearance of VLAB applications was needed. To this end I have developed dialog windows through which the look of browser and hyperbrowser can be easily customized. The user can, for example, choose the color of various components of the two dimensional rendering of the database, the font size and type, icon size and even the layout of the graphical tree representation of the object hierarchy. The various dialogs used in the customization of browser and hyperbrowser are shown in Figure 2-2. Since all VLAB tools now use Motif library for their graphical user interfaces, it is also possible to modify their appearances through the standard X-toolkit application resources mechanism [24].

2.7. Summary

The main goal of my thesis is to implement a new version VLAB, which will address the shortcomings and limitations of the previous version. Namely, a new mechanism for creating and maintaining alternate views of object databases will be designed and implemented. A remote access extension will allow VLAB users to access remote databases transparently. A new panel manager will allow VLAB users to edit data files using graphical controls, which can be created and modified visually. The new VLAB will also improve in the areas of performance, portability and customization.

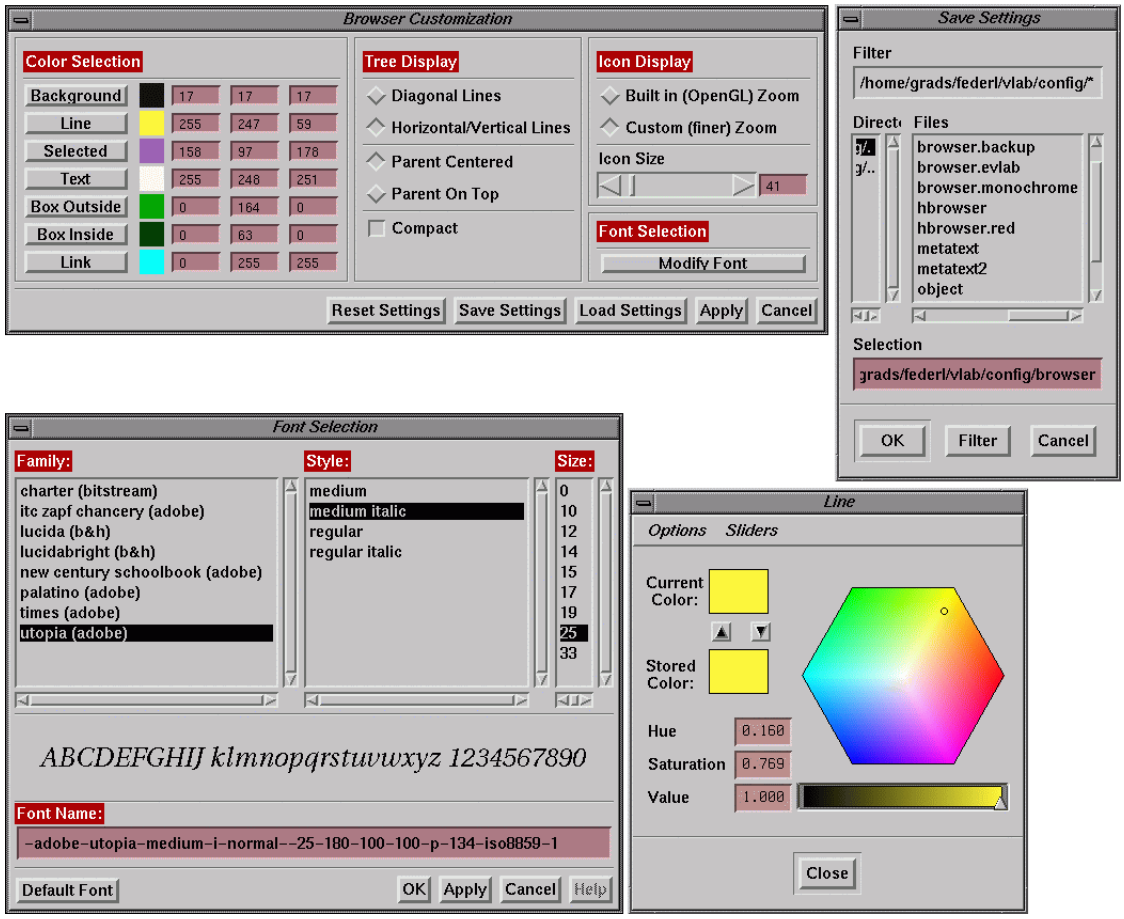


FIGURE 2-2: *Snapshot of browser's customization dialogs*

CHAPTER 3 *Survey of related concepts and previous work*

This chapter discusses concepts and previous work related to VLAB, as well as how they influence and relate to VLAB's design. The chapter is organized into two sections. In Section 3.1 I discuss previous work conceptually related to VLAB's design, while in Section 3.2 I discuss previous work related to VLAB at the implementation level.

3.1. Concepts related to VLAB

3.1.1. Monolithic versus open hypertext systems

Monolithic hypertext systems are hypertext systems which manage the storage of data as well as linking information. Their main drawback lies in their fundamental design - being complete tools they support only a limited range of media types and provide editing capabilities only through applications that are closely coupled with the underlying hypertext system. Also, the node data representation in these hypertext systems is structured and proprietary. As a result, these systems are closed, because if they do not provide the particular media type editor the user requires, only the system's developers can add the required functionality. The inability to link to external objects, as well as the inability to access the objects in the hypertext from the outside, are other disadvantages of closed systems.

The drawbacks of monolithic hypertext systems were addressed by Sun's Link Service [26]. Link Service provides support for creation and maintenance of bidirectional relationships between autonomous front-end applications. It does so by defining a protocol for open hypertext system. The system only maintains the links, the representation of the actual endpoints (objects) is left to the actual application. Link Service requires application side support for hypertext functionality, so it merely supplies the libraries which makes adding such support easier. For example, if an existing text editor was to be used in this system, the developers of this editor would have to add a mechanism to its

internals, which would allow the user to indicate the position in the text composing a new node. Also, some visual indication would have to be shown on parts of text that contain links to other nodes. This means existing application that do not support these mechanisms would have to be patched and recompiled in order to support such hypertext system. Another drawback of Link Service's approach is that the data files for nodes do not have to be stored in a standard place, which would make it nearly impossible to manipulate such database as a whole, e.g. search for information, delete a group of nodes, move the database to a different file system, etc.

The design of VLAB objects is based on the open hypertext system model. VLAB objects contain the necessary data files as well as the linking information between the data files and related applications. Users can invoke applications on the data files as specified by the linking information stored with every object. However, the format of data files is not dictated by the system but rather by the associated applications. Similarly, the applications are not part of the system, they are entirely user defined, external to the system. This gives the user absolute freedom to choose any media type for data representation, and freedom to choose any appropriate editor or any other application for the given media type. Thus, VLAB does not impose the same limitations on the user as do most monolithic hypertext systems.

3.1.2. Prototype-extension model

The mechanism for organization of VLAB objects originated from the object oriented design, namely the prototype-extension model described in [17]. VLAB users can create new objects as extensions of existing objects (prototypes), where extensions initially inherit all properties from their prototypes. Any of these properties can be later modified, for example, by adding new actions or data files, changing simulation parameters, etc. A similar prototype-extension model was used in the implementation of an interactive system for simulation, modeling and animation presented in [3]. In this system, all objects at creation time have the same properties, which change only by receiving messages. Extensions can be derived from prototypes, and these extensions can then differ from their prototypes by having received different messages.

3.1.3. Graphical versus command line interfaces

Scientists have often wondered how the world would be affected if some of the basic assumptions and principles of science were violated. It was these scientists who created new concepts, like non-Euclidean geometry, positrons and anti-matter. D. Gentner and J. Nielson tried to explore the types of interfaces that could result if the classical Macintosh

human interface design principles were reversed [10]. Macintosh interface is a very good example of the current interface paradigm, often referred to as WIMP (windows, icons, menus, pointer), which dominates the current user interface world. Undoubtedly, this paradigm suits the needs of average computer users very well, but does very poorly for experienced computer users. One of the main drawbacks of pure WIMP models is that everything has to be done through the point and click interface. Experienced users are limited to performing complex tasks inefficiently, as many UNIX users have noticed when exposed to a Macintosh-like user interface. As an example, consider a simple task of finding all files on a file system that have been modified less than 6 days ago and which contain the phrase 'University of Calgary'. This task can be performed on a UNIX operating system by executing one line at the command prompt, whereas on a standard Macintosh system each file would have to be inspected individually†. The need to support an experienced user was recognized in VLAB's design and was addressed in its implementation. VLAB's users can switch to a command line mode at any time and continue working with the objects without the limitations of a point and click model.

3.1.4. Tools

The idea of having tools associated with objects was used in Menv - Modeling and Animation Environment developed at Pixar [31]. Menv is an environment for developing production-quality and cost-effective modeling and animation systems. It is based on a modeling language called ML, used to describe and animate 3D models. Users are allowed to write their own tools, which can change various aspects of the models. VLAB users can also associate tools with components of VLAB objects and conveniently invoke these tools during experimentations.

3.2. Previous work related to the implementation of VLAB

3.2.1. Two way extensibility

Most inheritance based object oriented models provide support for extending their existing hierarchy of classes in two ways:

† At the time of writing this theses, the most recent version of MacOS did not provide any scripting capabilities.

- by adding a new class which will inherit methods from its superclasses;
- by adding a method to a superclass, which will be inherited by its subclasses.

However, in many OO designs this two way extensibility cannot be achieved under all circumstances. For example, the two way extensibility in C++ is not available when the sources of an existing class hierarchy are not available to the developer. It is still possible to create a new class by deriving it from one of the classes in an existing hierarchy. However, it is very difficult (and many times impractical) to add a new method to one of the classes in the existing hierarchy.

As an example, consider the following situation: a developer needs to add a new method to a group of classes distributed as a library. If the user has access to the sources of the library, the new method can be simply added to a common parent of all affected classes, the library would be then recompiled, and all classes would inherit the new method. Unfortunately, if the library is distributed in binary form, this is impossible to do. A work-around solution exists, where new subclasses are created from all classes that need to have the new method added, but this is an unattractive and possibly unfeasible solution for large libraries.

This two way extensibility problem and a reasonable solution was described in a paper by P. Strauss and R. Carrey [37]. This paper presents an object-oriented toolkit designed for developers of interactive 3D graphics applications, now known as Open Inventor [42]. The problem of two way extensibility was solved by using a two dimensional virtual table, in which each entry is a method that implements a certain action for a particular node. Therefore adding a new action involves adding a new row to the table, and similarly, adding a new node is equivalent to adding a new column. Since each action in Open Inventor is defined as a separate object, adding a new function to all standard nodes in the system involves creating a *new_action* class by deriving it from the provided *action* class. The resulting syntax for applying the new action to a node is then `new_action-> apply( node)`.

VLAB partially manages to address the two way extensibility problem by having a configuration file for object manager, in which the user can define actions that individual objects can associate with their data files. For example, an action called ‘EDIT’ could be defined in this configuration file as a set of sub-menus, each of which would invoke a different text editor. Then any object can use this EDIT action to be invoked on a data file, giving the user a choice of an editor to use. When a new editor is installed on a system, the only change needed to make this new editor available with all existing objects is to add a new item to the description of ‘EDIT’ in object manager’s configuration file. However, this assumes that the creator of objects will have to use globally defined functions, such as EDIT in this example, in order to take advantage of this support for two way extensibility.

3.2.2. Building graphical user interfaces

A similar idea for building user interfaces as used in VLAB's panel manager was described in a system called FormsVBT by G. Avrahami, K, Brooks and M. Brown [2]. This system allows building of user interfaces using a graphical (WYSIWYG) editor, or using a textual description of the user interface through a special purpose language. The similarity comes from the fact that the description files for each VLAB panel are also defined in a special purpose language, which can be edited either by a text editor, or by a graphical (WYSIWYG) editor. Another system that deals with similar issues of building graphical interfaces for programs is GROW [3] [32].

The Forms Library (XForms) is a library of C routines that allows the user to build user interfaces with buttons, sliders, input fields, dials, etc. Every regular XForms distribution includes a GUI builder utility called *fdesign*, where interfaces can be visually assembled. In fdesign, the individual components can be added, deleted, moved, resized, and their attributes, such as color, font, and appearance, can be edited in option dialogs. Group manipulation of objects is also possible, although editing of attributes on groups of objects only works if all objects in a group are of the same type (i.e. buttons). Therefore, there is no easy way to change an attribute for all components in a window, such as the background color or the font.

3.2.3. Frameworks for experimentation

In this section I examine three different software environments used in areas of experimental algorithms, technical computing and 3D modeling, respectively. All of these environments have some common characteristics with VLAB.

CSLAB

A framework for creation, configuration and execution of experimental algorithms was implemented in CSLAB [5]. It is a prototype of a simulation environment that is easy to use, dynamic and WWW accessible. Running and configuring experiments can be done either through its visual interface or through the use of script files. For managing experiments and their results, CSLAB provides a workbook. CSLAB is an open architecture, therefore new building blocks can be added to the system. Since all algorithm objects in CSLAB are written in Java, they can be easily shared between users around the world, whether as individual objects, or as whole experiments. This makes CSLAB a great tool for collaboration and education in the area of experimental algorithms.

VLAB also provides visual interface for managing and running experiments, through browser and object manager. Collaboration is encouraged in VLAB by making access to remote oofs databases transparent to its users.

Mathematica

Mathematica is a fully integrated environment for technical computing [43], extensively used in research and teaching. It is well suited for various user communities, as its many specialized toolkits allow for rapid system prototyping and experimentation in areas such as signal processing and control systems. The most common way to organize work in Mathematica is through the use of its notebooks. Mathematica notebooks are structured interactive documents organized into a sequence of cells. Each cell contains information of certain type, e.g. text, graphics, sounds or Mathematica expressions. Cells are organized into groups where a group can also include other groups. A group of cells can be either open or closed. When open, all its cells and groups are visible, when closed, only the first cell (or the heading cell) is shown. This allows for more comfortable viewing of large documents. Notebook actions can be programmed, for example by inserting buttons into cells, and then programming the actions that will be invoked when the button is clicked on. Buttons can be also organized into palettes, which are shown in Mathematica as floating windows, and these buttons can then be used in conjunction with any other notebook. Mathematica notebooks also provide simple hypertext functionality. Hyperlinks can be inserted into documents, which can take the user to a specific point anywhere in the notebook. The data in notebooks can be experimentally modified at any time, and the results can be automatically recomputed and then examined.

VLAB users also organize their work hierarchically, into oofs databases, which can be visually accessed using browser. Object trees can be expanded or collapsed, allowing the user to control the level of detail displayed on the screen. Various external tools can be associated with VLAB objects and invoked in object manager from pull-down menus. Eventhough VLAB does not provide support for hypertext documents, users can create alternate views of oofs databases and thus access objects in non-hierarchical fashion.

Alias

Alias from Alias/Wavefront [1] is a powerful 3D modeling environment developed for industrial designers, with a main focus on creation of mathematically accurate regular-shaped objects as well as freeform organic shapes, and photorealistic renderings and animations. Alias organizes its projects with the help of the UNIX filesystem: each project contains a set of subdirectories, where all files related to a project are stored. There are separate subdirectories for wireframe models, textures, lights, rendering results, temporary plot outputs, stages, etc. VLAB users organize their experiments in objects,

represented as separate directories, where all information pertinent to the experiment is stored in data files.

User interface in Alias is highly consistent and yet fully customizable. Most functions in Alias are invoked either through menus, tool palette, tool shelf, marking menus or hot keys. Menu functions are functions that do not require interaction within the 3D workspace, while the tool palette contains all functions that do require interaction within the 3D workspace. Menus and the tool palette cannot be customized by users although it is possible to change their layout and appearance to a certain extent, i.e. size of icons, vertical/horizontal organization, and position on the screen. Tool shelf is a fully customizable version of tool palette - tools in it can be added to, removed from, rearranged, options set and their icons modified with a paint program. Marking menus are similar to popup menus, and although they come predefined, they can be modified to include any functionality. Finally, any action in Alias can be assigned a hot-key, so that it can be invoked directly using a keyboard. Object manager in VLAB allows users to perform operations on objects using pull down menus. There are two types of operations that can be invoked in object manager: predefined actions and user defined actions. User defined actions can be arranged into sub-menus, and are used to invoke external applications on data files in objects.

Alias encourages experimentation by offering various tools for flexible management of objects, e.g. templates, layers and stages. The user can lock any subset of objects on the screen by putting them into ‘template mode’, and then modify the unlocked objects without affecting the locked (templated) ones. For a more flexible organization of work, layers can be used. Any model can include multiple layers of components, and these layers can be made fully editable, only visible, or completely hidden. Objects can also be developed independently of each other and then combined together in stage sets using a stage editor. Each object can be loaded into its own stage, and stages can be then made active, visible, or invisible. If a stage is active, all of its objects can be modified. If a stage is in a visible mode, its objects are shown on the screen, but cannot be modified, and when in invisible mode, the objects are completely hidden from the user. This allows the user to combine and reuse existing objects in complicated scenes, while keeping the objects as independent entities. VLAB also allows reuse of existing objects through the prototype-extension mechanism for object organization. New extensions of objects can be easily created and modified, while the originals remain intact. Unfortunately, VLAB does not offer any support for multiple inheritance, and therefore, multiple objects cannot be combined to create a single new object.

3.2.4. External parameter control

This section briefly describes two systems which implement functionality for external parameter control. Both systems allow their users to change parameters used in applications at run time through interfaces external to the application. VLAB supports similar functionality for external modification of parameters at run time, through panel manager.

UCofA

UCofA is a software environment for creating and running user interfaces used for external parameter control [14][27]. A visual tool for building interfaces is part of the system. The controls used for parameter modification are associated with messages. These messages are sent out by UCofA when the control is manipulated and intercepted by an interface program. The interface program translates them according to the needs of the external application whose parameters are being controlled. UCofA can be used for editing parameters in data files, by writing an interface program which translates UCofA messages into file editing operations.

Menv

As described earlier (Section 3.1.4), Menv is a modeling and animation environment in which 3D models are described using ML modeling language [31]. The variables used in ML programs can be modified by external tools at run time, through shared memory. These variables are referred to as articulated variables. Users can write their own tools, which can communicate with each other and also modify articulated variables used in ML programs. Since various aspects of models can be changed (through articulated variables) while being animated, animators can achieve a very high level of control.

3.3. Summary

In this chapter I have described previous work related to the Virtual Laboratory. I have discussed previous work conceptually related to VLAB as well as previous work related to VLAB's implementation.

This chapter discusses concepts and previous work related to VLAB, as well as how they influence and relate to VLAB's design. The chapter is organized into two sections. In

Section 3.1 I discuss previous work conceptually related to VLAB's design, while in Section 3.2 I discuss previous work related to VLAB at the implementation level.

CHAPTER 4 *Remote access server*

This chapter describes an extension that adds the capability of accessing remote databases of objects to VLAB's functionality. The implementation of this extension involved writing a new VLAB daemon, remote access server (RAserver), and a corresponding library, RAlibrary. In Section 4.1 I discuss the general problem of accessing remote databases in VLAB and summarize the issues considered in the design of its solution. In Section 4.2 I present a feasibility study, in which I evaluate four alternative mechanisms considered for the implementation of the remote access extension. In Section 4.3 I discuss the design of RAserver. In Section 4.4 I describe RAserver from the user's perspective, and in Section 4.5 I discuss its implementation details. In Section 4.6 I describe RAlibrary.

4.1. Background

The support for collaboration in VLAB 2.0 is very limited. Tools in VLAB 2.0 allow users to access databases of VLAB objects only on local filesystems. Objects have to be transferred manually (e.g. by FTP or e-mail) between users, making collaboration between scientists working from different geographic locations difficult. In order to improve the support for collaboration in VLAB, it is necessary to provide a mechanism for accessing databases on remote computers over the Internet. The old and the desired implementations of database access in VLAB are graphically illustrated in Figure 4-1. The dotted lines between workstations and databases represent supported database accesses under the two implementations.

At the conceptual level, the purpose of the remote access extension in VLAB is to:

- simplify collaboration (e.g. interchanging objects);
- allow users to modify experiments remotely (e.g. when working from home),
- make it possible to present research to the public.

At the implementation level, the new extension must fulfill the following goals. The remote access mechanism should be transparent to the user, so that there will be no

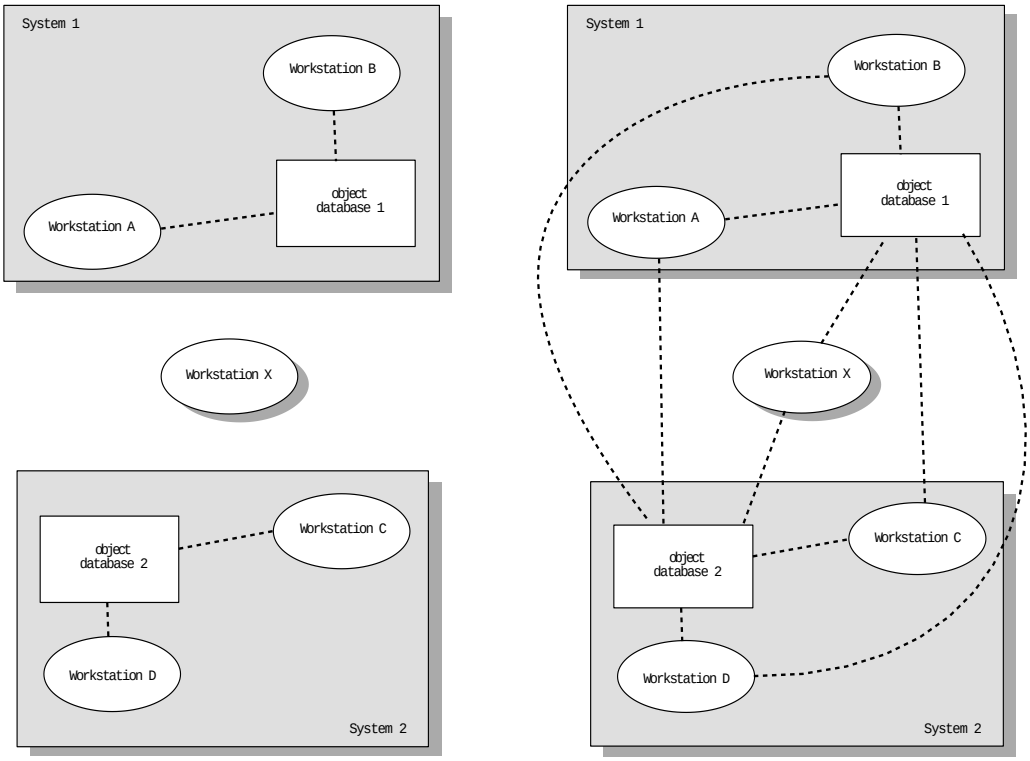


FIGURE 4-1: *Database access in VLAB 2.0 (left), desired database access (right)*

difference between accessing local or remote databases. There should be no limitations imposed on the user with regard to the functionality available when working on remote objects, or as few as necessary. The remote access should also be secure. It must be difficult, if not impossible, for unauthorized users to gain access to the remote databases. Finally, remote access should be fast and reliable. Waiting for extended periods of time to obtain information over the network, or having to worry about loosing information when working with remote objects, would render such a remote access extension unusable for VLAB's' users.

It should be mentioned that the goal of this extension is not to re-design the entire VLAB, nor is it to take a fundamental departure from its old versions. The goal is to build on top of existing (and most importantly - working) implementation, and to extend it.

Division of work between client and server

One of the most important issues related to the design of the remote access extension is the division of work between a client and a server. The client is the workstation from which

the user accesses a remote database. The server is the computer system on which the remote database resides. The problem to be addressed in the design of the remote access extension is in determining which tasks will be performed on the client, and which on the server.

There are two types of programs executed by a VLAB user: VLAB applications and external programs†. VLAB applications are used for general management of databases and objects, and for invocations of objects. External programs are required in order to experiment with the actual objects.

Execution of external programs

It is desirable that all simulation programs invoked from remotely retrieved objects are executed on the users' workstations, as opposed to running the simulation programs on the computer containing the object. First, the server cannot be assumed to be a powerful system, able to execute simulation programs of multiple users. Second, a vast majority of simulation programs used with VLAB to date are graphically intensive. Running simulations remotely would require the graphical information from these simulations to travel over a network, considerably inhibiting the user's interaction even on fast networks.

Execution of VLAB applications

The question regarding which VLAB applications should be run locally and which remotely is more difficult to answer. The disadvantage of running VLAB applications locally is related to the amount of data being transmitted over the network. For every file a VLAB application requires access to, it would need to be transferred over the network - from the server to the client. Such implementation is likely to cause unnecessary data to travel over the network. For example, if a user decided to copy and paste part of a remote database using browser, the information being copied would be sent from the database to Browser (the copy operation) and then the same information would be sent back from Browser to the database (the paste operation).

The most obvious disadvantage of running VLAB tools remotely is slow user interaction. Although the amount of graphical information for most VLAB applications is not large (such as Object Manager or Metatext), browsing large databases using Browser could become undesirably slow. Another limitation of this implementation is that its design would have to include a mechanism to allow VLAB users to run external programs on their workstations. Also, the distribution of tasks between the client and the server plays an important role when deciding whether to run VLAB applications locally or remotely. If

† External programs are applications which are not part of VLAB, such as simulation software, compilers, etc. They are required to experiment with objects.

VLAB applications were run remotely, the server would have to perform most of the tasks, while displaying the results would be the client's only responsibility. Such distribution of work is unwelcome, since it could imply heavy work-loads on the server.

A compromise must be met to address as many of these problems as possible. Some of the VLAB applications should be run directly on the server (having direct access to the remote database), and others on the client (so that the interactivity can be preserved).

