6-20-1988

A Window-Oriented User-Interface for Image Processing Systems on UNIX based workstations.

Sandeep Mehta

Rochester Institute of Technology

School of Computer Science and Technology

A Window-Oriented User-Interface for Image Processing Systems on UNIX based workstations.

by

Sandeep Mehta

A thesis submitted to the
Faculty of the School of Computer Science
in partial fulfillment of the requirements for the degree of
Master of Science in Computer Science.

Approved by:

Donald L. Kreher

Dr. Donald L. Kreher (advisor)

Evelyn Rozanski

Prof. Evelyn Rozanski

Peter H. Lutz

Dr. Peter Lutz

June 20. 1988

Title of Thesis: A Window Oriented User Interface for Image Processing Systems on UNIX Based Workstation

I SANDEEP MEHTA hereby (grant/~~deny~~) permission to the Wallace Memorial Library, of RIT, to reproduce my thesis in whole or in part. Any reproduction will not be for commercial use or profit.

or

I _____ prefer to be contacted each time a request for reproduction is made. I can be reached at the following address:

Sandeep Mehta

_____

Date: 06/23/88

A Window-Oriented User Interface for Image Processing Systems on UNIX† based Workstations

Sandeep Mehta

School of Computer Science, Rochester Institute of Technology,
Rochester, New York.

ABSTRACT

The advent of digital image processing has led to the availability of a very large number of software systems . However there is an absence of a cohesive general-purpose image processing environment isolated from hardware or the underlying operating system. The trend in computing in science and engineering, is towards distributed workstations, especially with the availability of high-performance microprocessors. Hence there is a need for a unified software environment on workstations for use in scientific applications.

This thesis describes the design and implementation of a window oriented user-interface. The interface runs on top of a Image Processing system, running on workstations under the UNIX † environment and uses the network transparent X window system ‡. The visual shell-like environment is targeted at the end-user with a scientific background needing image processing capabilities, but not necessarily with a computer background.

The User-Interface is primarily a tool for use by a single user, although the underlying system operates in a multiuser multitasking environment. The objective of this exercise is aimed at providing rapid, easy and visual capability in processing images at a session level. It integrates graphics capabilities with high speed computing. All processing capabilities provided at the command line level, are available via dialog boxes, buttons & multi-level menus.

June 20, 1988

† UNIX is a trademark of Bell Laboratories.

Acknowledgements

The author wishes to acknowledge the co-operation and encouragement provided by his advisor Dr. Donald Kreher, during the course of this work, as well as Prof. Evelyn Rozanski, for agreeing to be on the committee, and providing constructive criticism of the proposal and the thesis in its many forms of completion.

This work was carried out with