Distribution of external programs

It should be noted there is yet another very important issue related to external programs. VLAB objects do not store the external programs they require, they merely contain descriptions of how to run these programs on their data files. This raises the question of how to distribute external applications to the users when objects requiring such programs are invoked. Some answers can be found by considering the use of languages designed to be portable across multiple platforms - such as Java or Python, but many questions would remain unanswered still. Since I have not found a feasible solution to this problem, it will not be addressed in the implementation of the remote access extension. I look at this issue in more detail in Section 9.2.10 as suggested future work.

4.2. Requirements

Before a final decision was made regarding which method to use for the implementation of the remote access extension for VLAB, four five solutions were examined. The subject of this section is an evaluation of these five implementation types, conducted to determine how the remote access extension requirements would be met by each. I first describe the five implementation models, and then evaluate them in ten different categories.

4.2.1. Implementation models

Model 1: X-display redirection This method is based on the capability of X Window System to run applications remotely, while displaying their graphical user interface on a local computer [30]. Using this implementation, VLAB users trying to access a remote database would have to login remotely into the server containing the database. Then all VLAB tools and external programs would be run on this server, having their user interface displayed on the local workstation.

Model 2: FTP implementation This approach would take advantage of the fact that most UNIX computers run an FTP daemon at all times [9]. All VLAB's accesses to a remote database would have to be performed by an FTP daemon running on the remote system.

Model 3: rsh/rcp implementation This implementation would require the remote user to have shell access to the system serving the VLAB object database. With shell access and an appropriate setup of the `.rhosts` file, it is possible to perform any action on a remote file using the combination of `rsh` and `rcp` commands [9].

Model 4: NFS implementation This model is based on the ability of the UNIX operating system to mount filesystems across a network, using NFS [34]. Once a filesystem is NFS-mounted on a workstation, it can be accessed as if it resided directly on the computer. This implementation type requires the server to be properly configured, so clients can NFS-mount the part of the server's filesystem containing the VLAB database. The client must also be appropriately configured, so users are allowed to NFS mount filesystems on their workstations. The process involved in accessing a remote database by a user simply involves NFS mounting the remote filesystem containing this database. VLAB applications can then access the database as if it resided on a local filesystem.

Model 5: Special purpose daemon For this implementation type, a new VLAB daemon would be designed and implemented, which would run on the remote computer and perform actions as requested by other VLAB tools. In essence, this model is similar to the FTP model, but instead of using a standard FTP daemon, a special purpose daemon would be written. As I will show in the following section, this approach would bring many advantages over the FTP implementation.

4.2.2. Evaluation of implementation models

Each of the implementation models were evaluated and compared in ten different categories. The ordering of implementations from each category are summarized in Table 4-1.

Table 4-1: Evaluation of implementation models

	Implementation Type				
	Type 1: X-displ.	Type 2: FTP	Type 3: rsh/rcp	Type 4: NFS	Type 5: daemon
amount of work required	2	3	4	1	5

Table 4-1: Evaluation of implementation models

user interface interactivity	2	1	1	1	1
file access speed	1	3	4	2	2
availability of public access	2	1	2	2	1
security	1	1	1	1	1
inter-client communication	2	1	1	1	1
limitations of functionality	3	2	2	1	1
involvement of sys. adm.	2	2	2	3	1
extensibility	1	3	2	1	1
division of tasks	4	3	2	3	1
Total	20	20	21	16	15

Category 1: Amount of work required for implementation

In this category the implementations were evaluated based on the amount of work required for their development. The winner of this category is the NFS implementation model, since its implementation does not require changes to any of the VLAB tools. The second best implementation is the X-display redirection mechanism. Only minor modifications would be required for some of the VLAB tools, namely, assuring the `DISPLAY` environment variable is properly propagated to all spawned processes. The other two implementations would require significant modifications to all VLAB tools which need to access VLAB databases. rsh/rcp implementation would require slightly more programming than the FTP implementation, because all remote shell scripts would have to be written. Finally, the special purpose daemon implementation would require most amount of work, as all VLAB applications would have to be modified, as well as a new daemon would have to be written.

Category 2 - Interactivity of the user interface

In this category, I examine the impact these models would have on the interactivity of the user interfaces of VLAB applications accessing remote databases. X-display redirection is the only implementation model in which VLAB applications and external programs would run on the server and display its graphical user interface on the client. This would yield a significantly inhibited user interface interaction, since all of the graphical information would have to travel over the network. In the other four models, both the VLAB

applications and the external programs would run on the user's workstation, preserving the same speed of user interaction as achieved when working with local databases.

Category 3 - File access speed

Here I evaluate each implementation model by how fast VLAB tools could perform file related operations on remote databases. The speed of such operations determines how fast VLAB applications can complete database management operations. The fastest file access would be achieved using the X-display redirection implementation, because all VLAB tools would be run directly on the server containing the database, thus having direct access to the database. The second best implementation models are the special purpose daemon and the NFS approaches. Eventhough all file operations performed on a remote database have to travel over the network, the NFS communication mechanism is well optimized, and so could be the communication with the special purpose daemon. File operations in the FTP implementation model would be slower than in the NFS implementation, mainly because the messages between an FTP daemon and an FTP client are text-based and therefore need to be parsed. FTP also requires opening separate sockets for each file transmission. The rsh/rcp implementation would yield the slowest file operations, as explained below.

Prototype implementation of the remote access extension

I designed, implemented and tested a simple prototype of the remote access extension using the rsh/rcp implementation model. The main drawback of this implementation surfaced during the testing period - a very noticeable degradation in performance of VLAB applications was observed when working with remote objects. Two reasons causing such degradation of speed were identified. First, a large setup time is required on the server to execute even a simple script supplied by an rsh command. During this time the operating system on the server must perform authentication of the connecting user and then set up the environment for running a shell script. Therefore, this setup time is not affected by the speed of network. I performed and timed various tests of executing commands using rsh over a fast local network, and the results revealed that this setup time was often as long as one second per call. Additionally, there is an overhead associated with each call to `system()`, needed to execute rsh and rcp commands from within C++ code. The prototype implementation revealed that the combined setup time required by the server and the overhead of calling `system()` rendered the rsh/rcp implementation too inefficient. Common tasks, such as showing extensions of an object and invoking objects, would require 5 and 20 seconds to complete, respectively.

Category 4 - Availability of public access

In this category the evaluation is based on whether the implementation method would provide public access to remote databases. Special purpose daemon and FTP are the only implementation models which support public access. Anonymous FTP allows the server running an FTP daemon to be set up so that certain parts of its filesystem are open for browsing by general public, without compromising the server's overall security [9]. Similar functionality can be achieved using the special purpose daemon. The other three implementation types do not provide any reasonable means for public access. NFS is not suitable for this purpose mainly because its design focused on the use of fast and secure networks. Allowing to NFS mount parts of the server's filesystem can introduce various security holes to the entire system, and result in high loads on the server. The X-redirection and rsh/rcp models require that the user accessing the remote database has a shell account on the server. Providing anonymous shell accounts for the general public also has the disadvantages of possible security holes and high loads on the server.

Category 5 - Security

Here I evaluate the implementation models based on how much they would decrease the existing level of security for:

- the owner of the remote database,
- the entire system containing the remote database.

If public access is eliminated from the requirements of the remote access extension, none of the discussed implementation models would reduce the existing security of the database's owner, or the existing security of the entire system which contains the remote database. This statement is based on the fact that all of the described models use mechanisms (shell access, X-display, FTP and NFS) which are considered secure and are widely used on most UNIX based systems. All implementations are considered equal in this category.

Category 6 - Interclient communication

The current implementation of VLAB only supports communication among VLAB applications running on the same computer. Since inter-client communication among VLAB tools is necessary for providing functionalities such as cut, copy and paste, transferring objects between remote and local databases would be impossible without interclient communication. It would be possible to redesign the inter-client communication mechanism in VLAB to support applications running on different systems, but this would eliminate the X-display redirection implementation model from

the category of straightforward solutions. The only implementation type under which the interclient communication would be affected is the X-redirection model, because tools operating on remote database would be running on a different host than the rest of the tools. This means that with X-redirection model, operations such as cut, copy and paste would not work between tools operating on different remote databases.

Category 7 - Limitations of functionality

The evaluation of models in this category is based on the limitations they impose on the functionality of VLAB when operating on remote databases. The functionality of VLAB applications would be unaffected using the NFS approach, because the applications would access remote databases as if they resided on local filesystems. FTP and rsh/rcp models would preserve the functionality of all VLAB applications. However, it would be impossible to execute external programs which require direct access to files in remote databases, such as file managers or shells. The X-display redirection method would introduce the most limitations to the functionality available to VLAB users when working on remote databases. This would be caused by broken inter-client communication among VLAB applications run remotely and those run locally.

Category 8 - System administrator's involvement

The implementations are evaluated in this category based on the amount of system administrator's involvement (root access) necessary to access remote databases. In the first four implementation models the system administrator of the server has to be involved. For the NFS method, the system administrator has to add an appropriate entry to the `/etc/exports` file, so that the server will allow the new client to NFS mount the filesystem containing the remote database. For the other three methods, the system administrator is needed to create new accounts on the server for each new user. The NFS implementation method would also require the system administrator of the client system to mount the filesystem with the remote database. The only implementation time for which system administrator is not needed is the special purpose daemon.

Category 9 - Extensibility

It is anticipated that in the future versions of VLAB, a need for new operations on objects will arise. For example, file locks might be needed to control simultaneous access to remote databases. The FTP model is the least extensible model, as the operations which can be performed on remote files would be limited by the functionality of the FTP daemon. The rsh/rcp implementation takes the second last place in this category, as many file operations cannot be performed using shell scripts. The other three implementation are equally extensible.

Category 10 - Division of tasks

Here I try to evaluate to what extend it is possible to divide work between the client and a server in each of the implementation models. The X-redirection model scores most poorly in this category, since it requires that all tasks be performed on the server. The client is only responsible for displaying the results of operations. FTP and NFS models are also not well suited for task division, as both the FTP and NFS daemons provide a fixed and non-extensible functionality. For example, it would be impossible using these to approaches to ask the server to scale the icons of objects before they are transferred to the client. The rsh/rcp approach does support division of tasks, however, the tasks on the remote computer would have to implemented using scripting languages - i.e. large performance penalty. The special purpose daemon is a clear winner in this category, as it can be infinitely extended to perform any task entirely on the remote system.

4.2.3. Conclusion

The NFS model is not well suited for the implementation of the remote access extension, mainly because one of the requirements of the remote extension is to provide public access to remote databases. System administrators would be reluctant to allow general internet community to NFS mount file systems. The X-display redirection cannot be used for similar reasons - system administrators cannot give out shell accesses to their system to general public. The rsh/rcp model was rejected on a similar basis, in addition to being too slow in its prototype implementation. The FTP mechanism seemed to be the most suited implementation model for the remote access extension. However, the functionality offered by an FTP daemon is fixed and therefore not extendible. In the future, when VLAB's functionality will have to increase, such inability to extend an FTP daemon's set of operations could render the whole FTP implementation obsolete. From the above evaluation, it is obvious that the special purpose daemon is the most suitable choice for implementing the remote access extension.

4.3. Design

A new VLAB daemon, called Remote Access Server (RAserver) was developed. The most important advantage of this implementation over the FTP mechanism is its ability to address the needs for new functionality in future versions of VLAB. Also, RAserver is entirely managed by the user (such as maintaining his own database of authorized users), thus removing the need for a system administrator's involvement. The only significant disadvantage of the RAserver implementation model is the amount of work required for its

implementation. In addition to the work needed to modify all existing VLAB applications to take advantage of this new daemon, the RAserver daemon itself had to be written.

Originally, RAserver was implemented to operate at the level of individual files, although its functionality has been later extended to add operations which perform requests at the level of VLAB objects. This extension can perform certain operations more effectively, as in the case of `RA_GET_EXTENSIONS_REQUEST` operation, described in Section 4.5.1.

VLAB applications can request file operations to be performed by RAserver. RAserver attempts to perform such requests, and then sends the results of these operations back to the application. The communication flow between two VLAB applications and RAserver is graphically illustrated in Figure 4-2.

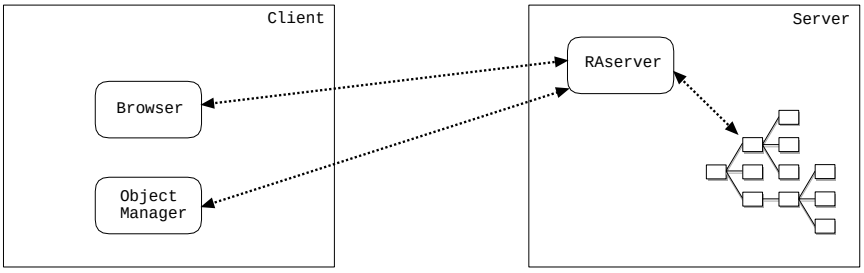


FIGURE 4-2: *Communication flow between two VLAB applications and RAserver*

RAserver spawns a separate process for each new client. This multiprocess implementation of RAserver provides a clean design, consistent with the UNIX philosophy of multiprocessing. It has very important advantages over a single-process solution. First, if a process serving one client crashes, the other clients will be unaffected, since each process runs in a separate address space. Secondly, the utilization of the server's resources will be higher than with a single-process implementation. Most of the requests made to RAserver daemon require disk access services. In a single process implementation, if one of the clients requested a lengthy operation, the rest of the clients would have to wait until such an operation is finished. This problem is avoided in a multi-process implementation because of the multitasking nature of UNIX operating systems. Last, multi-process implementation can automatically take advantage of multi-processor workstations, where each process could be run on a separate processor.

There is one aspect of single-process implementation which could make it favorable over a multi-process implementation. The single-process solution is less demanding on resources of the host computer, because with each process running on a UNIX operating system there is an associated overhead. If the number of concurrent users of a remote database was to become large, a conversion of RAserver daemon to a single-process

implementation would have to be considered, to prevent system loads from becoming too high.

The level of access to remote files granted to VLAB applications connected to RAserver is determined by two factors. First, since RAserver is run as a user's process, RAserver cannot provide functions on files that would not be granted directly to the user that invoked RAserver. The second factor that determines file access permissions is the level of access granted to the remote user, specified by the user running RAserver. The user running RAserver can set up two different levels of access for each remote user: read-only and read-write. When a new VLAB application connects to RAserver, it has to identify its user by a login and password. RAserver then looks up the corresponding account in its own database and determines its level of access. If the account is associated with read-only level of access, RAserver will refuse to perform any operation which would result in modifications to the filesystem.

To illustrate how the level of file access using RAserver is determined, consider the following example. User A has set up two accounts for RAserver: account one with write permissions, and account two with read-only access. There are three files on user A's computer: file1, file2 and file3. User A has read/write access for file1, read-only access for file2 and no access to file3. VLAB applications connected to RAserver using account one will have the same permissions to all three files as user A: read/write for file1, read-only for file2 and no access for file3. VLAB applications connected to RAserver using account two have read-only access for file1 and file2, and no access for file3. Write access for file1 has been effectively removed for all VLAB applications connected to RAserver using account two.

The RAlibrary is designed to automatically detect when the VLAB application is accessing a database on a local filesystem. When accessing a database on a local filesystem, the operations of RAlibrary are optimized - by directly performing the requested operations on the database using ordinary system calls, instead of requesting these operations to be performed by RAserver. This design allows a clean implementation of all VLAB applications requiring remote access, since the developer of the application does not have to distinguish between accessing local and remote databases. As long as the application uses RAlibrary for all its file-related operations, such an application will be automatically optimized when accessing local databases.

4.4. User's perspective of RAserver

RAserver is a VLAB tool allowing other VLAB applications to access remote databases through the Internet. A remote database is located on the filesystem of a remote computer,

and this filesystem is not shared with the computer from which the user is trying to access the database. RAserver can be run in two different modes: daemon and setup mode. In the first mode RAserver acts as a daemon, and serves VLAB applications connected through the Internet. In the setup mode accounts are created and managed for remote users.

4.4.1. Daemon mode

The user wishing to provide remote access to his/her databases invokes RAserver in the daemon mode, by issuing the following command at the shell prompt:

```
% raserver
```

RAserver then listens for connections from VLAB clients, establishes multiple connections and provides services to all authorized connections.

4.4.2. Setup mode

To invoke RAserver in setup mode, the following command is typed at the shell prompt:

```
% raserver -pe
```

When the user starts RAserver in the setup mode, RAserver presents the user with a command line interface for editing user accounts. At this prompt commands and their parameters can be entered, requesting information or changes to the user's accounts. The following commands are available: quit, help, ls, add, del, chlog, chpass, toggle. Their meaning and syntax can be obtained through the on-line help (by typing 'help').

4.4.3. Related files

RAserver stores the account information about remote users in the account file determined at runtime by evaluating the following expression:

```
$VLABCONFIGDIR/rapasswords
```

The account file is accessed by RAserver in both of its modes. If this file does not exist, RAserver in setup mode will automatically create an empty one. If RAserver in daemon mode is invoked while there is no rapasswords file, the user is appropriately notified and RAserver automatically starts setup mode.

4.5. Implementation details

In this section the implementation details of RAserver are discussed. These details include information necessary for maintaining the code as well as for adding new functionality to RAserver.

4.5.1. Overall structure of RAserver

The RAserver daemon works in the following way. When started, RAserver will make sure that the `rapasswords` file exists. If it does not exist, RAserver will switch to the setup mode. If the rapasswords file does exist, RAserver starts to listen for new clients on a port defined in `src/RA/RA.h` as `RA_PORT`. When a new client is connected, a child process is spawned to serve the client. The overall algorithm for VLAB daemon can be described with this pseudocode:

```
if option ‘-pe’ present on the command line:
     start setup_mode
if rapasswords file does not exist:
     issue warning to the user
     start setup_mode
loop:
     accept new connection
     spawn a new child process
     if child:
          serve_client()
```

`serve_client()` is a function that serves each client independently, on a separate socket connection [36]. The input to this function is the socket connection established in the main code. Before `serve_client()` serves any requests of the client, it will establish the identity of the user. The VLAB tool connecting to RAserver is responsible for sending a login request as the very first request, with a login name and a password. When `serve_client()` receives a login request, the login name and the password are matched against the entries in the `rapasswords` file. If a match is not found,

`serve_client()` will return a negative response and will terminate the connection. When authorization is confirmed, `serve_client()` will determine whether write access is granted for the connected user. Then the `serve_client()` will enter an infinite loop in which it will accept and perform requests, and return responses. Performing operations which modify the filesystem are disabled if the connected user does not have read-write access. In such a case a response indicating a failure of the requested operation is returned to the client. The pseudo-code for `serve_client()` follows:

```
serve_client( connection):
     receive request
     if request is not LOGIN request:
          terminate connection
          quit
     create an encrypted password from password
     lookup entry in rapasswords based on login
          and encrypted password
     if entry not found:
          send negative response
          terminate connection
          quit
     if entry indicates writable access:
          set writable flag
     else:
          clear writable flag
     while 1:
          receive request
          if request is a LOGOUT request:
               do_logout()
               close connection
               quit
          else if request is an UNLINK request:
               do_unlink()
          else if request is a RENAME request:
               do_rename()
          else if ...
          .
          .
          .
```

Once `serve_client()` establishes an authorized connection it will only quit when a LOGOUT request is sent, or when the client unexpectedly terminates the connection. RAserver currently implements 17 requests, designed to mimic the operations which VLAB applications need to perform on files when operating on databases:

`RA_COMPFILE_REQUEST`

- 2 parameters: file name 1, file name 2
- requests comparison between two local files located on the server

`RA_COPYFILE_REQUEST`

- 2 parameters: source_file_name, dest_file_name
- requests to perform a copying operation on the server

`RA_DELTREE_REQUEST`

- 1 parameter: directory name
- requests recursive deletion of a directory

`RA_FETCH_REQUEST`

- 1 parameter: filename
- requests contents of a file

`RA_GETDIR_REQUEST`

- 1 parameter: directory name
- requests a list of directory entries

`RA_GET_EXTENSIONS_REQUEST`

- 1 parameter: path to a VLAB object
- requests a list of extensions and their attributes for a VLAB object. This request has been added to eliminate unnecessary network traffic (the same functionality could be achieved by calling `RA_GET_DIR_REQUEST`, and then `RA_STAT_REQUEST` multiple times).

`RA_LOGIN_REQUEST`

- 2 parameters: login name, password
- requests access to RAserver

`RA_LOGOUT_REQUEST`

- no parameters
- requests RAserver to terminate connection

`RA_MKDIR_REQUEST`

- 1 parameter: directory name
- requests creation of an empty directory

`RA_PUTFILE_REQUEST`

- 2 parameters: file name, contents of a file
- requests creation of a file with supplied contents

`RA_READLINK_REQUEST`

- 1 parameter: name of a symbolic link
- requests the contents of a symbolic link

`RA_REALPATH_REQUEST`

- 1 parameter: file or directory name
- requests real path (unique full name) of a file

`RA_RENAME_REQUEST`

- 2 parameters: file name 1, file name 2
- requests rename

`RA_RMDIR_REQUEST`

- 1 parameter: directory name
- requests deletion of an empty directory

`RA_UNLINK_REQUEST`

- 1 parameter: filename
- requests deletion of a file

`RA_STAT_REQUEST`

- 1 parameter: file or directory name
- requests information about a file, such as file type (regular, directory, symbolic link, etc), and its attributes (readable, writable, executable)

`RA_SYMLINK_REQUEST`

- 2 parameters: source destination
- requests creation of a symbolic link

Each of the above requests is associated with an appropriate response message, which is always returned to the client - both in the case of a success and a failure. Some responses include only information about the success of the requested operation, some also include data. The format of messages is explained in the next section.

The following C++ classes have been implemented to simplify the development of RAserver and the associated RAlibrary (RAlibrary is the subject of section 4.5).

```
class Message;
class MessagePipe;
```

`Message` is a data structure implemented for holding information about a single message. `MessagePipe` class has been implemented to simplify the process of sending and receiving objects of type `Message` through socket connections. To demonstrate usefulness of these two classes I will show a complete implementation of `do_unlink()`, which is called as a response to the `RA_UNLINK_REQUEST`.

```
void do_unlink( const char * fname, MessagePipe & pipe)
{
    if( read_only)
    {
          // this client does not have a write access
          // send FAILURE to the client
          Message m( RA_UNLINK_RESPONSE, “n”, 2);
          pipe.send_message( m);
          return;
     }

     if( unlink( fname) // try to unlink the file
     {
          // unlink() failed
          // send a FAILURE to the client
          Message m( RA_UNLINK_RESPONSE, “n”, 2);
          pipe.send_message( m);
     }
     else
```

```
        {
             // unlink() was successful
             // send a SUCCESS response to the client
             Message m( RA_UNLINK_RESPONSE, “y”, 2);
             pipe.send_message( m);
        }
    }
```

4.5.2. Communication mechanism and format of messages

The communication between RAserver and VLAB tools is implemented using TCP/IP sockets. Each message sent between RAserver and a client, whether the message is a request or a response, follows the same format, as shown in Figure 4-3.

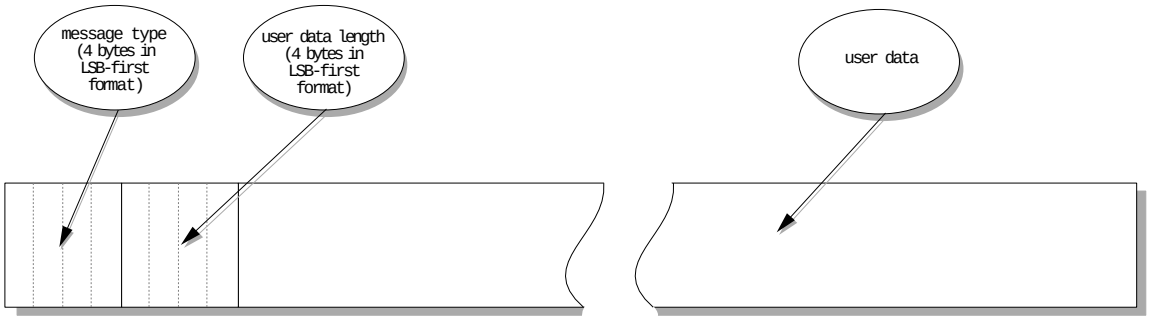


FIGURE 4-3: *Message Format in Remote Access Extension*

The first four bytes of each message determine the type of the message, for example by encoding the RA_SYMLINK_REQUEST constant into four bytes. The next four bytes determine the length of the user data included within the message. The remainder of the message is composed of user data. The total length of each message is:

```
4 + 4 + user_data_length
```

and can be determined by reading the first 8 bytes of the message. The shortest possible message is therefore 8 bytes (message with no user data), and the longest message could be approximately 4 GB long (8 + 2^32 bytes).

Strings are encoded in messages by including the terminating `\0`. If more than one string is to be sent with a message, such strings are simply concatenated, having `\0`’s as separators. If a variable number of strings are to be sent, for instance for

`RA_GETDIR_RESPONSE` message, the strings are also concatenated, and an additional empty string is included at the end.

Another common convention for all response messages is to set the first byte of the user data portion of the message to ‘y’ if the request was successfully performed, or ‘n’ if the request failed.

4.5.3. Implementation of RAserver’s setup mode

The implementation of RAserver’s setup mode is discussed only very briefly, because it is simply a command line interface for modification of the `rapasswords` file. RAserver does not cache any information regarding the `rapasswords` file - each time RAserver needs an account information, it reads the most current information from the `rapasswords` file. Also, changes made to the account list in RAserver’s setup mode are immediately written to the `rapasswords` file. This allows the user to modify the account list without shutting down the RAserver.

4.5.4. Account file format

The `rapasswords` file stores the following information about each user: login name, encrypted password, and file access level. This information is stored one line per account, where the fields are separated by colons. For example, the account file with three accounts could contain:

```
peter:qwZ9JWBYMpS/M:n:
dan:qwDq11qMqxuw2:y:
guest:qwGiniA4POTqY:n:
```

4.6. Remote access library

Remote Access Library (RAlibrary) has been implemented to provide an easy way for developers of VLAB to access the functionality offered by RAserver daemon. RAlibrary has been written in C++, as a set of functions encapsulated as static methods of a class called `RA` (Appendix A). These methods have been designed to provide the most common operations which an application performs on files. Each of these methods automatically

composes an appropriate request message, sends it over the network to RAserver, waits for a response, parses the response and returns the decoded result.

4.6.1. Optimization

The methods in RAlibrary were designed to optimize as many operations as possible. For instance, if `Compare_files()` method is called on two remote files located on the same computer, a simple `COMPFILE` request is issued to the remote server, which will compare the files directly on that machine. An alternative solution would be to download both files to the client and perform the comparison locally. On the other side, if both files are located on a local host, the operation is performed on a local computer, without accessing RAserver. Only if the connections point to two different hosts, one (or both) files have to be transferred over the network. and compared on a local computer.

4.6.2. Return values

The convention for all methods which return type `int` as a result is to return 0 on success, and values other than 0 for failure. For functions that return pointers, the convention is to return NULL in case of a failure, other values indicate success. There is a mechanism provided in class RA for determining more detailed cause of an error.

4.6.3. Example of using RAlibray

The following is source code of a complete C++ program that uses libRA to obtain a file from a remote host, display it on the screen, and then delete it from the remote machine.

```
#include <stdio.h>
#include <stdlib.h>
#include <RA/RA.h>

int main( void)
{
     // open a new connection to RAserver daemon on
     // rikki.cpsc.ucalgary.ca,
     // and login as ‘guest’ with password ‘guest’

     RA_Connection * connection = RA::new_connection(
```

```
          "rikki.cpsc.ucalgary.ca", "guest", "guest");
     if( connection == NULL)
     {
          fprintf( stderr, "RA::new_connection() "
                    " failed.\n");
          exit( -1);
     }

     // retrieve the contents of file /tmp/test.txt
     // from rikki
     char * buff;
     long size;
     if( RA::Read_file( connection, "/tmp/test.txt", buff,
                        size))
     {
          fprintf( stderr, "RA::Read_file() failed.\n"));
          exit( -1);
     }

     // display the contents of the received file
     for( long i = 0 ; i < size ; i ++)
          putchar( buff[ i]) ;

     // delete file /tmp/test.txt on rikki
     if( RA::Unlink( connection, "/tmp/test.txt"))
     {
          fprintf( stderr, "RA::Unlink() failed.\n");
          exit( -1);
     }

     // logout and close the connection
     RA::close_connection( connection);
     return 0;
}
```

4.7. Summary

In this chapter I described a new extension to VLAB, called remote access extension. It allows users of VLAB to transparently access remote databases, making it easy to invoke and interchange objects among collaborators. A new daemon, RAserver, has been developed, which runs on a remote computer and performs actions on the remote database on request by other VLAB tools. RAserver maintains a list of accounts with encrypted

passwords and access levels, thus preventing unauthorized access to remote databases. RAlibrary was designed and implemented to aid programmers in the development of applications that require RAserver's services.

CHAPTER 5 *Panel manager*

Performing computer simulated experiments often involves frequent changes to the parameter spaces of the inputs to the simulation programs. If these parameters are stored in data files, such modifications can be done using text-based editors. There are numerous disadvantages of using text-based editors for parameter modifications, such as:

- text-editing is time consuming,
- text-editing can interrupt the natural flow of a presentation,
- text-editing requires memorization of the location of parameters and valid options for parameters.

Panel manager 2.0 is a VLAB system tool designed to eliminate the need for text-based editors for parameter modification. The issues discussed in this chapter are related to the development of a new panel manager, version 3.0. First, the old implementation of panel manager is described, and some of its flaws identified. Then, the implementation goals for the new panel manager are set out, followed by the description of its design. Description from the user's point of view is given together with a short example for panel manager's usage. Implementation details related to panel manager are included at the end of this chapter.

5.1. Background - panel manager in VLAB 2.0

Panel manager is a VLAB system program used to control parameters during experiments. It displays and allows manipulation of a user-defined control panel with graphical user interface components. Every action performed on a GUI component is translated into a message, which is sent to a parameter editor. Parameter editor is a separate VLAB program which listens for messages from the controlling program (such as panel manager), and performs editing actions based on the received messages. The communication flow in VLAB 2.0 involving panel manager is shown in Figure 5-1.

Panel manager is invoked as a separate process from the application. The input to panel manager is a panel definition file, which describes the layout and functionality of the

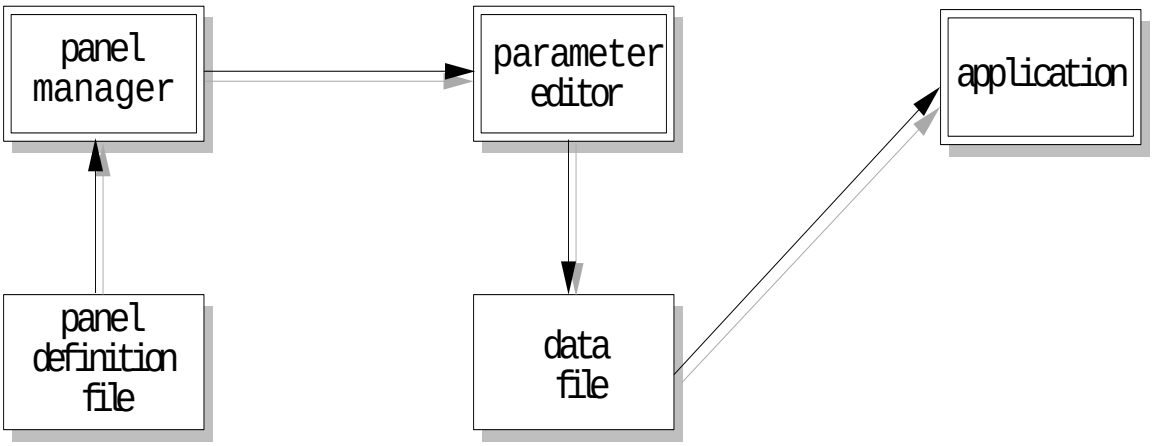


FIGURE 5-1: *Communication flow in VLAB 2.0's panel manager*

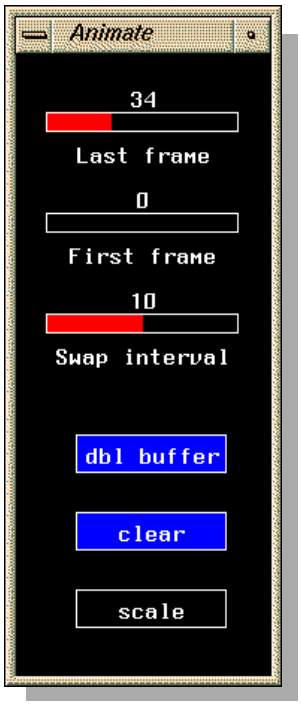
interface panel manager is to display. The panel specification file contains information about both the appearance of every GUI component, such as size, position, color, text, etc., as well as the messages to be sent whenever the GUI component is activated.

Panel manager 2.0 supports five types of components: sliders, buttons, menu items, labels and pages. Bistable buttons may be grouped so that only one button in the group will be on at any given time. An example of a panel and its associated specification file is shown in Figure 5-2.

Controls may all appear in a single window or may be divided among several pages. Popup menus are used for switching between pages, and may also contain optional user-defined items. A message format is specified for each control, and may refer to the current value returned by the control. When a control is modified, the manager will update and display its value, and send its associated message to the standard output device (`stdout`). Controls are manipulated by the left mouse button, and menus by the right mouse button.

5.1.1. Panel definition file format

The panel definition file contains all necessary information to create, display and manipulate the control panel. It first describes the window to be used, and then provides a description of each control (slider, button or menu item). Blank lines may be used to separate each control's specifications, but the details of a single control must be on consecutive lines. The panel definition file may contain the following specifications:




```
panel name: Animate
background: 0
size: 170 400

type: SLIDER
name: Last frame
colors: 7 1
origin: 20 330
min/max: 0 100
value: 34
message: n 6 1 0 %d

type: SLIDER
name: First frame
colors: 7 1
origin: 20 265
min/max: 0 40
value: 0
message: n 5 1 0 %d

type: SLIDER
name: Swap interval
colors: 7 1
origin: 20 200
min/max: 0 20
value: 10
message: n 4 1 0 %d

type: BUTTON
name: dbl buffer
colors: 7 4
origin: 40 130
value: 1
message: o 1 %d

type: BUTTON
name: clear
colors: 7 4
origin: 40 80
value: 1
message: o 2 %d

type: BUTTON
name: scale
colors: 7 4
origin: 40 30
value: 0
message: o 3 %d
```

FIGURE 5-2: *Example of a control panel and its definition file*

- The panel window
- Page definition
- Slider definition
- Button definition
- Group definition
- Label definition
- Menus

5.1.2. Interfacing with an application

Simulation programs in the virtual laboratory are assumed to obtain their initial parameter values from data files. Panel manager can be interfaced to any such program that has the ability to reread its data files. The resulting communication flow in such an interface is shown in Figure 5-1 on page 56.

The parameter editor interprets messages from panel manager and edits the corresponding parameter in the appropriate data file. The modified data file may be subsequently re-read by the application program. Thus, the steps involved in modification of the parameters are:

- user applies an action to one of the GUI components in the panel,
- panel manager sends messages to the parameter editor;
- the editor modifies the parameter file according to the messages received;
- the user instructs the application to reread the modified file.

The advantage of separating the user interface and file editing functionality into two applications (panel manager and parameter editor), is that only the parameter editor needs to be modified when connecting a panel to a different application†.

5.1.3. Example of usage

Assume a user is running an experiment, in which a simulation program `simulate` is to be invoked with file `parameters.dat`, where `parameters.dat` holds the parameters needed for the simulation. The user wishes to explore the parameter space of this experiment, and decides to use panel manager to assist him with parameter modifications. The panel specification file describing the panel's layout is created and stored in `parameters.panel`. To invoke the experiment, the user would type the following command at the command line:

```
% simulate parameters.dat &
```

To start up panel manager, the user would type:

```
% panels parameters.panel | awkped parameters.dat &
```

This command tells panel manager to read in the layout information and the message formats for each component from `parameters.panel` file. As the user manipulates the controls in the displayed panel, panel manager sends its messages to `awkped`, a parameter editor. `awkped` then performs editing actions on the file `parameters.dat` based on the messages it receives from panel manager.

† This assumes that the data file format does not change. If the format of the data file does change, the parameter editor has to be modified.

5.1.4. Drawbacks of panel manager 2.0

Although panel manager 2.0 has proven to be an invaluable tool for the users of VLAB, it has a number of shortcomings, which were identified during its frequent use. These deficiencies were generally summarized in Section 1.2.2, and the following is their elaboration.

Panel specification files have to be written in a special language using a text editor. In order to design a control panel the user has to edit the panel definition file, explicitly reload it in panel manager to see the result, and then repeat the process until a satisfactory result is obtained.

Panel manager 2.0 only supports one way communication - for sending out the messages required to perform the editing on the data files. There is no mechanism in panel manager to determine the current values of the parameters from the data file. Therefore, the values in a newly displayed panel could disagree with the actual values in the data files. The only way to avoid this asynchronism when using panel manager 2.0 is to manually store the last settings of parameters directly to the panel definition file, so that next time the control panel is invoked, the values shown by the controls will agree with the values of the parameters in the data file.

The graphical user components are implemented using an IRIS GL library, tying panel manager 2.0 to the SGI platforms. Another problem stemming from this fact is the lack of look-and-feel consistency with the rest of VLAB applications. All other VLAB applications use the Motif library for their user interfaces.

5.2. Requirements and design

5.2.1. Requirements

I have established the design goals for the new version of panel manager in direct correspondence with the limitations of the old panel manager, as stated in Section 5.1.4. They are:

- A graphical user interface builder for designing panels will be developed, where the user can design all aspects of the control panel visually (instead of using a text editor).
- A method for automatic synchronization of displayed information in control panels and the values of parameters in data files at initialization time will be incorporated into the design of the new panel manager.
- The new panel manager will provide facilities for editing various types of parameters, such as numbers and strings.
- Panel manager will allow flexible specification of parameter locations.
- Panel manager will be extensible to include new control types.
- The Motif library will be used in the new implementation for the graphical user interface, to preserve the consistency of GUIs among all VLAB components.

5.2.2. Parameter types

At the lowest level, all parameter types in ASCII data files can be categorized into two groups: numeric and non-numeric. Numeric parameters can be further classified as: integers and real numbers. Panel manager is designed to allow its users to control these three types of parameters.

Non-numeric parameters

For non-numeric parameters, panel manager provides a control component giving the user a set of a predefined choices which can be used for the parameter's value. It is often the case that the actual parameter values in data files are specified in a form which is non-intuitive to the user. Consider, for example, a parameter which can contain a value representing an RGB color, formatted as a hexadecimal number. The possible values for this parameter could be:

```
#ff0000 (Red)     #ffffff (White)
#000000 (Black)   #0000ff (Blue)
#00ff00 (Green)   #ff9900 (Orange)
#ffff00 (Yellow)  #7f00ff (Purple)
```

It is desirable the panels be constructed in such way, that alternate names for parameter values are presented to the user. The user makes the selection based on these alternate names, while panel manager will automatically use the corresponding value to modify the parameter. In the above example, the user would be given a list of colors (as shown above

in brackets) to choose from. Panel manager would automatically translate these into their hexadecimal equivalents and set the parameter in the data file using this translated value.

Numeric parameters

In order to use simple sliders for controlling numerical parameters, a valid range for the parameter must be specified. Panel manager will not allow the user to set the value of the parameter outside of this range. This design decision was made since the majority of numeric parameters can be controlled in this way, and Motif does not offer any support for controlling unbound numerical values. Interfaces, through which a user can set unbound numeric values, have been previously designed and implemented [33].

5.2.3. Extensibility

The first implementation of panel manager 3.0 only supports a limited number of controls. However, it was designed to be easily extendible when more control types are needed. To this end, each component type is defined as a C++ class, derived from a common superclass. The superclass defines default behavior for all components, as well as a group of methods which have to be defined when deriving a new component (using pure virtual functions). The process of adding a new control type to the panel manager's implementation simply involves deriving a new class from this superclass.

5.2.4. Component hierarchy

The components of each panel are hierarchically organized into a component tree. The component at the top of the tree represents the window of the entire panel. The children of the top component can be:

- control components (used for actual modification of parameters); or
- information components (e.g. labels); or
- decoration components (e.g. frames); or
- 'group' components, which themselves contain other components.

This design for hierarchical organization of GUI components stems directly from the model used in Motif applications for organization of widgets [4][12], and therefore yields a straightforward implementation. This design has many other good qualities. It is well suited for automatic inheritance of attributes. For instance, when a component is created with a specific background, then all children of this component inherit this background

color at their creation time. Another advantage of this design is that components can be grouped and then manipulated (such as resized or moved) together. It should be noted the current implementation of panel manager 3.0 only supports the first two types of components (control and information components), although an infrastructure is provided for the implementation of the other two types (decoration and group components).

An extension to this design could base the component hierarchy on the prototype-extension model. Each new component would be created as an extension of an existing one, only defining how it differs from its prototype. A change made to an attribute of a prototype would then automatically propagate to all of its extensions (provided the extensions have not redefined that particular attribute). Such a model for component organization would allow the user to create similar looking components efficiently.

5.2.5. Specification of parameter location

Flexibility was a very important issue in the design of the new panel manager. In order to maintain a reasonable flexibility of panel manager, it is desirable that the user can build control panels which operate on parameters in multiple files. Also, the mechanism for specifying the location of parameters within data files must accommodate many different file formats.

Panel manager requires the location of a parameter is specified for each control by:

- the name of the data file containing the parameter,
- the location of the parameter in the data file.

The location of the parameter in the data file can be specified in two different ways:

- by specifying the line number and the field number containing the parameter, or by
- specifying a prefix (given as a string) which can be found immediately before the parameter.

The first mode of specifying parameter's location is suitable for data files storing parameters in table format. For example, consider a data file with the following information:

```
1.2 9.81 0.0 3 5
12.3 20.7 7
```

The user would like to assign a control to the parameter whose current value is `20.7`. He could do so by specifying its location as: `line=2, field=2`. The drawback of this method for giving locations of parameters is that the parameter cannot change its location within the data file.

The second mode for specifying the location of parameters is suitable for data files which store parameters together with their textual descriptions, and their position in the file can vary. For example, consider the case where the user would like to control the gravity parameter in the following file:

```
Time Step: 0.00001
Gravity:   9.81
Friction:  1.02
```

The location of this parameter would be given as: `prefix='Gravity:'`. A control associated with a parameter whose location has been given in this manner will successfully locate the parameter even if the parameter's location is changed - as long as the string 'Gravity:' is positioned directly before the parameter. For example, this method would allow panel manager to correctly locate the gravity parameter in the following data files:

```
Time Step: 0.00001
Gravity:   9.81
Friction:  1.02
```

```
Gravity:       9.81
Friction:      1.02
```

```
Time Step: 0.00001
Gravity:
   9.81
Friction:  1.02
```

```
Friction: 1.02 Gravity: 9.81 Time Step:  1.02
```

5.2.6. Dual mode of operation

The implementation of panel manager 3.0 consists of two components: the graphical user interface builder and the visual parameter editor. Both components are combined into a single application, which can be run in two different modes. These two modes correspond to whether the user wants to design a panel, or use an existing control panel for modification of parameters. This dual mode of operation is consistent with the implementation of RAserver, which can also run in two different modes (described in Section 4.4).

5.3. User's perspective of panel manager

Panel manager can be run in two different modes: run mode and edit mode. In the run mode, the user manipulates the controls displayed in the panel, immediately translated by panel manager into editing actions. In the edit mode, the user can design new, or modify existing panels. The rest of this section will describe these two modes of panel manager in more detail.

5.3.1. Run mode

To invoke panel manager in a run mode, the user enters the following command:

```
% panels file-name
```

`file-name` determines the location of the panel definition file. Figure 5-3 on page 65 shows an example of panel manager in run mode.

Each control in a control panel can be associated with a parameter in a data file. It is possible to set up a control panel with controls modifying parameters in multiple files. At the initialization time, panel manager determines the current values of parameters for all of its controls. These are then reflected in the displayed information. If the values of the parameters are out of range, the user is notified and the affected controls display no values.

Panel manager currently supports 5 types of controls:

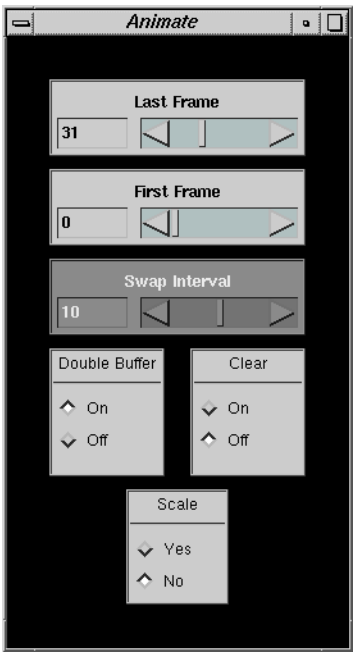
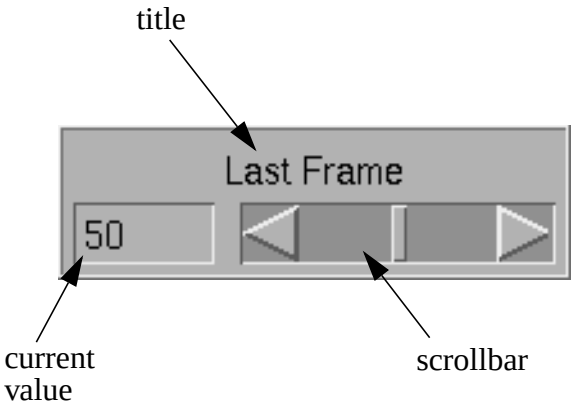


FIGURE 5-3: *Panel manager 3.0 in run mode*

- Integer range,
- Floating point range,
- Choice,
- Label, and
- Panel.

Integer range

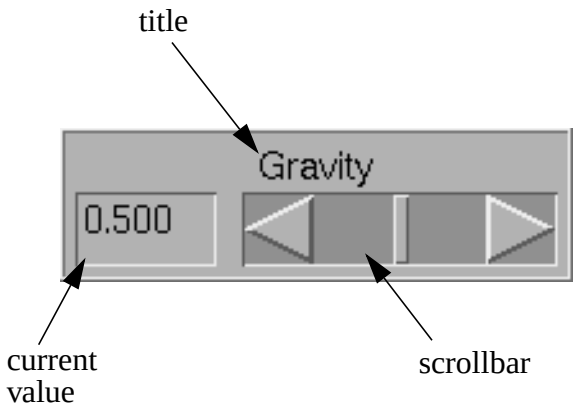
Integer range is used to control parameters whose domain is a range of integers. Each instance of this type must be associated with a minimum and a maximum value for the parameter it controls. Integer range control is made of three components:



The *title* component shows a user defined text, set at the creation time. The *current value* of the parameter is shown in a separate window in the left lower corner of the irange. The parameter can be modified by a *scrollbar*, which has 3 different manipulation mechanisms. The two arrows located at each end of the scrollbar are used to perform simple increments and decrements on the current value. The amount by which the value increments or decrements when these arrows are used is defined by the creator of the panel. Another way to increment and decrement the current value is to use the trough component of the scrollbar. Trough is displayed as the background of the scrollbar, and when clicked the value increments or decrements by a user definable amount. Finally, the user can change the value by dragging the slider left and right. The relative position of the slider with respect to the size of the scrollbar determines the new value of the parameter. The parameter is automatically updated in the data file every time the user manipulates the irange control, even when the change is not completely finalized (i.e. while the user drags the slider without releasing it).

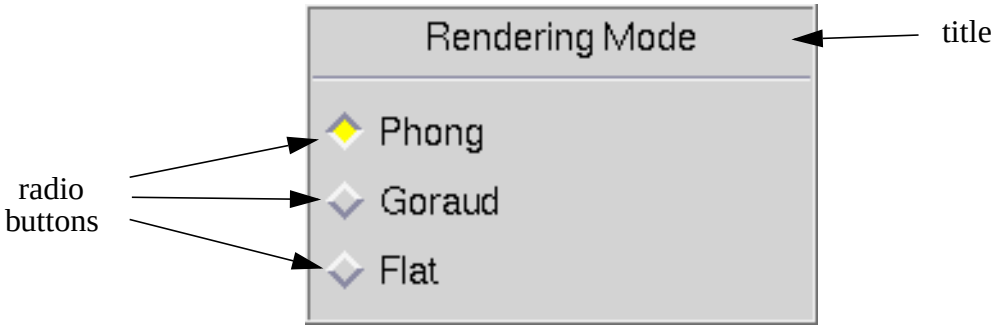
Floating point range

Floating point range is used to control parameters which can contain real numbers as their values. It is almost identical in its appearance and functionality to the integer range. The difference is that floating point range controls floating point numbers.



Choice

Choice is a control that allows the user to modify parameters which can only hold one of a predefined set of possible values. The values (their alternative names) are displayed in the control panel as a set of radio boxes, of which only one can be set active at any given time. The choice control consists of two parts:



The title shows a label that has been defined when the control panel was created. The radio buttons show the possible choices for setting the parameter's value. Each radio button is associated with a string describing the choice.

Label

The label component does not modify any parameters, and hence is not interactive. Label's sole purpose is to display a user defined text. The label appears in the control panel as a frame with a text inside:

Panel

Panel is another non-interactive component. Its function is to hold other components in the control panel. Panel is usually seen as the background of control panels.

5.3.2. Edit mode

The edit mode of panel manager is used to create or modify the layout and various other attributes of control panels. The editing actions are associated with controls while the panel is being built. When the design of the panel is done, the user saves the newly created control panel into a panel definition file. This panel definition file is then used with panel manager in a run mode, to modify parameters in data files. To invoke panel manager in edit mode, the user enters one of the following commands:

```
% panels -e
% panels -e file-name
```

The first command will open up an empty window, where a new panel will be created. The second command will result in opening an already existing panel specification file `file-name`. Panel manager in edit mode has a slightly different appearance than in run mode, namely, a menu bar is included in its window (Figure 5-4).

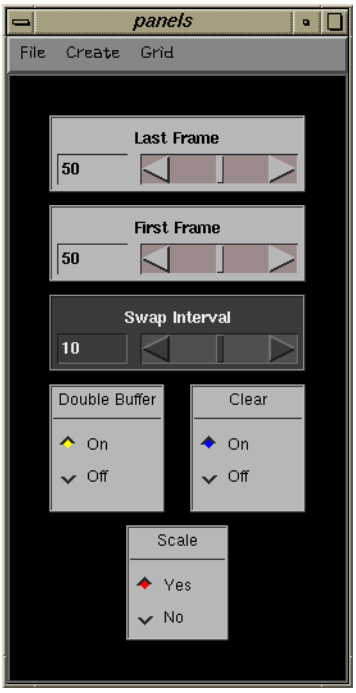
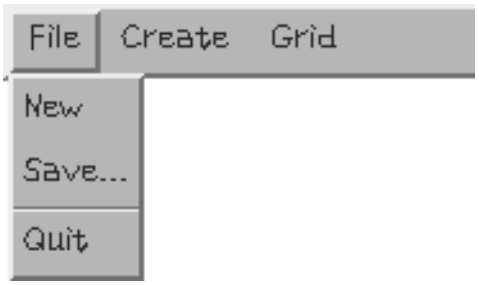


FIGURE 5-4: *Panel manager 3.0 in edit mode*

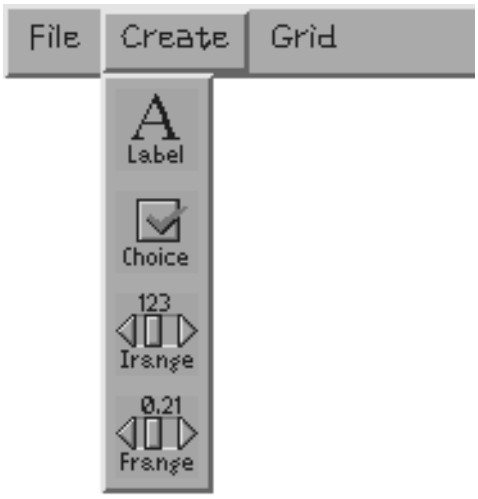
The menu bar contains three pull down menus:



New deletes all current components and creates an empty panel

Save... saves the current work into a user selectable file

Quit quits panel manager

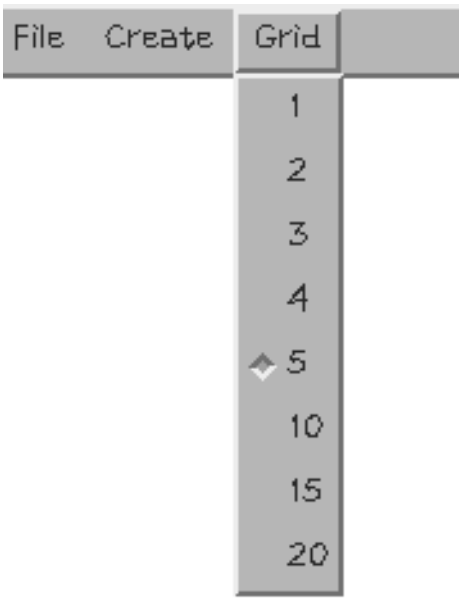


Label creates a new label component

Choice creates a new choice component

Irange creates a new integer range component

Frange creates a new floating point range component



Grid allows the user to control the granularity of the placement and resizing of the components

When a new component is created, it is given default colors, fonts, size and position. It can be then moved, resized and have the rest of its attributes modified by the user. To present the user with a visual identification of the component that is being manipulated, the active component always changes its border color to yellow.

The first mouse button is used to reposition an existing or a newly created component in the panel. No components can be moved completely outside of the panel component - panel manager will make sure that at least a portion of the component will remain visible. The second mouse button is used to resize a component. This is achieved by holding down the middle mouse button while the cursor is somewhere in the component, and then dragging the cursor in whichever direction the component is to be resized. The third mouse button is used to display a pop-up menu for a component, as shown in Figure 5-5. Panel manger currently supports only two of the actions offered by the pop-up menu: *Delete* and *Edit Attributes*. The *Delete* menu item allows the user to remove the selected component, but a warning message is first displayed to prevent accidental deletions.

FIGURE 5-5: *Popup menu for panel components*

Attributes such as background, foreground, font style, title, etc. can be edited for each component using attribute editors. To invoke an attribute editor for a component, the user selects the *Edit Attributes* function from the component's pop-up menu (Figure 5-5). Attribute editors can be displayed simultaneously for every component in the control panel.

Attribute editor for panel components

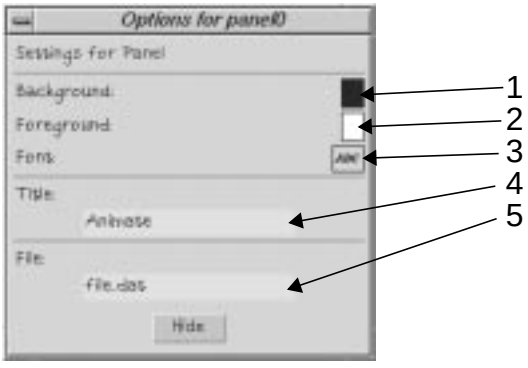


FIGURE 5-6: *Attribute editor for panel components*

When the attribute editing function is invoked on a panel component, a dialog similar to the one in Figure 5-6 is displayed. The current attributes of the panel are displayed in this dialog. To modify the *background* or the *foreground* color, the user has to click the mouse in box 1 or 2 respectively, which will pop-up a color-chooser dialog. Clicking the mouse button in box 3 will bring up a font-chooser dialog. The title and the name of the data file on which the panel controls will operate can be changed by editing text boxes 4 and 5 respectively. The contents of the title will determine the text that will appear in the title bar of panel manager window when invoked in run mode. Any change made in the attribute window will be immediately reflected in the panel component (for example when

modifying the background color by clicking on different colors in a color-chooser will immediately change the background color of the displayed panel).

Attribute editor for label components

The attribute editor dialog for Label components appears in Figure 5-7. All attributes can be modified the same way as in the attribute editor for Panel components. The contents of the *title* text box specify the text which will be displayed in the Label component.

FIGURE 5-7: *Attribute editor for label components*

Attribute editor for integer range components

FIGURE 5-8: *Attribute editor for integer range components*

The attribute editor dialog for integer range components is shown in Figure 5-8. Integer Range components have 13 attributes that can be modified. Besides the *Background*, *Foreground* and *Font*, the color of the *Top* and *Bottom Shadows*, as well as the *Trough Color* can be modified. The *Title* field determines the text displayed on the top of the Integer Range control. *Min, Max, Increment* and *Page Increments* have been explained on page 65. The parameter that the integer range component will modify is specified by two fields: *File* and *Field Prefix*. The *File* field specifies the name of the file in which the parameter is located. The *Field Prefix* field determines the position of the parameter in the file by specifying its prefix.

Attribute editor for floating point range components

The attribute editor for Floating Point Ranges is almost identical to the attribute editor for Integer Ranges. The difference is that real numbers can be entered into *Max, Min, Increment* and *Page Increment* fields.

Attribute editor for choice components

Choice component's attribute editor allows the user to edit 11 different attributes (Figure 5-9). *Background, Foreground, Top Shadow, Bottom Shadow, Font, Title, File* and *Field Prefix* attributes have the same meaning as the ones described with the range components. The two new attributes are *Toggle Color* and *Choices* fields. *Toggle Color* defines the color of the toggle of the active radio button. *Choices* field describes the choices available to the user. Each choice is described with two text fields: the left field contains the text to be displayed in a radio button, while the right field determines the value of the parameter which will be inserted into the data file when that particular radio button is selected. To delete a choice from the list, the user invokes the *Delete* button on the right side of the choice to be deleted. To add a new choice to the list, one of the *Insert* buttons is used - the position of the newly created entry will be based on the button's relative position from the top.

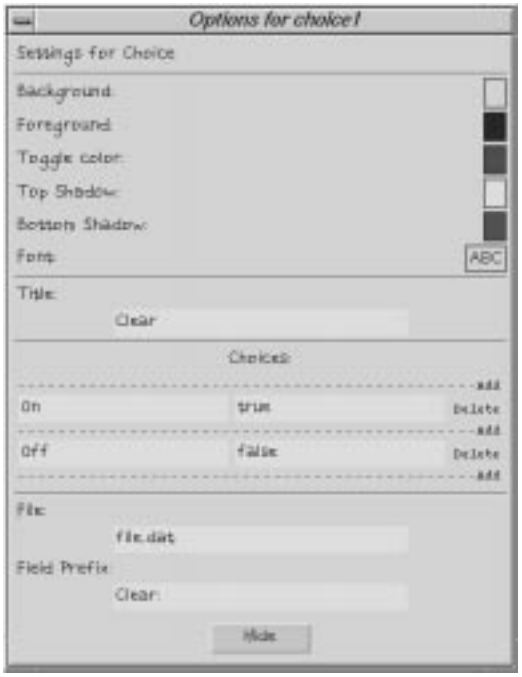


FIGURE 5-9: *Attribute Editor for Choice Components*

General comments for all attribute editors

There are two kinds of fields available in attribute editors for direct entering of information: text and numeric fields. When the information is being changed in text fields, the attribute editor notifies the associated component immediately with every change made. However, when the information is being changed in one of the numeric fields, the user has to press the **enter** key to register the changes. The reason for handling text and numeric fields differently is explained below.

When the user enters an incorrect value for one of the numeric fields, panel manager issues a warning. For example, the value in the *Max* field of both range components cannot be smaller than the value in the corresponding *Min* field. Since entering numbers usually requires intermediate states, in which the number might be in an illegal state†, the changes

† Consider entering a value of 100 as an example. When the user starts typing, the field contains no information, then '1', '10' and finally '100'.

are not registered until the user explicitly indicates that the changes are final. This prevents multiple warning dialogs being displayed while the numbers are being typed in. To visually remind the user of having to press the **enter** key when the modification of the number is done, the background of these fields turns pink as soon as the typing starts, and remains pink until the **enter** key is pressed. A detailed example of creating a complete control panel for a sample data file is presented in Appendix B.1.

5.4. Implementation details

5.4.1. Panel definition file format

Every control panel is defined as a tree of components, starting with one root component. In the panel definition file, the values for attributes for each component are defined first, and then its children. Syntactically, any component can have any number of children and attributes of any type. Panel definition files are stored in a structured text format and parsed by a parser built by yacc [16]. The structure of the file corresponds directly to the component tree of the control panel which the file represents. The grammar of panel definition files in yacc format is:

```
all->          component
components->   |
               component components
component->    ID ‘{‘ assignments components ‘}’
assignments->  |
               assignments assignment
assignment->   ID ‘=’ value ‘;’ |
               ID ‘=’ value
value_list->   ‘[‘ ‘]’ |
               ‘[‘ values ‘]’
values->       value |
               values ‘,’ value
value->        STRING |
               INTEGER |
               REAL |
               value_list
```

All capitalized tokens above are defined as regular expressions, and are parsed by a lexical analyzer built by flex [16], using following rules:

```
LETTER    [a-zA-Z]
DIGIT     [0-9]
WHITE     [\t ]+
ALNUM     {LETTER}|{DIGIT}
COMMENT   #.*
ID        {LETTER}({ALNUM}|"_")*
INTEGER   {DIGIT}+
REAL      -?[0-9]+(.[0-9]+)?
STRING    \"([^\"\n]|(\\\"))*(\n|\")
```

The following is an example of a file that has a valid syntax based on the above grammars:

```
panel {
        width = 260 ; height = 470 ;
        title = "Animate" ; background = "gray60" ;

    irange {
        x = 30 ; width  = 100 ; file = 'simulation.dat';
        y = 30 ; height = 60; prefix = 'N-iterations:';
        label = 'Number of iterations';
        min = 1; max = 100; inc = 1; page_inc = 10;
        font = "-adobe-helvetica-bold-r-normal--12-*';
     }

    choice {
        x = 30 ; y = 60 ; width = 100 ; height = 120 ;
        choices = [ [ "Wireframe", "wire" ],
                    [ "Flat", "flat" ],
                    [ "Goraud", "goraud" ],
                    [ "Phong", "phong" ]
                  ];
    }
}
```

A graphical representation of the component tree representing the control panel defined using the above file is shown in Figure 5-10.

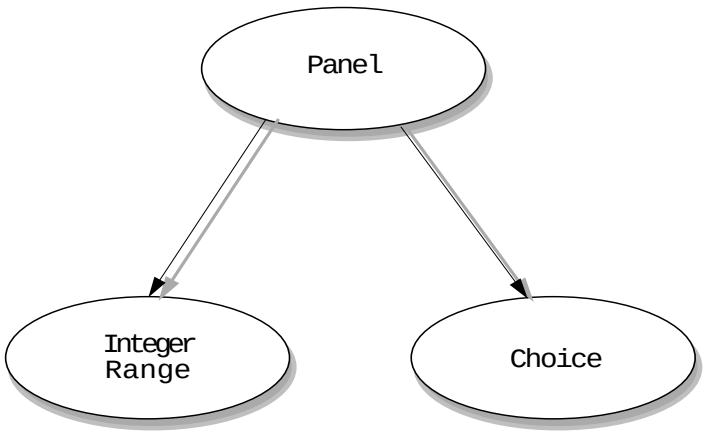


FIGURE 5-10: *Example of a component tree*

Semantically, the root of the component tree must be of type panel, and it can have any number of children. Panel's children can be of any type except Panel. In the current implementation of panel manager only the Panel components can have children, which effectively limits the depth of the component tree to two.

Syntactically, any type of a value can be assigned to any attribute. Semantically, each component has a pre-defined set of attributes and their types. Therefore an assignment 'min = -20' in a Panel component would cause an error, because panel does not have an attribute called 'min'. Also, assignment 'min = "red"' in the integer range component would cause an error, because min attribute has been defined as an integer variable. Some attributes do not have to be defined for a component, while some have to. If an attribute that does not have to be defined is not specified in the definition file, the value of the attribute is set to a default one. Similarly, if an attribute has to be specified for a component but isn't, a semantic error occurs. Assigning a value multiple times to an attribute also results in a semantic error. The list of attributes and their types for each component are compiled in Table B-1.

5.4.2. Implementation of run mode

When panel manager is invoked in a run mode, it first parses the panel definition file supplied on the command line and creates a component tree corresponding to the parsed information. Then each component is asked to retrieve its initial settings from the data file with which it is associated. After all components are successfully initialized, the tree is

recursively rendered on the screen and callbacks are appropriately setup for each component. When the user starts manipulating the components in the control panel, the appropriate callbacks are invoked. Each component type defines its own callback methods, but their functionality is similar. The callback method is first responsible for updating the interface of the component which triggered the callback, and then for updating the actual parameter in the corresponding data file.

The following is the algorithm used in panel manager to invoke its run mode:

```
    // initialize a new Motif interface
    ...
    // create a new instance of parser
    Parser p;
    // parse file_name and create tree of components
    Component * top_panel = p.parse( file_name);
    // report errors and warning (stored in p.messages[])
       ...
    if( top_panel == NULL) {
        // fatal error occurred in parsing - report to
        // the user and exit
           ...
    }
    // initialize all components
    if( top_panel-> init()) {
        // components could not be initialized (i.e. the
        // data file is missing for one of the components)
        // report to the user and exit
           ...
    }
    // render the components
    top_panel-> render( top_shell);
    // realize the whole Motif application and enter the
    // infinite event loop
    ...
```

Parser is a C++ class which encapsulates code generated by flex and yacc. It has following methods and variables:

```
public:

    Parser ( );
    ~Parser ( );
```

```
Component * parse( const char * fname);
long n_errors;
long n_warnings;
char ** messages;

static Parser * curr;

void add_error( const char *);
void add_warning( const char *);
long line_num;
Component * tree;
char * fname;
```

The `parse()` method parses a file defined by the supplied argument `fname` and returns a pointer to a tree of components. If a fatal error occurs in parsing, `parse()` will return a `NULL` pointer. The `parse()` method also sets variables `n_errors` and `n_warnings` to the number of errors and warnings that occurred during parsing, so that errors and warnings can be shown to the user. The actual error and warning messages are stored in an array `messages` of size `n_errors + n_messages`.

The constructor is responsible for initializing the variables, and the destructor frees up all memory allocated by the Parser class. The static variable `curr` was created for a lack of a better mechanism for incorporating flex/yacc generated code into C++ classes. The code generated by flex/yacc is a plain C code, and cannot be easily included in a C++ class. To allow this generated code to access variables and methods of the instance of the Parser class that invoked this code, a global variable holding a pointer to the class instance had to be created. Just before the parsing is started, the `parse()` method sets the `curr` pointer and then calls the yacc generated function `yyparse()`:

```
this-> curr = this;
yyparse();
```

The code generated by flex/yacc can access methods and variables of the Parser class by using the `curr` pointer, for instance:

```
Parser::curr-> add_warning();
```

or

```
Parser::curr-> line_num ++;
```

The variable `line_num` is used at parse time to store the current line number, and `fname` contains the copy of the file name passed to `parse()`. These two variables are needed by `add_error()` and `add_warning()` methods to format the messages to include the file name and the line number on which a parse error or warning occurred. The variable `tree` is set in the code generated by yacc when the top panel component is parsed in, and then used as a return value in `parse()`.

Component is a C++ class, from which all other component types in panel manager have to be derived. It defines a base set of types, methods and variables common to all other components, such as `Event` type, `foreground` variable and `add_child()` method. Some of the methods in the Component class are declared to be pure virtual, so that every derived class has to implement such methods. The list of Component's types, methods and variables with appropriate descriptions can be found in Table B-2.

All five controls currently supported by panel manager have their own classes, derived from the Component class: `IRange`, `FRange`, `Panel`, `Choice` and `Label`. These classes define some new attributes and methods and override some of the inherited ones. As an example, consider the Label class definition - the simplest component of panel manager:

```
class Label : public Component {

public:
                            Label( AssignmentList * al);
                            Label( );
    virtual                ~Label( );
    virtual char *          to_str( void);
    virtual void            render( Widget parentw);
    virtual void     set_geometry( long x_ret,
                                   long y_ret,
                                   long width_ret,
                                   long height_ret);
    virtual void      get_root_xy( long & x,
                                   long & y);
    virtual Widget    get_rwidget( void);
    virtual void    edit_settings( void);
    virtual void           redraw( void);
    virtual void             dump( Mem_IO &,
                                   long indent = 0);
    char label[ 256];

private:
```

```
    // callback for the EDIT mode event handler
    static void  xtEventHandlerProc( Widget,
                                     XtPointer,
                                     XEvent *,
                                     Boolean *);
    void          CommonConstructor( void);
    void             _set_highlight( Boolean h);
    // callback for the options dialog
    static void          options_cb( const char *,
                                     const void *,
                                     void *);
    Widget label_w;
    Widget frame;
};
```

Label only overrides the pure virtual methods of the Component class, and adds a few new methods needed in the edit mode. The `xtEventHandlerProc()` is installed as an event handler for all widgets of the Label component. The `options_cb()` is a callback needed by the OptionsDialog class. Both are only needed in edit mode, and are therefore explained in Section 5.4.3.

5.4.3. Implementation of edit mode

Panel manager's setup procedure for invoking the edit mode is similar to the procedure for setting up the run mode, which was described in Section 5.3.1. If a file-name is supplied on the command line, the component tree is created by parsing this file, otherwise an empty tree is created. Then each component in the tree is setup for the edit mode, by calling the `set_edit_mode()` method of the root component of the tree. After that the menu bar is created and the whole tree rendered. Then the control is given to the infinite Xt event loop:

```
    if( file_name was specified) {
        // create a new instance of parser
        Parser p;
        // parse file_name and create tree of components
        Component * top_panel = p.parse( file_name);
        // report errors and warnings
           ...
        if( top_panel == NULL) {
            // fatal error in parsing - report to the
            // user and exit
```

```
            ...
        }
    }
    else
        top_panel = NULL;

    // create the interface (menu-bar, etc.)
       ...
    // initialize all components
    if( top_panel != NULL) {
        top_panel-> set_edit_mode( handler);
        if( top_panel-> init()) {
           // report initialization failure to the user
           // and exit
           ...
        }
        // render the components
        top_panel-> render( top_shell);
    }
    // display the entire interface and give up control
    // to Motif. Callbacks will be called automatically
    // whenever the user invokes any action.
       ...
```

Calling `set_edit_mode()` of the root component puts every component in the tree into an edit mode. When a component is in edit mode, all mouse actions applied to the component are intercepted by the component's `xtEventHandlerProc()` method. This method will then call the function specified in the call to `set_edit_mode()`, and pass to it an instance of the `Event` class (Table 5-1). Therefore in the edit mode, panel manager is notified of any mouse events which occur anywhere in the component tree. These events are then translated into editing actions (such as move and resize).

The type of the handler function is defined as:

```
void handler( Component & c, Component::Event & ev);
```

The first argument which is passed to the `handler()` is the reference to the component which sent the event. The second argument contains the details about the event itself.

Table 5-1: Class event

```
int type;
```

Table 5-1: Class event

```
// This variable describes the type of the event. Possible values are:
// ButtonPress, ButtonRelease and MotionNotify.
int mouse_x, mouse_y;
// coordinates of the mouse cursor
long button_num;
// number of the button pressed or released
Event();
// constructor
XEvent * xevent;
// xevent contains a copy of the original event as sent to the
// component's event handler by the Xt toolkit.
Widget widget;
// widget contains the widget which triggered the event. For example,
// the label component is made of three different components:
// XmFrame, XmForm and XmLabel.
```

Now I discuss the implementation details for every function available to the user in panel manager's edit mode:

Save...

Saving is implemented by first obtaining the description of all components in the tree by calling the `dump()` method of the root level component. This information is then stored in a file specified in the Motif's file-chooser dialog. A warning is issued if an existing file is to be over-written.

New

The existing component tree is first destroyed by calling the destructor of the root level component, and then a new Panel component is created with no children. Before the old panel is destroyed, the user is notified and given a chance to cancel the operation.

Quit

The user is asked for a confirmation, and then the program exits.

Create

When one of the four component types is selected for creation, panel manager makes sure that the currently selected component can accept children. If not, its closest ancestor that can have children is found, and the new component is added to this component. Creation of a component involves constructing it, adding it to the parent, setting it to an edit mode, initializing and rendering.

Grid

Selecting a new grid results in setting a global variable `grid_size` to a new value. Grid size affects only move and resize functions. For moving, the left upper corner always snaps to the closest grid point. Resize snaps the resized edge to the closes grid line.

Move

Moving involves three events received by the `handler()` function. Moving is initialized with a `ButtonPress` event with the `button_num` set to 1. Then a series of `MotionNotify` events follow. Moving operation is finished when a `ButtonRelease` event is received.

`ButtonPress` event is handled by storing the pointer to the component which sent this event, its original geometry and the coordinates of the mouse cursor in state variables. The selected component is also highlighted, by first calling `highlight_off( True)` on the root component, and then `highlight_on( False)` on the component that sent the event. When `MotionNotify` event is received, the displacement between the original and the new position of the mouse cursor is added to the stored geometry of the component, then snapped to `grid_size` and applied to the component by calling its `set_geometry()` method. `ButtonRelease` event will cause a reset of the state of the `handler()`, so that further `MotionNotify` events will not result in moving of any components.

Resize

Resizing is also implemented as a response to the three events mentioned with the description of the move operation. The response to the `ButtonPress` and

`ButtonRelease` is almost identical as with the move operation, except that with `ButtonPress`, an additional state variable indicating which edges are being resized is initialized. When a MotionNotify event is received, depending on which edges are being moved, the geometry of the selected component is updated, or new edges are added to the ones being moved. When the set of edges being moved is updated, the mouse cursor changes the shape appropriately, as shown in Figure 5-11.

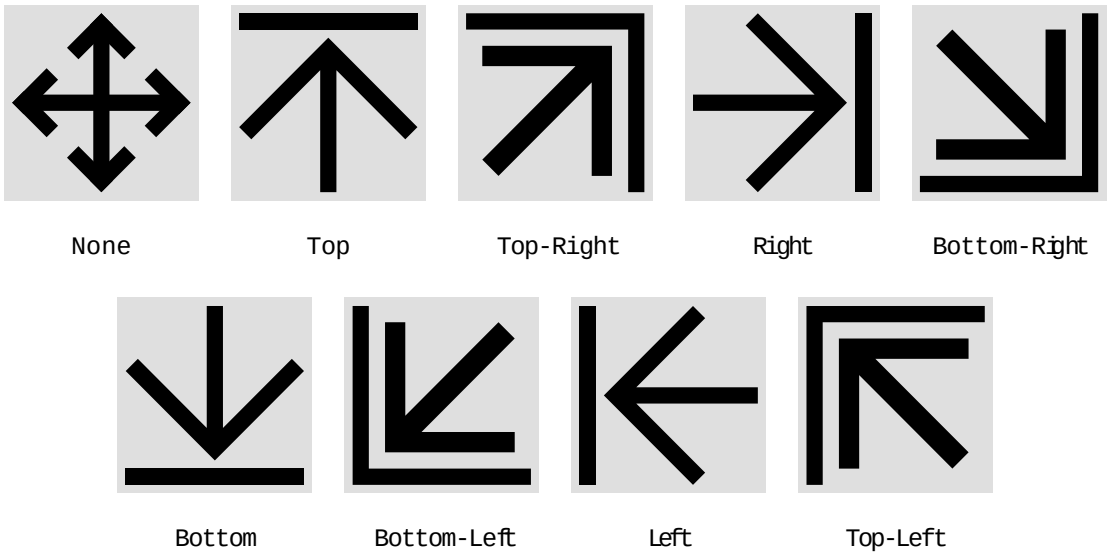


FIGURE 5-11: *Resize cursors*

Delete

The delete operation is executed by destructing the selected component. The destructor of the component is responsible for releasing all resources allocated by the component, such as colors, fonts, widgets, other dialogs (color choosers, font choosers), etc. Since the destructor in the base class `Component` includes a call to `remove_child()` method of the parent component, the parent component will be automatically notified about the deletion of its child.

Edit Attributes

Edit attributes is performed by calling the selected component's `edit_settings()` method. This method in turn creates an instance of a class `OptionsDialog`, which will create a user interface for editing all component's attributes. The following is the implementation of the `Label::edit_settings()` method.

```
void Label::edit_settings( void)
```

```
{
    // if options dialog not created, create:
    if( this-> options_dialog == NULL)
    {
        char title[ 4096];
        sprintf( title, “Options for %s”, name);
        options_dialog = new OptionsDialog();
        options_dialog-> init(
            top_shell,
            “label_settings”,
            title,
            options_cb,
            (void *) this,
            “l1”, OD_LABEL,     “Settings for Label”,
            “--”, OD_SEPARATOR,
            “c1”, OD_COLOR, “Background:”,
                            background.get(),
            “c2”, OD_COLOR, “Foreground:”,
                            foreground.get(),
            “f1”, OD_FONT, “Font:”,
                            font.get(),
            “--”, OD_SEPARATOR,
            “t1”, OD_TEXTFIELD, “Title:”, label,
            NULL);
    }
    options_dialog-> manage();
    return;
}
```

The options-dialog is only created once per component. At all subsequent times it is only managed. The above code makes it possible for the user to modify four attributes of the label component (as shown in Figure 5-7 on page 72). The options-dialog will be responsible for calling the `options_cb()` method of the Label component any time the user makes a modification to one of the four attributes. `options_cb()` will be then responsible for updating the current rendering of the Label component. The `OptionsDialog` class is explained in more detail in Section 5.4.4.

5.4.4. Options dialog

The OptionsDialog class has been developed to simplify the development of new components, namely for providing of user interfaces needed for modification of the various attributes of components (Attribute Editors). OptionsDialog has been designed in

such way that the programmer simply lists all attributes (their names, types, and default values) when the dialog is being created, and supplies a function which will be called every time one of these attributes changes. OptionsDialog is responsible for management of the user interface allowing modifications of these attributes and also for calling the supplied callback function with the new values of the attributes. The `init()` method has the following type:

```
void OptionsDialog::init( Widget parent,
                          const char * widget_name,
                          const char * title,
                          OptionsCallback callback,
                          void * user_data,
                          ...);
```

`parent` is usually the top shell widget of panel manager and is used in options dialog as a parent of its own widget hierarchy. `widget_name` specifies the name which the options dialog will assign to its top level widget's instance, so that the dialog can be customized through the standard X-resources mechanism. `title` specifies the name to appear at the top of the options dialog's window. `callback` is a pointer to the function which will be called whenever one of the parameters is modified by the user, and `user_data` is a user supplied data which will be passed to this callback. The rest of the parameters is a list of interfaces to be created in the dialog, terminated by a `NULL` pointer. The format of each entry in this list is:

```
name, type, arguments
```

where `name` is a user defined name of the interface, `type` determines the type of the interface, and `arguments` is a list of parameters (depending on `type`, zero or more parameters can follow). The interfaces requested in the list are rendered in a column, with an optional scrollbar displayed on the right side of the dialog when the combined size of the entire dialog would exceed 1/2 of the screen size. `OptionsDialog` currently supports 8 types of interfaces:

`OD_SEPARATOR`

This interface is used to separate other interfaces in the OptionsDialog by a horizontal line. It does not generate any callbacks.

`OD_LABEL, title(char *)`

Label is used to add text between interfaces. The text is determined by the supplied parameter `title`. Label does not produce callbacks.

`OD_TEXTFIELD, title(char *), default(char *)`

Textfield is used to modify strings. `title` specifies the text to be displayed in the interface, while `default` is used as an initial value of the text to be modified. When the user modifies the text (displayed in an `XmTextFieldWidget`), the user callback function is called with the value of the new text.

`OD_COLOR, title(char *), default(char *)`

Color interface is used to modify colors. `title` is a string which is displayed in the interface, and `default` is the initial value of the color. The current value of the color is displayed in the interface as a square of that color. When clicked in this box, a `ColorChooser` dialog is displayed which will allow the user to modify the color. When the color is modified using the `ColorChooser`, the `OptionsDialog` will invoke the callback function with the new value of the color.

`OD_FONT, title(char *), default(char *)`

This interface is used to select different fonts. `title` specifies the text to be displayed in the interface, while `default` determines the initial font. The current font is shown in the dialog as a rectangle displaying the letters ABC in the currently selected font. Clicking anywhere in this rectangle will bring up a `FontChooser dialog`, where the user will be able to modify the font. Any change to the font will result in a callback, passing the new value of the font as a parameter.

```
OD_INTEGER, title(char *), default(long)
OD_REAL, title(char *), default(double)
```

These two interfaces are used to modify integer and real values. `title` specifies the label of the interface, and `default` is the initial value of the number. As with the `OD_TEXTFIELD` interface, the current values are displayed and can be modified in an `XmTextFieldWidget`. The changes made to the numbers are however not sent automatically to the callback function, instead, they are sent only when the user presses the ENTER key.

`OD_DOUBLE_LIST, title(char *), n_values(long), list1(char **), list2(char **)`
This interface is used to modify two lists of strings. The two lists are specified in `list1` and `list2` parameters, and their length in `n_values`. The current values of both lists are displayed in two columns of `XmTextFieldWidgets`. Buttons for insertion and deletion anywhere in these lists are also created. When a change is made to the double list, the callback function is called with the new contents of the lists passed as a pointer to a `DoubleListCBD` structure with fields: `size(long)`, `list1(char **)` and `list2(char **)`.

The type of the callback function used by options dialog is defined as:

```
void callback( const char * comp_name,
               const void * data,
               void * user_data);
```

When this function is called by the `OptionsDialog` as a result of a change to one of the values, `comp_name` contains the user defined name of the component, data points to the new value, and `user_data` contains the data as passed to `OptionsDialog::init()`.

5.5. Summary

The design of panel manager has been improved, resulting in the implementation of a new version. Panel manager now allows users to create and modify panels visually. The new panel manager also supports two-way communication, thus removing the inconsistency between the information displayed in the panel and the information contained in the data files.

CHAPTER 6 *Browser*

Browser is a VLAB program which allows easy browsing and modification of oofs databases. This chapter describes my implementation of browser 3.0, whose design was largely based on the functionality provided by browser 2.0. Browser 3.0 implements new design concepts, namely support for collaboration and external references to objects. Also, compared to version 2.0, the new version improves the areas of portability, performance and customization.

6.1. Design

Browser 2.0 was designed and written by Earle Lowe. Since it was never developed past the prototypical stage, large parts of it remained implemented using Tcl/Tk (language/ library combination). Browser 2.0 was noticeably slow, unreliable and did not provide a consistent user interface with the rest of the VLAB tools. Browser 3.0 is a complete re-implementation of version 2.0, with many additional features.

6.1.1. Support for external references to VLAB objects

There is no support in VLAB 2.0 for maintenance of external references to objects in oofs databases. External references have to be stored as UNIX paths, often rendered invalid with the smallest organizational change made to an oofs database (e.g. by renaming an object). This problem has been addressed in VLAB 3.0 by assigning an identification number (ID) to each object, unique within its oofs database. Once generated, the ID for an object does not change. External references to objects can now be stored as the IDs of the referenced objects. This new support for external references is used in the implementation of a mechanism for creating alternate views of object databases, described in Chapter 8.

6.1.2. Objects and oofs databases in VLAB 3.0

Organization of data in objects and overall structure of oofs databases in VLAB 3.0 are almost identical to those implemented in VLAB 2.0, described in Section 1.2.1. Two changes have been introduced in VLAB 3.0 to accommodate the support for external references:

- addition of an identification number to all VLAB objects;
- addition of an object lookup table for each oofs database.

Identification numbers

The ID number is stored in textual form in a file `.id` located in the object's directory.

Object lookup table

Given the ID of an object, it is inefficient to traverse the entire oofs database to locate the matching object. To this end, VLAB 3.0 maintains the list of objects and their IDs for every oofs database in an object lookup table. The object lookup table is stored in a file `.dbase`, located in the directory of the root object and has the following format:

```
number-of-objects
ID1 location_1
ID2 location_2
ID3 location_3
     .
     .
     .
```

Object locations in an object lookup table are stored as their relative paths from the root of the oofs database. This makes it possible to move the entire oofs database to a new location without rendering the contents of its `.dbase` file invalid. Whenever a change is made to an oofs database, its `.dbase` file is automatically updated to reflect such a change.

6.2. User's perspective of browser

Browser provides the user with a two-dimensional view of the database (Figure 6-1). Each object is represented by a folder symbol, object name, and an optional icon. Prototypes and extensions are connected by lines forming a tree structure. Objects with folder symbols of type 3 have extensions, while objects with folder symbols type 1 represent leaves of the tree. Folder symbols that contain the letter 'L' indicate symbolic links to different object databases (usually the object oriented file systems of other users). Whenever objects are created, copied, moved, or deleted, browser dynamically updates the displayed tree. If the tree of the object hierarchy does not fit into the window, the scrollbars can be used to adjust the view in both horizontal and vertical directions.

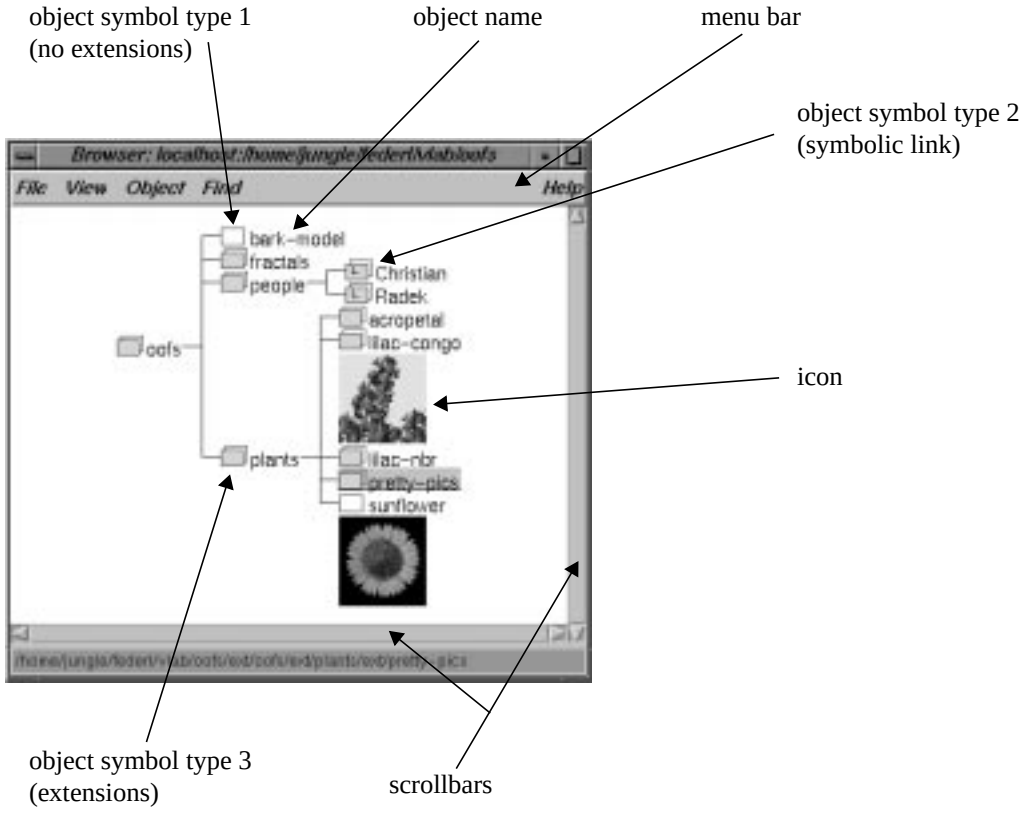


FIGURE 6-1: *Browser's window*

6.2.1. Start-up information

Browser is usually invoked from the command line, using the following syntax:

```
browser [-p password] [[[login@]hostname:]dirname]
```

Valid examples of invoking browser are:

```
browser
browser ~/vlab/oofs
browser acs6.acs.ucalgary.ca:/scratch/vlab/oofs
browser joe@cs2:/usr/u/vlab/oofs
browser -p ecret678 joe@cs2:/usr/u/vlab/oofs
```

Browser assigns the following default values for unspecified parameters:

```
password = NULL (unspecified)
login = the current user name (whoami)
hostname = localhost
dirname = $(VLABROOT)/oofs
```

When browser is invoked on a remote database, the user is prompted to enter his login name and password (Figure 6-2), unless both the login name and password are specified on the command line. If the authentication process with RAserver fails, the user is notified and prompted for the login information again.

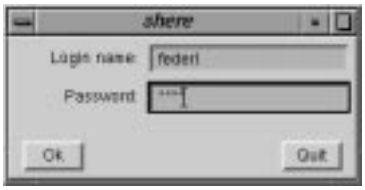


FIGURE 6-2: *Browser's login window*

Menu bar

The menu bar provides an interface to most of the functionality available in browser. The ***File*** menu groups actions related to general operations of browser. The ***View*** menu contains operations used to change the view of the database. The ***Object*** menu groups actions for database management and object invocations. The ***Find*** button is used for searching through object databases. The ***Help*** menu contains operations for invoking browser's on-line documentation. A detailed description of the operations available through these menus follows:

File: New browser

Invokes a new browser window with the initial view of the same oofs database.

File: New hbrowser

Invokes a new hyperbrowser window with a view of the hofs database associated with the oofs database currently being displayed by browser.

File: Open shell

Opens a UNIX shell window. The directory in this window is automatically set to be the directory of the selected object. This is a useful feature for users wishing direct access (command line) to the internals of objects.

File: Open file

Displays a file selection dialog, listing files in the directory of the selected object. If no object is selected, the files in the directory of the root object are listed. When a file is selected from the file selection dialog, browser invokes a text editor on this file.

File: Customize

Opens a dialog window where the user can customize the visual appearance of browser, i.e. change the colors, font, etc.

File: Exit

Exits browser.

View: Show extensions/Hide extensions

Toggles the display of immediate extensions of the selected object.

View: Show all extensions

Shows (recursively) all extensions for this object. Symbolic links to objects are not expanded, unless the currently selected object is a symbolic link. This prevents browser from entering an infinite loop when owners of oofs databases have cyclical links to each other's databases.

View: Show icon/Hide icon

Toggles the display of the thumbnail icon for the selected object.

View: Show all icons

Recursively shows thumbnail icons for the highlighted object and all of its displayed extensions.

View: Hide all icons

Recursively hides all thumbnail icons of the selected object and all of its displayed extensions.

View: Center object

Adjusts the view of the database so that the position of the selected object in the browser's window is as close to the center as possible.

View: Begin tree here

Hides all ancestors of the selected object. After this operation is applied, only the tree starting at the selected object remains visible. This operation can be reversed by the following operations.

View: Show parent

The immediate parent of the selected object and all of the parents extensions are displayed.

View: Begin tree from root

Shows the tree beginning at the root of the object hierarchy independently of the currently selected object.

Object: Get

Invokes the object manager on the selected object.

Object: Rename

Allows the user to rename the selected object. The user is prompted for a new name which has to be entered in a dialog window. If the selected object cannot be renamed as requested, the user is notified appropriately.

Object: Cut

After a confirmation is obtained from the user, the selected object and all its extensions are recursively copied into a temporary space (clipboard) and then deleted from the original location. This entire tree can be later copied into any other location (including different databases) using the ***Paste*** function.

Object: Copy node

Copies a single object (without its extensions) to the clipboard, from where it can be pasted.

Object: Copy subtree

Copies the selected object and its entire subtree to the clipboard for a subsequent paste operation.

Object: Paste

The object and its extensions, if any, stored in the clipboard become an extension to the selected object. The user is notified if the paste operation cannot be completed.

Object: Delete

After a confirmation from the user, the selected object and all of its extensions are removed from the object oriented file system. If the delete operation cannot be completed, the user is notified.

Object: Keep h-links/Move h-links

The state of this toggle button determines how the object IDs are affected when objects are copied. When the toggle button is set to ‘Keep h-links’, the IDs remain associated with the original objects and new IDs are created for the new copies. When the toggle button is set to ‘Move h-links’, the IDs are re-assigned to the copies, while new IDs are generated for the original objects.

Find

Makes it possible to search for an object in the object tree by specifying a substring of the name that is being looked for. When a match is found, the object is located in browser’s window by expanding the appropriate branches of the database tree, and the user is given the option to either continue searching for the next match, or to abort the search completely. The choice of whether the find algorithm will expand symbolic links when searching for the object is selectable by the user (Figure 6-3).

FIGURE 6-3: *Browser’s find dialog*

Help: About

Displays general information about the current version of browser.

Help: On-line help

Invokes an on-line documentation using SGI’s showcase.

Mouse operations

In addition to selecting menus from the menu bar, the mouse can also be used to perform a number of actions.

Left Button

- Clicking on an object's folder symbol, name or icon selects the object.
- Double clicking on the object's folder symbol places the object on the lab table by invoking the object manager on the object.
- Double clicking on the object's name shows or hides object extensions.

Middle Button

- Selects an object and makes it possible to copy it using the *drag and drop* operation. The user selects an object and drags it to a new location, where a new child is created. The view of the object hierarchy is automatically scrolled if the destination object is located outside the viewing area. *Drag and drop* operation only copies the object, not the object's extensions. Its functionally is equivalent to performing ***Copy node*** and ***Paste*** operation. Dropping an object on itself or releasing the mouse button with no object selected cancels the *drag and drop* operation.

Right Button

- Clicking on the object's name or folder symbol shows or hides the object's thumbnail icon (toggle action).

Advanced features

In order to allow the user to simultaneously view different parts of a database, multiple copies of browser can be invoked and each browser can then display different parts of the database. Similarly, multiple copies of browsers can be invoked on different databases, allowing the user to easily transfer objects between databases through the use of drag/drop or cut/copy/paste functions. All instances of browser invoked by the same user can communicate using VLAB daemon, described in Section 1.2.2. Changes made to a database in one browser are broadcasted to all other browsers to maintain consistency between the real and displayed information. It should be noted that if more than one user accesses the same database at the same time, changes made to this database by one user are not automatically reflected in browser invoked by the other users.

Customization

The look of the visualized object database can be customized by the user through a customization window. The customization window (Figure 6-4a), is invoked by selecting the ***Customize*** menu item. The user can choose various colors used in browser's display,

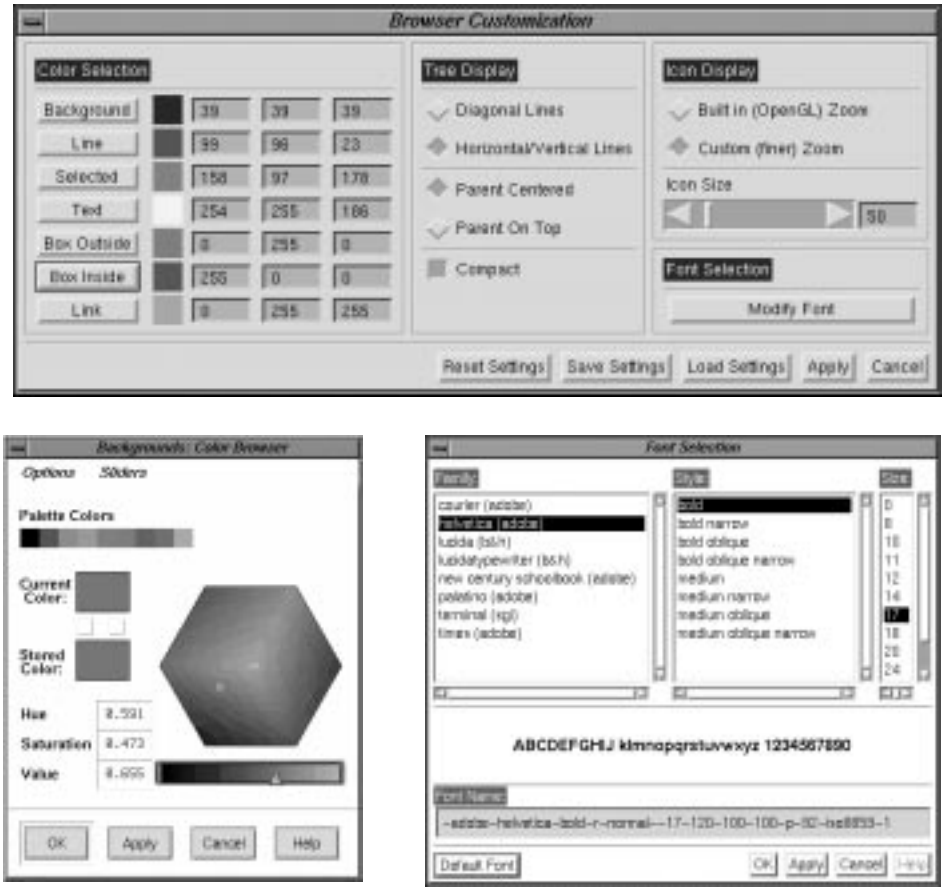


FIGURE 6-4: *Browser's customization dialogs: a) main dialog, b) color chooser dialog, c) font chooser dialog*

change the format of the tree display, modify fonts and select icon size and icon zoom methods. Colors and fonts can be changed using color and font chooser (Figure 6-4b and Figure 6-4c). Push-buttons are located in the bottom of the window for saving or loading the selected settings, and to apply or cancel the selections. By default, custom settings are saved in a file specified by `$(VLABCONFIGDIR)/browser`. A different file can be specified by the user for saving and loading settings for browser.

The appearance of browser can also be changed by modifying its application resource file [6][24][30]. This file is stored in a file `app-defaults/Browser`.

6.3. Implementation details

Since the implementation of browser consists of over 12,000 lines of C++ code, not all of its implementation details can be covered in this thesis. Only the essential aspects of browser's implementation are discussed.

Remote access

To take advantage of the functionality provided by RAserver, browser performs all filesystem operations on the object databases using methods from RAlibrary. Since RAlibrary is optimized for performance when operating on local filesystem, browser's performance is not affected by this design. Also, when the network connection between browser and the remote database is fast, the speed of browsing remote databases is comparable to the speed of browsing local databases. Accessing a remote database on a server connected to the client on a fast local network is even faster then accessing the database using NFS. Such speed improvement can be attributed to a better distribution of tasks between a client and a host when using RAserver, especially when RAserver is performing tasks at an object level as opposed to at a file level (e.g. the operation associated with `RA_GET_EXTENSIONS_REQUEST`, described in Section 4.5.1).

Support for external references

Every time an object database is modified, the object lookup table for that database has to be appropriately updated. In some cases the `.id` files for each involved object have to be adequately adjusted. For example, when an object is cut from an object database, its entry has to be removed from the object lookup table. But when the object is pasted back into the database at a different location, its old `.id` file has to be re-used and re-entered into the object lookup table. Otherwise external references to the modified object would become invalid. However, if an object is cut and pasted between two different object databases, the `.id` file cannot be reused, because each object database has a different numbering scheme.

Drawing trees

Two distinct algorithms for graphical tree layout have been implemented (Figure 6-5). The *sparse* layout algorithm allocates a separate vertical space for each sibling's subtree, so that other siblings cannot intersect this vertical space. This vertical space is recursively determined by summing up the heights of each subtree of a node. The second algorithm is based on the algorithm developed by Moen [22], which allows the trees of siblings to intersect each other's vertical space, as long as the actual nodes and interconnecting lines

do not cross. This algorithm is based on calculating and merging contours of trees, where the contours are represented as poly-lines.

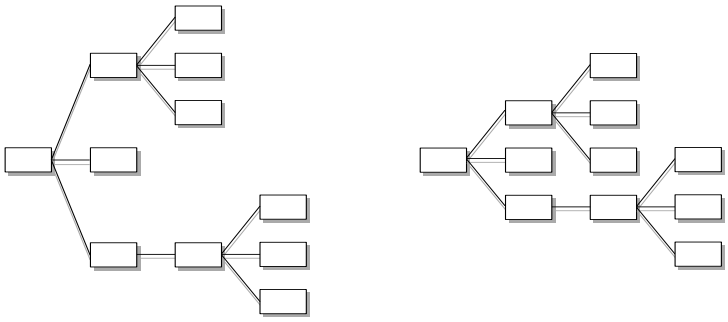


FIGURE 6-5: *Two different tree layout methods: sparse (left) and compact (right)*

Some minor options for tree renderings have also been added, namely diagonal versus vertical lines, and parent positioning center versus top. Figure 6-6 illustrates these various modes of rendering using the compact layout. These rendering methods, together with selectable font styles and sizes, colors and icon sizes make browser highly customizable.

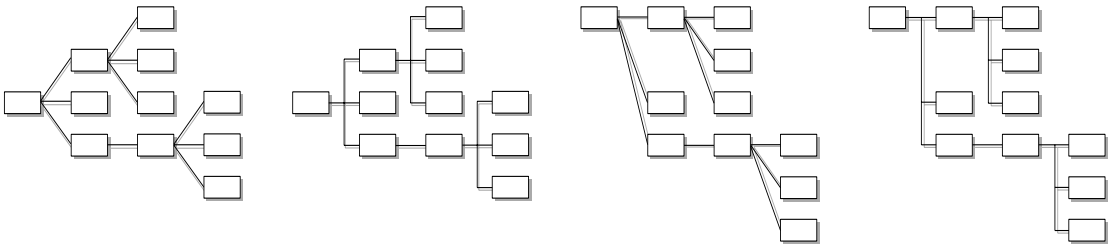


FIGURE 6-6: *Different tree drawing methods*

To increase the performance of browser, OpenGL display lists have been used for drawing of all folder symbols, icons and texts. The tree is drawn into a double buffered 24 bit OpenGL widget. Double buffering was used to create the effect of smooth scrolling through large trees.

Clipping

Browser uses OpenGl library for drawing the graphical tree representation of object databases. Every OpenGl function call is associated with a certain amount of overhead. If every node of the tree was to be drawn for a reasonably sized tree, such overhead could

easily accumulate into a few seconds. In order to keep browser's interactivity at a reasonable level, as many OpenGl function calls had to be eliminated as possible. To this end, a customized clipping mechanism has been implemented, where only parts of the tree that are potentially visible are drawn. The rendering of trees in browser is performed recursively, using the following pseudo-code:

```
void render( NODE * node):
     render_folder( node)
     render_name( node)
     render_icon( node)
     for all children of node:
          render_line( node, child)
     for all children of node:
          if( ! out_of_view( child))
               render( child)
```

First, the folder, name and the optional icon for a node is rendered. The next step involves drawing a line between the node and each of its children. At last the children of the node are rendered recursively. However, before a child is rendered, the bounding box of the subtree starting at that child is checked for intersections with the viewing area. If the bounding box of the subtree does not intersect the viewing area, then no part of this subtree will be visible to the user and the entire subtree is skipped (not rendered). Figure 6-7 demonstrates clipping of trees on a sample tree: the shaded objects represent nodes that are rendered (i.e. `render()` is called upon these nodes), while the empty objects depict nodes which are not rendered.

Support for multiple instances of the browser

For certain operations browser requires unique access to a database. If the database is modified by another application while browser is performing such an action, some of the information could be inadvertently lost. As an example, consider the action of *copying* a subtree of objects. If the subtree is sufficiently large, such an operation could take several seconds to execute. The copy operation is performed by archiving the contents of the subtree into an external file, as described in the previous section. If another instance of browser attempted to change the same subtree while the first browser was creating the archive file, some of these changes could be reflected in the archive file. This would result in inconsistencies between the contents of the source and destination subtrees.

To prevent corruption of databases resulting from simultaneous access, browsers synchronize themselves by sending each other messages through VLAB daemon. Whenever one instance of browser needs unique access to a database, it broadcasts this request too all other browsers. Upon receiving such a message, all other browsers suspend

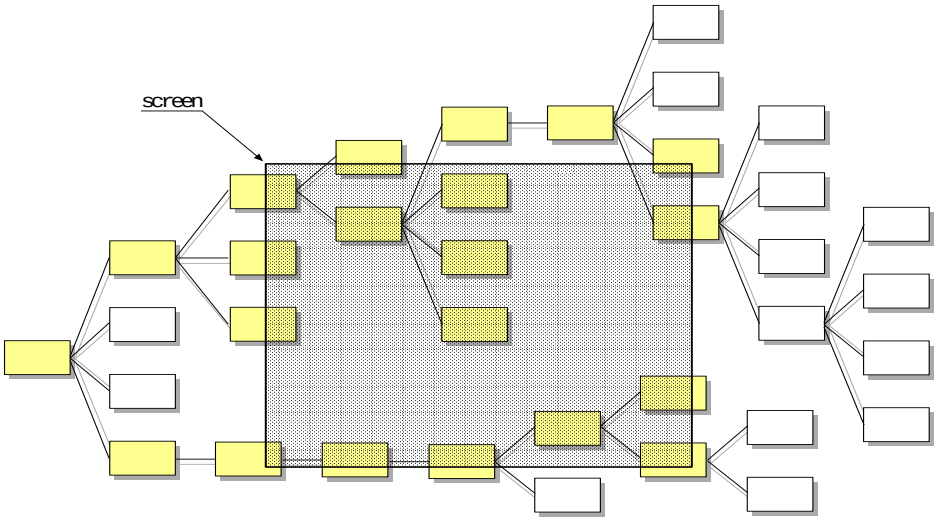


FIGURE 6-7: *Tree clipping in browser*

their user interfaces, so the user cannot invoke any actions. When the browser with unique database access completes its operation, it issues the second broadcast to notify all other browsers that the database is unlocked. At this point, all browsers resume their normal mode of operation. It should be pointed out that this mechanism only works among browsers run by the same user on the same workstation. No mechanism has been implemented for locking access to databases for remote users. Also, it is possible that race condition occurs among browser instances run by the same user.

Communication between instances of browser is also used to notify other browsers about updates to databases. Each time a database is modified, the responsible browser broadcasts a message identifying the affected database and includes details about the operation it performed. This ensures all other browsers can update the displayed information to reflect the real contents of the database. However, if a database is modified without the use of VLAB tools (e.g. through a UNIX shell), such changes will not be automatically reflected in browser.

Speed improvements

Browser 3.0 is implemented entirely in C++. It uses Motif library for its graphical user interface, and OpenGl library for rendering graphical representations databases. This combination of a fast language and efficient libraries delivers the performance needed for browsing in practice.

The most evident speed bottleneck in browser 2.0 is the mechanism it implements to read in and display the thumbnail icons for objects. The icon files stored with objects are in IRIS RGB format. Since browser 2.0 does not understand this format, it uses an external utility to convert the icon file to a GIF file. Now that the format of the image is understood by browser, it is read in and displayed. The process of invoking external programs within other programs is associated with large amounts of overhead, since it usually involves spawning a new UNIX process and loading a new program. Showing icons for a reasonably sized subtree of objects in browser 2.0 can easily take several minutes to complete.

Browser 3.0 implements a native support for IRIS RGB files. To speed up the drawing process, it also adds support for scaling of icons. The routines for reading IRIS RGB pictures and scaling of images have been acquired from the sources of the XV program and adequately modified†.

Implementation of cut, copy and paste

The cut, copy and paste operations have been implemented in browser 3.0 as follows. When a copy or cut operation is applied to an object, its contents are stored in a temporary location (and in the case of cut, the object is also removed from the database). At the time of paste, the object stored in the temporary location is copied into its new location as an extension of the selected object.

Objects are stored in the temporary location as archive files. A customized archiving library has been developed to this end, which has very similar functionality to the standard UNIX utility tar . The reason for developing a new archiving library instead of using tar is two-fold. First, for performance reasons it is desirable to avoid using external utilities whenever possible. Second, tar does not archive symbolic links in a suitable way for use with oofs databases. In order to properly archive a subtree of VLAB objects, all relative symbolic links of the top level object have to be expanded, while all other objects in the subtree have to be archived as symbolic links.

The archived subtree is stored in a file determined by evaluating the expression:

```
${VLABTMPDIR}/cutCopyPaste<userid>/data.ar
```

The name of the top level object in the above file is stored in a separate file given by the expression:

† with the permission from John Bradley, author of XV

`${VLABTMPDIR}/cutCopyPaste<`*`userid`*`>/---FILENAME---`

The reason for storing the name of the root object in a separate file is to be able to quickly determine whether an object can be pasted as an extension to the selected object. An object subtree from the archive cannot be pasted if the selected object already has an extension with the same name as the archived object.

The oofs format would be an alternative way to store temporary copies of objects. An archiving library would not have to be developed, as a simple recursive algorithm could be used for such an implementation. However, the advantage of using an archive file for storing temporary copies of files can be seen in the performance of cut, copy and paste operations. The cut and copy operations need only create two files: `data.ar` and `---FILENAME---`, independent of the size of the object subtree being copied. This saves browser a lot of the overhead associated with creation of files. Similarly, the paste operation only needs to read two files, saving browser many file opening operations.

Another benefit of this implementation is the possibility of speeding up the cut, copy and paste operations when working with remote databases. RAserver could be extended with high-level operations allowing the transfer of entire subtrees of objects as archives, as opposed to the current implementation where individual files are being sent over the network.

6.4. Summary

This chapter described the design and implementation of the new version of browser - a VLAB tool assisting users in navigation and management of object databases. An object lookup table is maintained with every oofs database - to support external references to VLAB objects. Browser also takes advantage of the new remote access extension and allows users to browse and modify remote object databases.

CHAPTER 7 *Metatext*

VLAB users often need to access objects in an order different from the hierarchical organization of the databases in which they reside. For example, a user may want to present the results of his work - a number of different models of plants, each developed and located in its own object subtree. Metatext is a VLAB utility which allows access to VLAB objects in an arbitrary order, independent of the hierarchical organization of the databases [13][29]. Using metatext, it is possible to create many different views of the same object database.

This chapter describes my implementation of metatext 3.0, based on the functionality provided by the previous version 2.0. A small sub-section (Section 7.2) describing metatext from the user's point of view is included. Eventhough metatext 3.0 is only a re-implementation of version 2.0 using C++ and the Motif library, its description is included here as the basis for the discussion of hyperbrowser (Chapter 8).

7.1. Structure of metatext

A metatext database, as shown in Figure 7-1a, consists of two types of files: frames and index files. A frame is the basic information unit in metatext. It contains links to other metatext nodes as well as links to objects in oofs databases. A frame also includes a textual description and a list of commands. This text is displayed and commands executed when the frame is invoked.

An index file is a list of names grouping related frames into a section. A frame can be listed in several index files and an index file can be included in several frames. This allows metatext users to build graphs of any shape, not restricted to tree structures as oofs databases.

Metatext runs as a number of concurrent processes (Figure 7-1b), which can be divided into four classes:

- *metatext* processes, which read index files and allow the user to interac-

tively select a frame,
- *text display* processes, which display the frame text,
- *object manager* processes corresponding to the links to objects listed in the frame,
- *application* processes, corresponding to the commands in the frame.

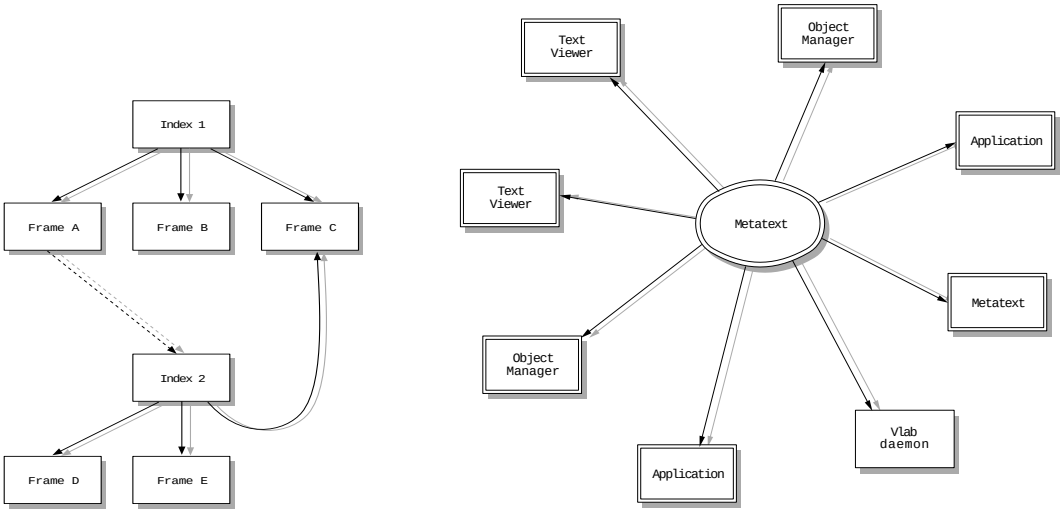


FIGURE 7-1: *Structure of a) metatext database, b) metatext processes*

Metatext processes are created in the same way as any application process, by executing the commands as extracted from frames. Many instances of metatext can be running at the same time, offering the user alternative views of the database.

7.2. User's perspective of metatext

A running metatext process is manifested by a small window (Figure 7-2) with 4 buttons and the name of the current index file. The right most button is a down arrow, which is used to create an extended window (Figure 7-2) with the metatext menu. The extended menu may also be displayed as a popup style menu, using the right mouse button. This menu includes an item for each frame in the section, plus several additional items. The other three toggle buttons turn *on* or *off* operations invoked automatically when a frame is selected, specifically:

FIGURE 7-2: *Snapshot of metatext without (top) and with (bottom) expanded menus*

- ***Exec*** - turns on or off execution of UNIX commands extracted from the frame,
- ***View*** - turns on or off display of text extracted from the frame,
- ***Edit*** - turns on or off automatic invocation of a text-editor in which the frame can be edited.

Two additional buttons are included in the extended window:

- ***Next*** - selects the next item on the menu. Allows stepping through the frames in forward order. The button associated with the last invoked frame has a distinct color in the extended window, giving the user a visual clue as to what frame will be invoked next.
- ***Previous*** - selects the previous item on the menu, allowing stepping through the frames in reverse order.

Metatext menu

The metatext menu contains the list of frames as specified in the index file, as well as four additional menus:

- ***Modify*** - gives the user a shell window with the current working directory set to the one of the index file, so that the user can modify the index/frame files with his favorite editor,
- ***Reread*** - rereads the index file and updates the menu,
- ***Restart*** - restarts the metatext process,
- ***Quit*** - quits the metatext process.

7.2.1. Start-up information

Metatext is invoked from the command line using the following syntax:

```
metatext [-e] [-exec] [-view] [-edit]
         [-md|-mc|-mt]
         [-rootdir oofs] index_file
```

The following settings determine the state of metatext when invoked:

- **-e** metatext is open with the extended menus
- **-exec** set the executable toggle
- **-view** set the view toggle
- **-edit** set the edit toggle
- **index_file** the name of the index file

In addition, the following settings are provided for backwards compatibility:

- **-md** demo mode (equivalent to -exec -view)
- **-mc** command mode (equivalent to -exec)
- **-mt** text mode (equivalent to -view)
- **-rootdir** specifies the location of the default oofs database. When an object manager is invoked, it needs to know where the root of the oofs database resides, so that the `.dbase` can be updated if needed. In the previous version of VLAB, .dbase file did not exist and therefore old frames contain links to objects which do not provide the root directory of their oofs databases. If such a link is found, object manager is invoked with root directory set to the value specified with this command line option.

7.3. Implementation Details

7.3.1. Index file format

An index file is a text file that contains the names of some or all frames in the metatext database. Each name must be listed on a separate line; these frame names are then included in the metatext menu. For example, the following index file Demo resulted in the menu shown in Figure 7-2:

```
2D-plants
barentree
lychnis-demo
random-moss
cypress
```

7.3.2. Frame file format

Frame files contain a mixture of text and UNIX commands. When the user selects one of the menu items read from the index file, the text portion of the corresponding frame file is displayed in a window and the included UNIX commands are executed. Syntactically, every line that does not start with a ':' is considered part of the text and every line that starts with a ':' is assumed to include a UNIX command. For example, the frame file 2D-plants could contain the following information:

```
Understanding A Formalism
2D Plants
:xclock
:object /usr/u/vlab/oofs/ext/plants/ext/2D-plants
```

If the user now selects the ***2D-plants*** item in the menu shown in Figure 7-2, then a window including the descriptive text associated with the object will be displayed, and the UNIX commands embedded in the frame file will be executed. The resulting display is shown in Figure 7-3. Window 2D-plants shows the object '`2D-plants`' as a result of the last command '`:object`'. The xclock window is a consequence of the '`:xclock`' command. The xless window displays the text from the `2D-plants` frame file.

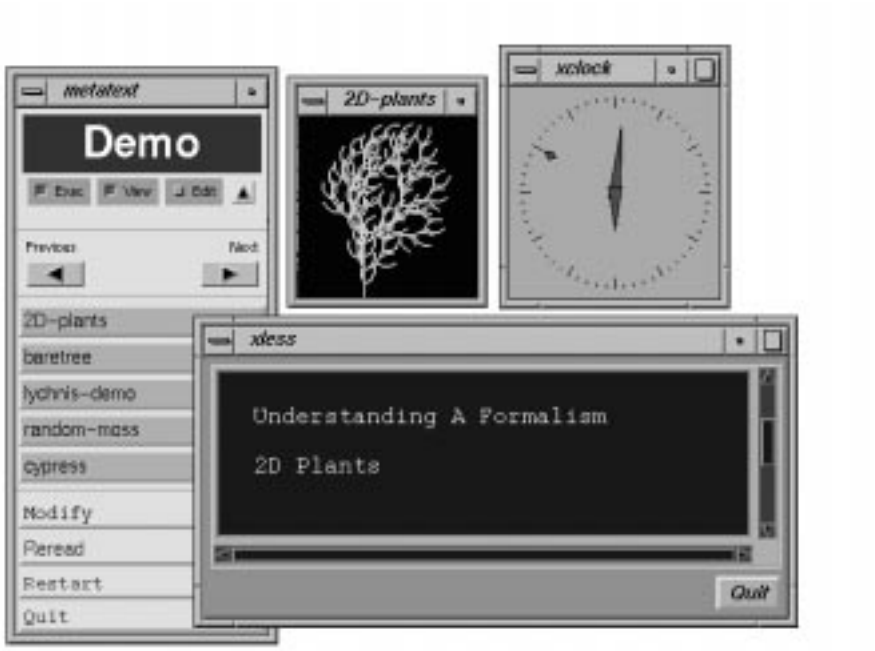


FIGURE 7-3: *Example of metatext display*

It should be noted that all commands which invoke object manager (commands starting with `:object`) are treated as special cases in metatext 3.0. In the old version of metatext such commands were simply executed as any other commands extracted from the frames. Since, in VLAB 3.0, it is VLAB daemon's responsibility to invoke object manager, metatext must send a message to VLAB daemon whenever such a command is found. Furthermore, metatext also sends a message to all currently running browsers† whenever object manager is invoked. Browsers respond to this message by updating the displayed database tree, so that the invoked object is selected and centered in the browsers' windows. This allows the metatext user to see automatically where the demonstrated object is located in the object database.

† this is done by sending the message to VLAB daemon, which re-distributes it to all running browsers

7.3.3. Organization of metatext databases

Metatext does not enforce any restrictions on the organization of nodes and frames. However, it is common to organize related index frame files into sub-directories, and these sub-directories into tree hierarchies. Cyclic references between metatext nodes are also possible.

7.3.4. Customization

Users may specify which external applications to use for performing editing and viewing functions by editing the ${VLABCONFIGDIR}`/metatext` file. For example, the file could contain the following information:

```
Editor xwsh -e vi
Viewer xless -auto
```

The first line tells metatext how to invoke an external text-editor on the frame file. Metatext invokes a text-editor on a frame file when the frame is selected from the list of frames while the ***Edit*** toggle is set. The second line specifies the external application to be used to view the textual contents of frames.

7.4. Summary

Metatext is a VLAB utility which allows access to VLAB objects in an arbitrary order, independent of the hierarchical organization of the databases. Using metatext, it is possible to create many different views of the same object database. This chapter described my re-implementation of metatext, based on the functionality provided by the previous version. The main difference between the new and the old version is the use of the Motif library for the graphical user interface in my implementation.

CHAPTER 8 *Hyperbrowser*

An object in the database may be of interest in several contexts. For instance, a model originally developed as a part of a comparative study of lilac inflorescences for horticultural purposes may also serve as an example of a particular branching architecture, an illustration of model construction according to field data, an instructional example of programming using L-systems, a realistic model available for incorporation in complex scenes, and the source of an image for a paper. To facilitate the presentation of a VLAB object in several different contexts, a new mechanism for maintaining alternative views of oofs databases was designed and implemented. This chapter describes its design and implementation.

8.1. Requirements and design

In this section I describe the design for VLAB's new support for alternate views of databases. First I look at the shortcomings of the hypertext functionality offered by metatext in VLAB 2.0. Then I describe the requirements for the new system and two potential implementation methods. Lastly, I detail the overall design of the current implementation.

8.1.1. Shortcomings of metatext

Basic functionality for creating alternative views of object databases is provided in VLAB 1.0 and 2.0 by metatext. Unfortunately, metatext imposed significant limitations on the user, which narrowed its usefulness as a tool for maintaining alternative views of oofs databases. The most visible drawback of metatext is that it does not provide a clear visual representation of the organization of metatext databases, and it does not offer any support for the management of metatext databases. The user has to organize metatext databases manually, from the UNIX shell prompt, and a text editor has to be used to create and update links to VLAB objects. Another important drawback which metatext users are exposed to is the issue of dangling links. Whenever an object in the oofs database is moved or renamed, all metatext frames referring to this object become out of date and

hence unusable. These frames have to be laboriously found and fixed. Another, minor limitation of metatext, is related to the fact that each representation of metatext nodes is displayed on the screen in a separate window. This often becomes a problem when browsing large databases of metatext nodes, as the screen becomes cluttered with a multitude of windows.

8.1.2. Design goals

A new mechanism for supporting alternative views of oofs databases in VLAB was developed. The design goals for such a system were mostly derived based on the limitations of metatext, as identified in the previous section:

- The new system should implement a well organized structure for storing hyperlinks. It should also provide an intuitive interface for the management of such databases.
- Hyperlinks in the database should be automatically updated whenever the objects they refer to change their locations.
- The new system should provide a convenient visual representation of hyperlink databases.

8.1.3. Implementation models

Two models for representing alternate views of object databases have been considered. Under the first model, the alternate views would be represented as hypertext documents, where textual information is mixed with hyperlinks in a single flow. The hyperlinks would represent pointers to other hypertext documents, or pointers to the actual objects in oofs databases. This model is similar to the design used in HTML documents. A simplistic mechanism for invoking VLAB objects from HTML pages has been successfully developed, verifying that an implementation of this model is possible.

Using the second approach, hyperlinks would be organized in a hierarchical fashion. In this model, each hyperlink can be associated with a textual description which contains information pertinent to the linked object in its current context. This design is compatible with the organization of objects in oofs databases.

Since both models could be developed to satisfy the design goals stated in Section 8.1.2, I have chosen the second approach, the easier one to implement. A similar hierarchical

model has already been implemented in VLAB for storing objects in oofs databases and has proven successful.

8.1.4. Hyperobjects

To accommodate the information needed to be stored with each hyperlink, a new VLAB entity, called hyperobject, has been designed. The following information is stored with each hyperobject:

- *hyperlink* to an object in an oofs database. Not all hyperobjects need to point to an object in an oofs database. Those that do not contain a hyperlink are used as place-holders for other hyperobjects.
- *textual description* of the hyperobject. This description is usually related to the object the hyperlink points to under the given context.
- *name* of the hyperobject. If a hyperobject does not specify its name, the name of the object it points to is used instead. If the object later changes name (e.g. is renamed in the oofs database), the hyperobject automatically changes its name as well.
- *children*. Each hyperobject can contain other hyperobjects, forming a tree structure, as described in the following section.
- *order* of children. The order of children in a parent can be changed, which affects the traversal of hyperobject databases.

Hyperobjects take advantage of VLAB's new support for external references to objects, described in Section 6.1.1 The hyperlink in a hyperobject is specified as the `ID` of the object to which the hyperobject points, allowing hyperlinks to stay intact as long as the objects to which they point remain in the oofs database. Quick access to objects from hyperobjects is made possible by using the object lookup table.

8.1.5. Hyperobject file system

Hyperobjects are organized hierarchically into an hyperobject file systems (hofs). At the implementation level, the structure of hofs is similar to the structure of oofs - it is a hierarchy of UNIX directories and files forming a directory tree. It is possible to think of hyperobjects as symbolic links in a UNIX file system, and of hofs databases as directory hierarchies entirely composed of symbolic links associated with textual descriptions.

However, there is a very important conceptual difference between oofs and hofs databases. Hofs databases are not based on the prototype-extension mechanism. The reason for this design decision is simple - the prototype-extension mechanism is not needed for hyperobject databases. The prototype-extension model is advantageous when extensions share data with their prototypes, but this is uncommon for hyperobjects.

The hierarchical organization of hyperobjects allows VLAB users to create and describe various conceptual relations between objects.

8.1.6. Hyperbrowser

Hyperbrowser is a VLAB program designed and implemented to assist users in creating, modifying, and navigating through hyperobject databases. The rest of this chapter describes hyperbrowser. The user's perspective of hyperbrowser is presented, and its implementation details follow.

8.2. User's perspective of hyperbrowser

Both hyperbrowser and browser operate on databases which store information using the same file structure. To preserve consistency between VLAB applications, hyperbrowser offers the same interface for database manipulating functions as browser. To avoid unnecessary repeating, I will only describe the features of hyperbrowser that are different from those of browser.

8.2.1. Overview

Hyperbrowser is manifested on the screen as a window displaying a hierarchy of hyperobjects (Figure 8-1). Notice the evident similarity between browser's and hyperbrowser's appearances. To avoid confusion between browser's and hyperbrowser's windows, the user may customize hyperbrowser differently than browser, for example, by changing its background color and the icon size.

The title of the window indicates the location of the hypertext database displayed in hyperbrowser. The location is given in a format *host:path*, where the host is the name of the computer on which the database resides, and path indicates the root level object in the database. The bottom part of the window contains linking information about the selected

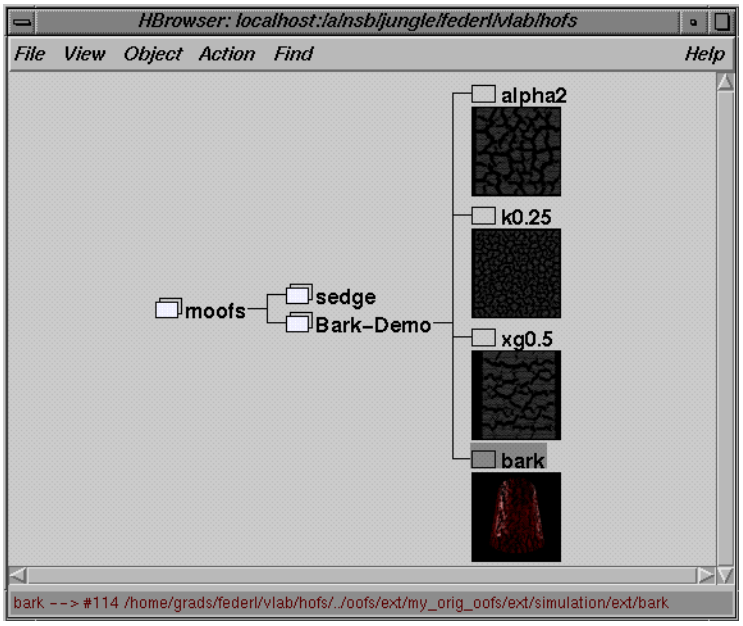


FIGURE 8-1: *Example snapshot of hyperbrowser's window*

object, i.e. the name of the associated object in the oofs database. If the hyperobject is not associated with any object in oofs, this field is empty. The middle part of hyperbrowser's window graphically depicts the hypertext database. Similar to browser, each hyperobject is represented by a folder symbol, object name and an optional icon. Parents and children are connected by lines forming a tree structure. Hyperobjects with folder symbols that are filled have children, while objects with single folder symbols represent leaves of the tree. The graphical tree is automatically modified whenever a change is introduced to the hypertext database. Scrollbars can be used to shift the displayed tree to display hidden parts of the database.

8.2.2. Start-up information

The syntax used to invoke hyperbrowser is identical to the syntax for invoking browser:

```
hbrowser [-p password] [[[login@]hostname:]dirname]
```

The meaning of the command line parameters has been described in Section 6.2.

8.2.3. Invoking hyperobjects

The most obvious difference between interfaces of browser and hyperbrowser is the additional pull-down menu ***Action*** in hyperbrowser's menu bar (Figure 8-2). Functions available from this pull-down menu are related to the traversal of hypertext databases and the way hyperobjects are invoked.

FIGURE 8-2: *Action menu in hyperbrowser*

Hyperbrowser allows its users to systematically traverse the hierarchies of hyperobject databases. It does so through the use of ***Next*** and ***Previous*** buttons. When the user invokes the ***Next*** button, hyperbrowser automatically highlights the next† hyperobject in the hofs hierarchy, and invokes this hyperobject. Invoking a hyperobject in hyperbrowser results in performing a combination of the three actions described below. This combination of actions is determined by the state of the toggle buttons at the bottom of the ***Action*** pull-down menu. The algorithm for selecting the next hyperobject is depth-first traversal. Figure 8-3 shows an example of an hofs tree and the order in which the hyperobjects in this tree would be selected if the ***Next*** button was used in succession. The ***Previous*** button would access the hyperobjects in the reverse order.

The ***Get object*** function is used to invoke a panel manager on the object associated with the selected hyperobject. When the selected hyperobject is not associated with any object in an oofs database, this button is grayed out and not available to the user. The user can view the textual description associated with the selected hyperobject using the ***Show text*** button. When the selected hyperobject does not contain a description, this button is grayed out and not available to the user. The ***Edit text*** button is used to create a new description, or to modify an existing description of the selected hyperobject. When invoked on a selected hyperobject, hyperbrowser spawns an external text editor on the description file.

There are three toggle buttons at the bottom of the ***Action*** pull-down menu, specifying what are the default actions performed on a hyperobject when it is invoked. Hyperobjects can be invoked either by double-clicking on their folder icon, or by using the ***Next*** and ***Previous*** buttons. ***Get object*** toggle specifies whether object manager is invoked on the object associated with the selected hyperobject. ***Show text*** and ***Edit text*** toggles determine how the textual description associated with the selected hyperobject will be displayed (it

† If the next hyperobject is hidden, the currently selected object is automatically expanded by one level.

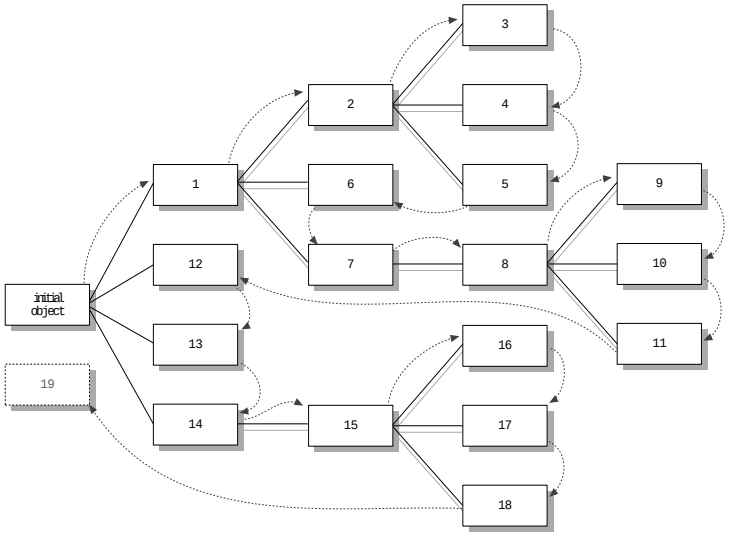


FIGURE 8-3: *The order of database traversal using the Next and Previous functions in hyperbrowser*

can be: not shown at all, displayed in a text dialog window, or displayed in a text editor where it can be edited).

8.2.4. Changing the order of hyperobjects

A method for changing the order of hyperobjects is needed for hyperobject databases, because this order determines their traversal. Hyperbrowser allows the users to change the order of children of a hyperobject using the keyboard. To change the position of a selected child, the ***UP*** and ***DOWN*** arrow keys are used. The UP arrow key is used to move the selected child one position upwards, and the DOWN key is used to move the selected child one position downwards.

8.2.5. Invalid hyperobjects

To indicate invalid hyperobjects in hofs databases, hyperbrowser displays three question marks in the name field of invalid hyperobjects. Invalid hyperobjects are hyperobjects which contain pointers to nonexisting objects. Invalid hyperobjects result as a

consequence of deleting information from oofs databases. Unfortunately, there is currently no mechanism implemented in browser to warn the user when an operation on an oofs database results in making some of the hyperobjects invalid.

8.2.6. Renaming hyperobjects

A hyperobject can be renamed in hyperbrowser using the same interface as browser uses to rename objects. However, the user can specify an empty name for a hyperobject, indicating that the hyperobject should inherit the name from an object it points to.

8.2.7. Adding hyperobjects to hofs databases

The user can create a new hyperobject in an hofs database in one of the following ways:

- by copying a (single) object from browser, and pasting it into a hyper-browser; or
- by dragging an object from browser into hyperbrowser.

Hyperbrowser will respond to both of these actions by creating a new hyperobject in the selected destination, with a hyperlink pointing to the source object. No textual information will be associated with this new hyperobject, and the name of the newly created hyperobject will be empty (i.e. inherited from the source object). A description can be then added to the hyperobject by invoking the ***Edit text*** function from the ***Action*** pull-down menu.

It is also possible to transfer a hyperobject from hyperbrowser into browser, using either the copy and paste, or the drag and drop operations. Such transfer results in creating an extension of the destination object, where the extension is the object linked with the source hyperobject. This operation is therefore identical to copying the object to which the source hyperobject points to using a browser, and pasting it into an oofs database. Notice that hyperobject which do not contain hyperlinks cannot be transferred to oofs databases.

8.3. Implementation details

8.3.1. Structure of hyperobjects

Similar to VLAB objects, hyperobjects are represented as directories. All information pertinent to a hyperobject (as described in Section 8.1.4) is stored in the hyperobject's directory. The textual description associated with the hyperobject is stored in a file `text`. This file can be missing, indicating no description has been provided with the hyperobject. The children of a hyperobject are stored in the `ext` subdirectory. The rest of the information is stored in a file `node`.

8.3.2. Format of the node file

The `node` file, present in every hyperobject, contains three components: hyperlink to an object, name of the hyperobject, and order of children. The format of the `node` file is:

```
ID
name
number-of-children
child1
child2
.
.
.
```

The `ID` is an integer, specifying the object in the oofs database to which the hyperobject points. If the ID contains a value of `-1`, the hyperobject does not point to any objects. This is useful for creating a parent for a number of children, without associating the parent with any object from the oofs database. It should be noted that all hyperobjects in an hofs database can point to object located within only one oofs database. The oofs database associated with an hofs database is assumed to reside in the same directory as the root-level hyperobject of the hofs database.

The `name` field in the `node` file determines what textual name will hyperbrowser render in its graphical view of the object. If this field is empty, the name of the object in the oofs database will be used. If it is non-empty, the contents of the field will be used, no matter what the corresponding object in the oofs database is called. For an `ID` of `-1` the hyperobject always has to have a non-empty `name` field.

After the `ID` and `name` fields, a list of children follows. The order of children in this list determines in which order the children are rendered by hyperbrowser. The order of children is important when traversing an hofs database systematically, using the ***Next*** and ***Previous*** functions.

8.3.3. Implementation of hyperbrowser

Most of the functionality offered by hyperbrowser is related to the management of hofs databases. Since the organizational structure of hofs and oofs databases is so similar and browser provides all database management functionality on oofs databases, hyperbrowser was implemented by adopting a majority of browser's code. Some adjustments were needed to convert browser's code into hyperbrowser.

Names and icons

Hyperbrowser displays textual names for hyperobjects based on the contents of the `name` field in their `node` files, as described in Section 8.3.2. The icons which hyperbrowser displays with hyperobjects are loaded from their linked object's directories in the oofs database. If a hyperobject is not associated with any object, a default icon for the entire hofs database is used. This default icon is stored in the root level object, called `hofs`.

Ordering of children

Hyperbrowser determines the order in which the children of a hyperobject are displayed based on the order given in its `node` file, whereas browser displays them in alphabetical order. Hyperbrowser also implements a mechanism for modifying the order of children, accomplished by using the keyboard's arrow keys.

Rename

The implementation of the rename operation in hyperbrowser is quite different from that of browser's. When a hyperobject is renamed, only the `name` field in its `node` file is affected, while the name of the hyperobject's directory remains unchanged. Hyperbrowser allows the user to specify empty names for hyperobjects, indicating that the name to be displayed for the hyperobject should be obtained from its linked object). Also, the specified name for a hyperobject does not have to be unique among its siblings.

Paste and drop operations

In order to allow transfer of objects into hofs databases, hyperbrowser has to distinguish whether the data being pasted is an object or a hyperobject. If the data represents a hyperobject, a new copy of the source hyperobject is simply created and added as a child to the destination hyperobject. However, if the data represents an object (originated in browser) a new hyperobject has to be created, with a hyperlink to the source object.

The implementation of the paste and drop operations in browser also had to be extended, to allow transfer of hyperobjects from hyperbrowser to browser. When a hyperobject is transferred to browser, it is first asserted that the hyperobject contains a valid hyperlink. If this is not the case, the user is properly notified and the transfer is aborted. If the hyperobject contains a valid link, the object associated with this link is then used to finish the paste (or drop) operation.

8.4. Summary

A new mechanism that allows the creation and maintenance of alternate views of object databases has been designed and implemented. A new system for hierarchical organization of hyperlinks was designed. Hyperlinks point to objects in oofs databases by using the unique IDs stored with each object. A new VLAB application, hyperbrowser was developed, allowing VLAB users to visually manage and navigate hyperobject databases.

CHAPTER 9 *Conclusion and future work*

9.1. Conclusion

9.1.1. Accomplishments

A new mechanism that allows the creation and maintainance of alternate views of object databases has been designed and implemented. First, support for maintaining external references to VLAB objects has been implemented, by assigning unique IDs to objects. Secondly, a new system for hierarchical organization of hyperlinks was designed. Finally, a new VLAB application, hyperbrowser was developed, allowing VLAB users to visually manage and navigate hyperobject databases.

The remote access extension to VLAB has been designed and implemented to improve VLAB's support for collaboration. This extension allows users of VLAB to transparently access remote databases, making it easy to invoke and interchange objects among collaborators. A new VLAB tool, RAserver, has been developed. RAserver is a daemon running on a remote computer, performing actions on the remote database on request by other VLAB tools accessing the database. RAserver maintains a list of accounts with encrypted passwords and access levels, thus preventing unauthorized access to remote databases. Browser, hyperbrowser and Object Manager are capable of accessing remote databases using the services provided by RAserver. RAlibrary was designed and implemented to aid programmers in the development of applications that require RAserver's services.

The design of panel manager has been improved and a new version implemented. Through its new GUI builder facilities panel manager now allows users to create and modify panels visually. The new panel manager also supports two-way communication, thus removing the inconsistency between the information displayed in the panel and the information contained in the data files. Panel manager 3.0 uses the Motif library for its user interface, making its look and feel consistent with the rest of the VLAB components.

Performance related limitations associated with the earlier version of Browser have been eliminated, allowing it users to fully explore its capabilities. Speed improvements were mainly achieved by re-implementing Browser in C++, by using faster GUI libraries, and also by reducing its dependence on external utilities.

Various customization mechanisms were implemented. The users are allowed to change the visual appearances of most VLAB applications.

VLAB's portability was improved to make it accessible to a wider range of users. Its portability was successfully tested by porting it to three different UNIX operating systems on four architectures. A project is presently under way which will make VLAB available on personal computers.

9.1.2. Impact of VLAB 3.0

VLAB 3.0 has been installed in many places and with great success used to support scientific research for over one year. The new support for alternate views of object databases and for customizing appearances of VLAB applications has been successfully used to give live presentations. Hyperbrowser allows scientists to conveniently design the order of their demonstrations and to add descriptions to the presented objects. During the actual presentations, hyperbrowser is used to systematically invoke the prepared objects together with their descriptions.

Older versions of VLAB have been successfully used to support individual research of many users around the world. With the new support in version 3.0 for accessing remote databases, many researchers use VLAB for collaboration purposes. For instance, VLAB is currently being used for collaborative work between users in Calgary and Australia.

Thanks to the improved portability of all VLAB components, version 3.0 has been compiled and installed in France on a previously untested platform - DEC Alpha running Linux 2.0. Also, unmodified sources of VLAB 3.0 successfully compile and execute on a new version of the IRIX operating system (6.2).

9.2. Limitations and future work

In this section I discuss ideas to be considered for future work. Most of these ideas stem directly from the limitations of VLAB 3.0 and from the lessons I learned during its

implementation. Some of these ideas introduce new concepts, while others would contribute to VLAB by merely increasing its functionality.

9.2.1. Find

The find operation in the current implementation of browser does not take advantage of the object lookup tables available in every oofs database. It uses a recursive algorithm to traverse object trees to search for a match, and is therefore inefficient. A more efficient solution should be implemented, where the search would be performed by examining the contents of the `.dbase` file.

9.2.2. Improved GUI designer for panel manager

A number of improvements are needed to improve the support for visual building of control panels in panel manager 3.0. Support for changing attributes for multiple components is desired. For example, it is tedious to change the background of all components in a control panel to the same color, as this can only be done one component at a time. The user should be able to select multiple components and then invoke an attribute editor, where common attributes for all selected components could be modified.

Panel manager also needs a better selection of components. For example, the choice control is not well suited for long lists of selections. A combo-box or a scrolled list would be more appropriate for this purpose. Also, group components are desired, so components can be organized and manipulated in groups. Text box would be another useful component, which could allow the users to edit blocks of text using the keyboard.

9.2.3. Undo

One of the most important features which unfortunately all VLAB applications lack is an undo function. The undo functionality is mostly needed in Browser and hyperbrowser for operations involving database management, as it could revert accidental, or no longer wanted, modifications.

The need for an undo mechanism has been identified shortly after the release of the first version of VLAB. Unfortunately, no acceptable solution has been found to this date. Undoing changes made to VLAB oofs and hofs databases is equivalent to undoing

changes made to a UNIX filesystem, because VLAB databases are stored as a hierarchy of directories and files. Most flavors of UNIX operating system do not offer any means for reverting modifications of filesystems. Without support from the operating system, programs implementing undo would have to record all changes made to the filesystem, so that they could be undone. The mechanism for recording such changes has to be efficient if the performance of VLAB is not to suffer.

The implementation of an undo mechanism is complicated even in a single-user environment. The difficulties arise, for example, when one application cannot complete an undo request because the database has been further modified by another application. For instance, consider a scenario where the user is running two copies of Browser on the same oofs database. The user deletes an extension of a prototype using the first Browser, followed by moving the prototype to another location using the second Browser. Undoing the last operation in the first Browser involves recreation of the removed tree. However, this is now impossible, as the prototype was moved to another location.

The issues concerning the implementation of undo functionality increase in complexity when a multi-user environment is considered. Now one has to contemplate the possibility of other users (possibly connected remotely) modifying the same database and thus impeding the execution of an undo operation. A simple solution (from the implementation point of view) is to report to the user whenever undo cannot be completed. However, such solution would give the user a false sense of security, because the undo command could fail to work in some cases.

9.2.4. Extended objects

For many classes of experiments, such as development of source code, it is natural to organize information in sub-directories. VLAB's extension which would allow sub-directories to be stored within objects, is therefore imperative if VLAB is to be successfully used for the management of experiments of this kind. Although it is possible to manually store subdirectories with objects in oofs database, all VLAB tools assume that objects only contain regular files. Consequently, the information stored in subdirectories could get lost if the object was manipulated by any of the VLAB tools.

Many of the tasks invoked by users at the beginning and at the end of experimentation are routine. For example, it is common to store large text data files in a compressed form to save space. Every time an object containing such compressed data is invoked, the data needs to be un-compressed before it can be used. Examples of routine tasks performed when the user is finished with an experiment include: compression of large files, clean-up of temporary data files, etc. VLAB should allow each object to be associated with user definable actions, executed when an object is invoked and closed.

9.2.5. Extended hyperobjects

The current design of VLAB's hypertext system has an important limitation: each hofs database is associated exactly with one oofs database, and all hyperobjects in this hofs database must point to objects its associated oofs database. It is impossible to have two hyperobjects in the same hofs database pointing to objects in two different oofs databases. This means, in particular, that users are not allowed to create hyper-links to other user's objects, and that they are not allowed to copy hyperobjects between two different hofs databases. The current design of hypertext system needs to be extended so that hyperobjects can point to objects in any oofs databases.

9.2.6. Unified oofs and hofs databases

The only difference between objects and hyperobjects is that objects contain real information and hyperobject contain pointers to other objects. There is no need to manage these entities in two separate databases. The current design can be extended by unifying oofs and hofs databases into a single database. In this new database an object would either represent a real information, or contain a hyperlink to another object in the database. This would eliminate the need for two separate VLAB applications - a browser for oofs databases and a browser for hofs databases. Their functionality could be combined into a single application. Several issues would have to be, however, addressed in such design. There is no equivalent of object manager for hyperobjects - how can a user manipulate the internals of a hyperobject? Could hyperobjects become prototypes for objects?

9.2.7. Unique access to databases

It is a known fact in computer science that simultaneous modifications of a shared resource may result in an unexpected corruption of information. The present implementation of VLAB does not prevent concurrent access to databases, and therefore does not protect its users against such damage. A possible solution is to 'lock' the entire database whenever an application is accessing the database, so that all other applications have to wait. Such locking could be implemented for example using file locks [35]. The disadvantage of this simple implementation is that in some cases the performance of other concurrently running applications unnecessarily suffers. Some operations can be performed on a database concurrently, without the fear of damaging any information (such as invoking two objects, or performing two different searches). A mechanism is needed which will guarantee unique access to databases but will affect the performance of VLAB applications to a minimum.

9.2.8. Multiple inheritance

The inheritance mechanism adopted by VLAB allows only one prototype per object. It is impossible to create an extension which would inherit from more than one object simultaneously. This limitation of VLAB has been identified in its early design stages [21], however no satisfactory mechanism for implementing multiple inheritance has been found to this date. Having the possibility to create extensions inheriting from a number of prototypes would be appreciated in many areas of computer based experimentation. For example, in biological modeling of plants multiple inheritance could be used to create gardens, where the prototypes would represent individual plants and the extension the complete scene. In source code management domain, the prototypes could represent various libraries, while the extension could represent a program that uses these libraries.

9.2.9. Alternate methods for storing databases

In the current implementation of VLAB, both the oofs and hofs databases are stored as hierarchies of files and subdirectories in a UNIX filesystem. The most important advantage of this design is the fact that the user is able to manually access and modify databases using standard UNIX tools, allowing him to augment the functionality of existing VLAB tools. This design has proven to be beneficial in the early stages of VLAB's development, when the functionality offered by VLAB tools was limited. With the increased database management capabilities provided by the current version of VLAB, occasions when the user has to resort to a command line are very rare. Another important advantage of the current design is the simplicity of its implementation. The UNIX hierarchical filesystem and symbolic links lent themselves extremely well to this design.

The disadvantages of the current design are related to the fact that even moderately sized databases are usually composed of a large number of directories and small files. Many operations on databases need to examine contents of several files, which is translated into a number of system calls, such as opening and closing files. The action of opening and closing files under UNIX operating systems is generally associated with a significant amount of overhead, resulting in performance penalties for VLAB applications. Also, disk controllers usually read and write data in chunks†. If the data files are smaller than the size of these chunks, most of the I/O time is spent by reading and writing unnecessary information. Another disadvantage of storing a small amounts of information in many files is the waste of available resources - in this case the resource is the disk space. Independent of its size, every file on a UNIX filesystem occupies a minimum amount of space,

† determined by the hardware - usually the size of the disk's sectors, which is approximately 512 bytes

equivalent to one block† [35]. If the size of an information to be stored is 1Kb, but the information is split and stored in 10 different files, the total disk space used to store this information is 10Kb - ten times as much as is required.

As an alternative to the current design, the oofs and hofs databases could be stored in a single file using one of the general purpose database engines, such as Oracle [39], or a custom build database engine. The advantages of such design would be visible in many areas. For example, the speed of Browser and hyperbrowser would increase, as the work would be more efficiently distributed between the database engine and the VLAB applications. Most of the system calls performing file related I/O operations in Browser and hyperbrowser could be eliminated, reducing the amount of overhead associated with many operations. Another advantage of such new design is that some of the limitations of the current design would be easier to remove, such as guaranteed unique access to a database, or the undo functionality. Many database engines have such functionality built in.

RAserver can be considered to be a very primitive database engine, providing a set of low-level database operations. If an existing database engine (such as Oracle) was to be used for storing of oofs and hofs databases, the need for RAserver is likely to be completely eliminated, as such database engine could easily perform the duties of RAserver. If a custom based database engine was to be designed, large parts of RAserver's design and implementation could be reused for such purpose.

9.2.10.Distribution of external programs

The information contained in a VLAB object is comprised of data files and a description of actions which can be invoked on these data files. The description of actions specifies how to run external programs (e.g. plant generation software, raytracers, compilers, etc.) on the particular data files. The external programs, however, are not part of the object. It is assumed that the user invoking an object has all the necessary programs properly installed and setup. VLAB itself does not support any mechanism for distributing these external programs.

Since VLAB is to this date mostly used for plant modeling, the problem of distributing programs is 'solved' by bundling the distribution of VLAB with the plant generating software. Unfortunately, this does not eliminate the problem when VLAB is used for managing other types of experiments. For example, if VLAB is used to manage experiments requiring some non-standard simulation software, the majority of users won't

† the smallest number of bytes that can be read and written at one time, usually about 1Kb

be able to invoke such experiments as the non-standard simulation software will most likely not be installed on their systems. This limitation of VLAB is most visible when users try to make the results of their research available to the general public.

The distribution of programs used by VLAB objects has to be properly addressed to improve VLAB's flexibility and usability in areas other than biological plant modeling. VLAB users should be able to invoke any objects from any database, without having to manually obtain, install and setup the software needed to experiment with these objects. The process of acquiring and installing external software should be either completely automated, or at least adequately assisted by VLAB.

The design of a mechanism for external program distribution has to address several important issues. For example, how to deal with incompatibility of binaries among different architectures and versions of operating systems? Do we distribute external programs as sources or as binaries? Using programming languages designed to be portable across multiple platforms, such as Java [8][11] or Python [19][40], might help to solve this problem to some extent, but other issues would have be addressed still. How, if at all, do commercial programs get distributed? How does VLAB enforce that the proper version of Java an Python is installed on the client?

Other questions which need to be answered in the design of mechanism for distribution of external software include: where is the external software going to be installed? Does it remain installed after the experimentation with the particular object is finished? How is the mechanism going to deal with software dependencies - e.g. one program may need other programs in order to function correctly, which in turn can require special libraries. When does the transfer of external programs occur? If the process of distributing the external programs is automated, how can the user downloading the software protect his system from harmful programs such as viruses?

References

[1] Anderson, P., Baran, C., Flanagan, J., Ford, L., Hiyate, S. *Learning Alias V8.* R.R. Donnelley, 1996.

[2] Avrahami, G., Brooks, K. P., Brown, M. H. *A Two-View Approach to Constructing User Interfaces*. Computer Graphics, Volume 23, Number 3, July 1989, pp. 137-146.

[3] Barth, P. S. *An Object-Oriented Approach to Graphical Interfaces.* ACM Transactions on Graphics, Vol 5, No. 2, April 1986. pp. 142-172.

[4] Bernstein, D. J. *Using Motif with C++*. SIGS Books. March 1995. ISBN 1-884842-06-2.

[5] Chatterjee, S., Paramasivam, M., Yakowenko, M. J. *Architecture for a Web-Accessible Simulation Environment.* Computer, June 1997, pp. 88-91.

[6] Cutler, E., Gilly, D., O'Reilly. T. *The X Window System in a Nutshell.* O'Reilly & Associates, second edition, April 1992. ISBN 1-56592-017-1.

[7] Federl, P. *Browser and Landscape Editor for Virtual Laboratory in Biology.* CPSC 502 Final project report. University of Calgary. April 1995.

[8] Flanagan, D. *Java in a Nutshell.* O'Reilly & Associates, first edition, 1996. ISBN 1-56592-183-6.

[9] Garfinkel, S., Spafford, G. *Practical UNIX & Internet Security.* O'Reilly & Associates, 2nd edition, April 1996. ISBN 1-56592-148-8.

[10] Gentner, D., Nielson, J. *The Anti-Mac Interface.* Communication of the ACM August 1996, pp. 70-82.

[11] Harold, E. R. *Java Network Programming.* O'Reilly & Associates. February 1997. ISBN 1-56592-227-1.

[12] Heller, D. *Motif Programming Manual.* O'Reilly & Associates. February 1994. ISBN 1565920163.

[13] Hernadi, I. *The Virtual Laboratory.* http://www.cpsc.ucalgary.ca/Redirect/bmv/vlab. March 1996.

[14] Knelsen, C. *A multipurpose interface for interactive control of multiple parameters.* Master's thesis. University of Regina. 1988.

[15] Lehey, G. *Porting UNIX Software.* O'Reilly & Associates. November 1995. ISBN 1-56592-126-7.

[16] Levine, J. R., Mason, T., Brown, D. *lex & yacc.* O'Reilly & Associates second edition, 1995. ISBN 1-56592-000-7.

[17] Lieberman, H. *Using prototypical objects to implement shared behavior in object oriented systems.* In Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (New York, 1986), Association for Computing Machinery, pp. 214-223.

[18] Lowe, E. M. *Extensions to the Virtual Laboratory.* Master's thesis. University of Calgary, 1995.

[19] Lutz, M. *Programming Python.* O'Reilly & Associates. October 1996. ISBN 0-937175-75-7.

[20] Mercer, L., Prusinkiewicz, P., Hanan, J. *The concept and design of a virtual laboratory.* In Proceedings of Graphics Interface '90 (1990), CIPS, pp. 149-155.

[21] Mercer, L. *The virtual laboratory.* Master's thesis. University of Regina. 1991.

[22] Moen, S. *Drawing dynamic trees.* IEEE Software (July 1990), pp. 21-28.

[23] Nardi, B. A. A *small matter of programming: perspectives on end user computing.* MIT Press, 1993.

[24] Nye A., O'Reilly T. X *Toolkit Intrinsics Programming Manual.* O'Reilly & Associates, second edition, 1990. ISBN 0-937175-62-5.

[25] Nye, A. *Xlib Programming Manual.* O'Reilly & Associates, third edition, 1993. ISBN 1-56592-002-3.

[26] Pearl, A. *Sun's Link Service: a Protocol for Open Linking.* Hypertext'89 Proceedings, pp. 137-146, 1989.

[27] Prusinkiewicz, P., Knelsen, C. *Virtual control panels.* Proceedings of Graphics Interface '88, pp. 185-191.

[28] Prusinkiewicz, P., Lindenmayer, A. The *algorithmic beauty of plants.* Springer-Verlag, New York, 1990 (second printing 1996). With J. S. Hanan, F. D. Fracchia, D. R. Fowler, M. J. M. de Boer, and L. Mercer.

[29] Prusinkiewicz, P., Hanan, J. A *hypertext environment for UNIX*. In Proceedings of Graphics Interface '88 (Edmonton, Canada, June 6-10). pp. 50-55, Canadian Information Processing Society, 1988.

[30] Quercia, V., O'Reilly, T. X *Window System User's Guide.* O'Reilly & Associates, 4th edition, May 1993. ISBN 1-56592-014-7.

[31] Reeves, W. T., Ostby, E. F., Leffler, S. J. *The Menv Modeling and Animation Environment.* The Journal of Visualization and Computer Animation, Vol. 1: 33-40, 1990.

[32] Smith, R. G., Barth P. S., Young, R. L. A *Substrate for Object-Oriented Interface Design.* A Substrate for Object-Oriented Interface Design, pp. 253-315. MIT press, 1987. ISBN 0-262-19264-0.

[33] Snider, A. *An interactive, physically-based simulation system. Master's thesis.* University of Regina, Regina, Canada. 1992.

[34] Stern, H. *Managing NFS and NIS.* O'Reilly & Associates. June 1991. ISBN 0-937175-75-7.

[35] Stevens, W. R. *Advanced Programming in the UNIX Environment.* Addison-Wesley. June 1992. ISBN 0201563177.

[36] Stevens, W. R. *Unix network programming.* Prentice-Hall, Englewood Cliffs, 1990.

[37] Strauss, P. S., Carey, R. *An Object-Oriented 3D Graphics Toolkit.* Computer Graphics, 26, 2, July 1992, pp. 341-349.

[38] Stroustrup, B. *The C++ Programming Language.* Addison Wesley, second edition, 1991. ISBN 0-201-53992-6.

[39] Urman. S. *Oracle PL/SQL Programming; The Essential Guide for Every Oracle Programmer.* Osborne, April 1996. ISBN 0078821762.

[40] Watters, A. *Internet Programming with Python, with CD-ROM.* M & T Publishing. September 1996. ISBN 1558514848.

[41] Wegner, P. *The Object-Oriented Classification Paradigm. Research Directions in Object-Oriented Programming*. pp. 479-560. MIT press, 1987. ISBN 0-262-19264-0.

[42] Wernecke, J. *The Inventor Mentor.* Addison-Wesley, 1994. ISBN 0-201-62495-8.

[43] Wolfram, S. *The Mathematica Book.* Wolfram Media & Cambridge University Press, 1996.

[44] Zeleznik, R. C., Conner, D. B., Wloka, M. M., Aliaga, D. G., Huang, N. T., Hubbard, P. M., Knep, B., Kaufman, H., Hughes, J. F., Dam, A. *An Object-Oriented Framework for the Integration of Interactive Animation Techniques.* Computer Graphics, Volume 25, Number 4, July 1991, pp. 105-112.

APPENDIX A *RA class*

```
static RA_Connection * new_connection (
          const char * host_name,
          const char * login_name,
          const char * password );
```

- Establishes a new connection to RAserver daemon running on a computer `host_name`. Then a `LOGIN` request with `login_name` and `password` is sent. If the response received from RAserver is positive, a valid connection is returned. The result of this function is a pointer to an object of type `RA_Connection`, which has to be supplied to all other methods of the `RA` class.
- If `new_connection()` is called with a `host_name` that represents a local machine, then RAserver is actually not accessed, but a special type of `RA_Connection` is created and flagged local. Otherwise `RA_Connection` is flagged remote. When performing operations on a file through a local connection, the requests are actually not sent over a network to RAserver, but rather invoked locally, directly from the called method. This allows for a consistent mechanism for accessing any VLAB object files using the same calls, whether in a local or a remote database.

```
static void close_connection (
          RA_Connection * connection );
```

- Closes an existing connection. If connection type is remote, first a `LOGOUT` request is sent, and a `LOGOUT` response received. Then the socket connection is shut down. If the connection type is local, nothing happens.

```
static int Compare_files (
          RA_Connection * connection1,
          const char * fname1,
          RA_Connection * connection2,
          const char * fname2 );
```

- Compares two remote files to each other. If the two connections point to the same RAserver, a single RA_COMPFILE_REQUEST is sent to RAserver, which will complete the operation by performing the file comparison on the server.
- If the connections are not on the same host, but they are both remote connections, both files are first downloaded to a local computer, compared and then deleted. If one of the connections is remote and the other local, only one file is downloaded to the local computer. If both connections are local, a local comparison is performed.

```
static int Copy_file (
          RA_Connection * src_connection,
          const char * src_fname,
          RA_Connection * dst_connection,
          const char * dst_fname );
```

- Copies file `src_fname` from `src_connection` to `dst_fname` on `dst_connection`. If both files are on the same remote computer, `RA_COPYFILE_REQUEST` is sent to RAserver instructing it to perform the operation directly on the server, without downloading any data to the client. Otherwise the `src_fname` is downloaded from `src_connection`, and then uploaded as `dst_fname` through `dst_connection`.

```
static int Unlink (
          RA_Connection * connection,
          const char * fname );
```

- Deletes a file specified by `fname` on the server.

```
static int Deltree (
          RA_Connection * connection,
          const char * dirname );
```

- Recursively removes a directory on the server specified by `dirname`.

```
static int Symlink (
          RA_Connection * connection,
          const char * src_fname,
          const char * dst_fname );
```

- Creates a symbolic link between `src_fname` and `dst_fname` on the server.

```
static int Rename (
          RA_Connection * connection,
          const char * src_fname,
          const char * dst_fname );
```

- Renames a file specified by `src_fname` to `dst_fname`.

```
static int Stat (
          RA_Connection * connection,
          const char * fname,
          RA_Stat_Struc * stat_struc );
```

- Obtains various information about a file specified by `fname`. The results are stored in `stat_struc`.

```
static int Get_dir (
          RA_Connection * connection,
          const char * dir_name,
          char *** list );
```

- Obtains a list of entries in a directory specified by `dir_name`. The result is stored in `list` as an array of strings. The end of the list is denoted by a `NULL` pointer.

```
static int Readlink (
          RA_Connection * connection,
          const char * fname,
          char * (& result) );
```

- Returns the file to which a symbolic link `fname` points.

```
static int Is_link (
          RA_Connection * connection,
          const char * fname );
```

- Finds out whether `fname` on a remote host is a symbolic link.Calls `Stat()` to do the actual work.

```
static int Write_file (
          RA_Connection * connection,
          const char * fname,
          const char * buffer,
          const long size );
```

- Creates a file `fname` on the server. The contents of this newly created file will be taken from the supplied parameter `buffer`. The parameter `size` specifies the number of bytes in `buffer`.

```
static int Put_file (
          const char * local_fname,
          RA_Connection * connection,
          const char * remote_fname );
```

- Copies a local file `local_fname` into a file `remote_fname` located on the server. This function is implemented by first reading the contents of the local file into memory, and then calling the `Write_file()` method.

```
static int Read_file (
          RA_Connection * connection,
          const char * fname,
          char * ( & buffer),
          long & size );
```

- Retrieves the contents of the file `fname` on the server. The contents of the file are returned to the caller in `buffer` and the size of the contents in `size`.

```
static int Fetch_file (
          RA_Connection * connection,
          const char * remote_fname,
          const char * local_fname );
```

- Copies the contents of a file `remote_fname` located on the server into a local file `local_fname`. This method is implemented by first using the `Read_file()` method to read the contents of the remote file into memory, and then the local file is created with these contents.

```
static int  Get_file_type (
          RA_Connection * connection,
          const char * fname,
          RA_File_Type & type );
```

- Determines the type of a remote file `fname`. Uses method `Stat()` to retrieve the information about the remote file, from which the file-type is determined.

```
static int Mkdir (
          RA_Connection * connection,
          const char * path,
          mode_t mode );
```

- Creates a directory `path` with permissions `mode` on the server.

```
static int Rmdir (
          RA_Connection * connection,
          const char * path );
```

- Removes an empty directory `path` on the server. If the directory to be removed is not empty, the operation will fail.

```
static int Access (
          RA_Connection * connection,
          const char * fname,
          const int amode );
```

- Determines a possible access to a remote file `fname`. For example, this method is used to determine whether a directory is writable. Uses `Stat()` method to obtain the necessary information.

```
static int Realpath (
          RA_Connection * connection,
          const char * path,
          char * (& result) );
```

- Returns a real path to the remote file `path` in `result`.

```
static int Get_extensions (
          RA_Connection * connection,
          const char * path,
          char ** (& list) );
```

- Returns a list of extensions and their attributes of a remote object specified by `path`. The result is stored in `list` as a list of strings. Each string is composed of two parts, extension name and extension's attribute, separated by a `\0` character.

APPENDIX B *More on panel manager*

B.1. Example of creating a control panel

In this section I present an example, in which a complete control panel is created. Imagine that the user wants to design a control panel for a data file which is to be used by some physically based simulation program. The data file is called `simulate.dat`, and contains the following information:

```
Gravity:       9.81
N-iterations:  10
Rendering:     gouraud
```

The user would like to control three parameters: gravity (ranging between -100.0 and 100.0), number of iterations (ranging between 1 and 100), and the rendering method (available options being: wire, flat, gouraud and phong).

The user creates the control panel from scratch, by invoking panel manager in the edit mode without specifying any file-name (Figure B-1). Since no file-name was specified on the command line, panel manager shows an empty window with no components. Before the user can start adding new components into the control panel, he has to first create a new panel by choosing ***File-> New*** menu from the menu bar. After the panel is created, he sets the title of the panel to `Animation` in its attribute editor (invoked from the Panel's pop-up menu) as seen in Figure B-2. After the title is set, the user disposes of the attribute editor by clicking its ***Hide*** button, located at its bottom edge.

Now the user creates a floating point range control which will be used to modify the gravity parameter - by choosing ***Create-> Frange*** from the pull-down menus. This will position a new floating point range (frange) at the left top corner of the empty panel. The user drags the new frange into the top center of the panel, and resizes it so that it spans from the left edge to the right edge. Then he invokes its parameter editor (Figure B-3) and sets its Font to family: `helvetica(adobe)`, Style: `medium normal` and size: `14`. After that the *Min, Max, Increment* and *Page Increment* values are set to '-100.0', '100.0', '-0.01' and '0.25'. Then the user sets the value of the *File* field to `simulate.dat`, and the *Field Prefix* to '`Gravity:`'.

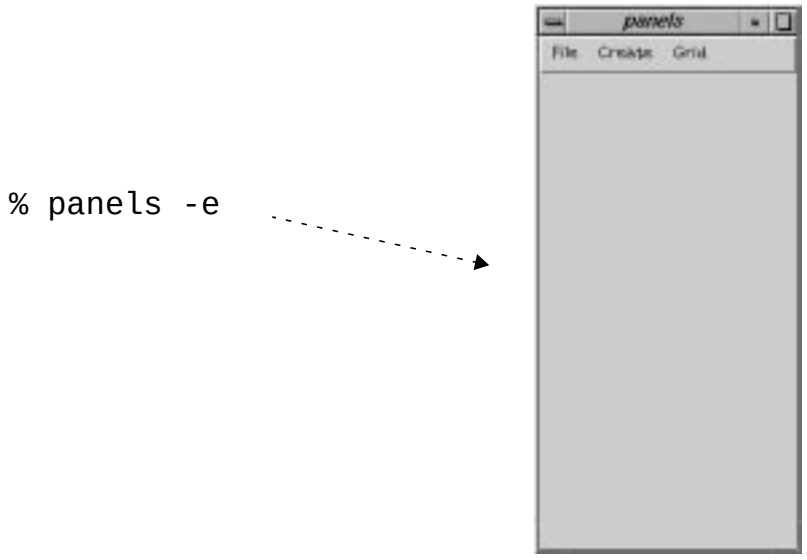


Figure B-1: Invoking panel manager in edit mode

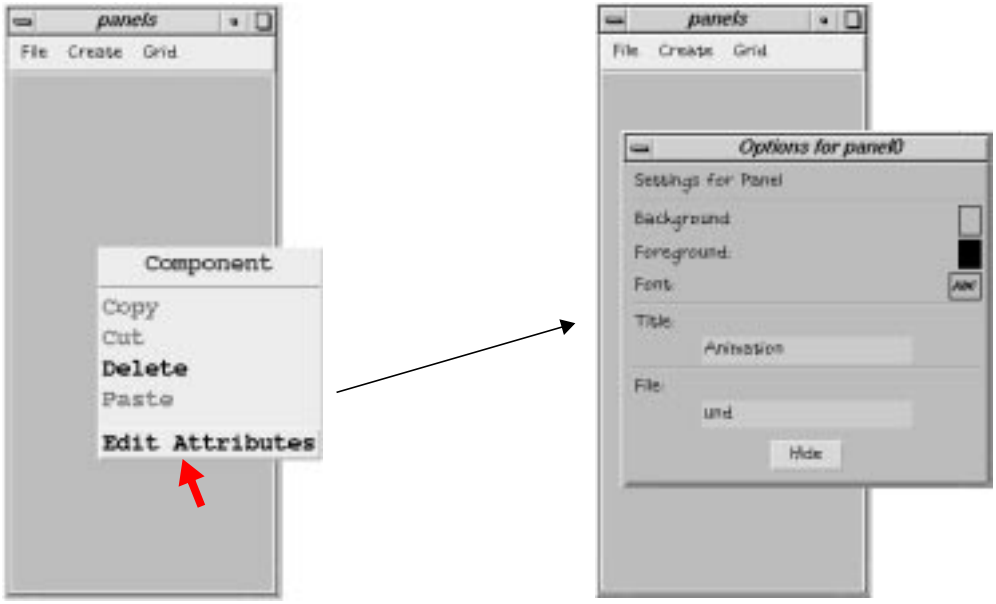


Figure B-2: Setting the panel's title in panel's attribute editor

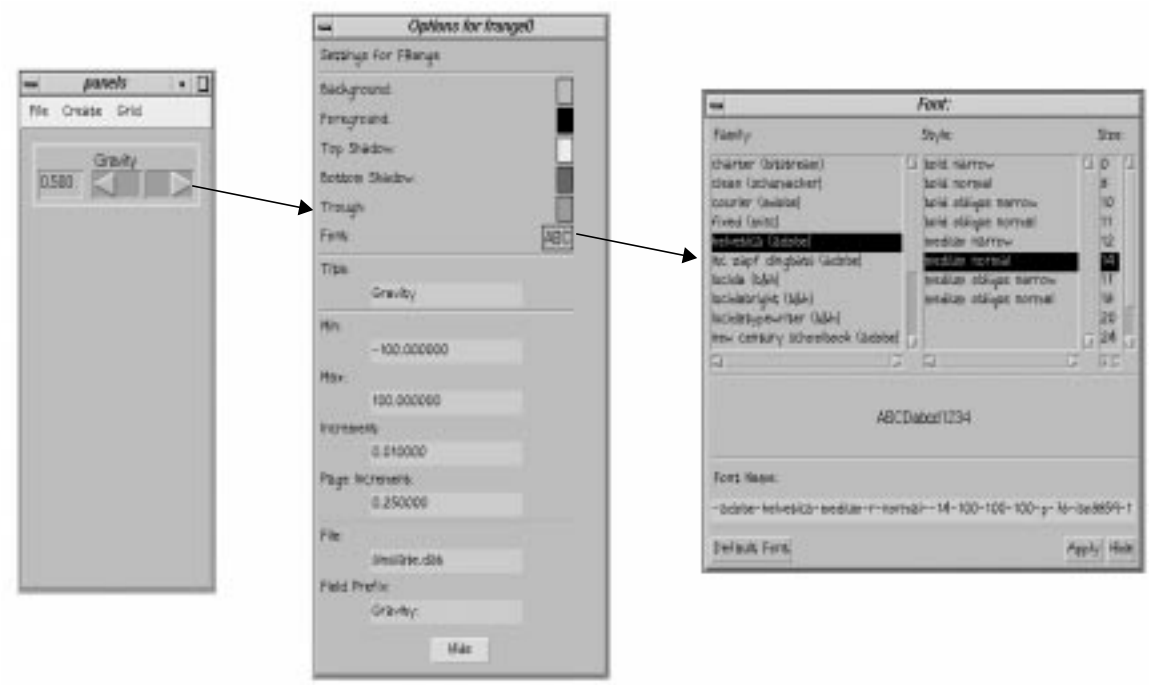


Figure B-3: Using the floating point range's attribute editor and the font chooser

Now the user creates an integer range control (irange) for modifying the number of iterations and position/resize it to fit under the frange component. Then the attribute editor is invoked, and the fields are set to the following values: *Min*: `1`, *Max*: `100`, *Increment*: `1`, *Page Increment*: `10`, *Title*: '`Number of iterations`', *File*: `simulate.dat`, *Field Prefix*: '`N-iterations:`'. Figure B-4 shows panel manager after these steps are performed.

Figure B-4: Panel manager in edit mode with two components

Finally, the user creates the choice component by choosing the ***Create-> Choice*** menu. After resizing and placing the choice component to be aligned with the previous two components, he invokes its attribute editor and set the fields to *Title*: '`Rendering Model`', *File*: '`simulate.dat`', and *Field Prefix*: '`Rendering:`'. Since four options are needed for the rendering type and by default the choice control comes with only three choices, the user adds one more entry to the list of choices by clicking on any of the ***Add*** buttons. Then he initialize the list to values as shown in Figure B-5.

Figure B-5: Editing choice's attributes

The design of the control panel is now complete, so the user saves the panel into a file called '`panel.pnl`'. He chooses ***File-> Save...*** from the pull-down menus, which will display a file selection dialog. In this dialog he enters `panel.pnl` into the *Selection* box, and click the ***OK*** button.

To modify the parameters in a file called `simulation.dat` using the control panel that was just created, the user would invoke panel manager using the following command:

```
% panels panel.pnl
```

Panel manager would then read the current values of the parameters and display the control panel as shown in Figure B-6. Modifying the values through the displayed controls in the control panel will automatically result in changing the values of the parameters in the file `simulation.dat`. If the user wants to change the interface of the control panel, he would invoke panel manager in the edit mode on the panel definition file, i.e.:

Figure B-6: Final appearance of the control panel

```
% panels -e panel.pnl
```

B.2. Component attributes

The following table summarizes the attributes and their types for each control supported by panel manager 3.0.

Table B-1: Component attributes

Component Type	Attribute Name	Attribute Type	Attribute Name	Attribute Type
panel	width	integer	file	string
	height	integer	foreground	string
	title	string	background	string
	name	string	font	string
integer range	x	integer	label	string
	y	integer	field_prefix	string
	width	integer	file	string
	height	integer	background	string
	name	string	foreground	string
	min	integer	trough_color	string
	max	integer	bottom_shadow	string
	inc	integer	top_shadow	string
	page_inc	integer	font	string

Table B-1: Component attributes

Component Type	Attribute Name	Attribute Type	Attribute Name	Attribute Type
floating point range	x	integer	label	string
	y	integer	field_prefix	string
	width	integer	file	string
	height	integer	background	string
	name	string	foreground	string
	min	double	trough_color	string
	max	double	bottom_shadow	string
	inc	double	top_shadow	string
	page_inc	double	font	string
label	x	integer	label	string
	y	integer	background	string
	width	integer	foreground	string
	height	integer	font	string
	name	string		
choice	x	integer	file	string
	y	integer	background	string
	width	integer	foreground	string
	height	integer	toggle_color	string
	name	string	bottom_shadow	string
	label	string	top_shadow	string
	choices	list	font	string
	field_prefix	string		

B.3. Class component

The following table describes the class `Component`.

Table B-2: Class component

```
class Event;
typedef void (* Handler) (Component &, Event &);
```

- These two types are used for the callback function in edit mode. They are explained in Section 5.4.3.

```
enum ComponentType type;
```

- stores the type of the component

```
char name[ 256];
```

- stores the name and number of the component (for example, choice components would be named choice0, choice1, ...)

```
long x, y, width, height;
```

- [`x,y`] defines the position of the left upper corner of the component when rendered, with respect to the parent component. `width` and `height` define the component's dimensions.

```
Component * parent;
```

- stores the pointer to the parent component

```
Component();
virtual ~Component();
```

- constructor & destructor

```
static Component * create( const char * name,
                           AssignmentList * al);
```

- Constructor used by the Parser class to build a new component with attributes gathered in the the parameter `al`. AssignmentList is a class that holds a list of attribute names and their values as parsed in from the file.

```
virtual void add_child( Component * c);
```

- Adds a component to the list of children. This method is declared virtual, but it should not be modified by the derived class, unless the class needs to be able to do some geometry restraints on its children at their creation time.

Table B-2: Class component

```
virtual char * to_str( void);
virtual void print( long indent = 0);
```

- Used to access the information about a component in a printable text format.

```
virtual void render( Widget parent) = 0;
```

- Renders the component by creating and managing Xt widgets. The top-level widget is created as a child of the supplied parameter parent. After the component is rendered, the `render()` method is called for every child. Every derived subclass has to define its own method for rendering.

```
virtual int init( void);
```

- Calls `common_init()` of the component, and then `init()` on all children. This method is supposed to be overloaded by all derived classes, but if overloaded, it should still include a call to `common_init()`.

```
void highlight_off( Boolean recursive);
void highlight_on( Boolean recursive);
```

- Sets/unsets the highlight for a component, and if `recursive` is set, then for all children as well.

```
Component * get_root( void);
```

- Convenience function - locates and returns the root component of the tree which the component is part of.

```
int common_init( void);
```

- Makes some of the current attributes to inherit the values from the parent (i.e. `background`, `font`). Then the current parameter value is extracted into `token` - the parameter location is defined in `fname` and `field_prefix`.

```
void set_run_mode( void);
```

- Sets the mode of all components in the tree to the run mode. This function should be called only on the root component of the tree.

```
void set_edit_mode( Handler handler);
```

Table B-2: Class component

- Sets the mode of all components in the tree to edit mode, and assigns to each component an event handler as defined by the parameter `handler`. This function should be called only on the root component of the tree.

```
virtual void get_geometry( long & x_ret, long & y_ret,
                    long & width_ret, long & height_ret);
```

- Returns the geometry of the component.

```
virtual void get_root_xy( long & x_ret, long & y_ret) = 0;
```

- Returns the coordinates of the left upper corner of the rendered component with respect to the root window (screen).

```
virtual void set_geometry( long x, long y,
                           long width, long height) = 0;
```

- Every derived component has to define this function to be able to accept resize/re-position requests. These requests can be granted or refused.

```
virtual Widget get_rwidget( void) = 0;
```

- Returns the top-level widget used to render this component.In the current implementation, this widget is of type `XmFrameWidgetClass` for all components.

```
virtual void remove_child( Component * comp);
```

- Removes a child from the list of children. This method should be called in the destructor of any component on the parent component.

```
virtual void edit_settings( void);
```

- Displays the attribute editor for the component.

```
virtual void redraw( void);
```

- When attributes of a component are changed, this method has to be called in order to synchronize the changes made to the component with its visual appearance.

```
virtual void dump( Mem_IO &, long indent = 0);
```

- Dumps all information about the component into memory. This function is called by the main program when the user decides to save the control panel into a file. This method also calls `dump()` on every child, and therefore the calling program should only call `dump()` on the root-level component.

Table B-2: Class component

```
virtual void _set_highlight( Boolean highlight) = 0;
```

- All derived classes have to define this method. This method will change the appearance of the class (`highlight` specifies whether the border of the component should be of distinct color or not). This method is called by `highlight_on()`.

```
Color select_color;
```

- Defines the color of the highlight.

```
Handler callback_handler;
```

- Pointer to the user defined callback function. This is only used in edit mode, and therefore will be described in the next section.

```
Color background;
Color foreground;
FontStyle font;
char file_name[ 4096];
char field_prefix[ 4096];
```

- Basic attributes for every component.

```
Boolean highlighted;
Boolean rendered;
char edit_mode;
```

- Various state variables.

```
Mem token;
```

- The value of the parameter as extracted by `common_init()`.

```
long n_children;
Component ** children;
```

- Stores the list of all children.

```
OptionsDialog * options_dialog;
```

- A pointer the attribute editor, created by `edit_settings()`.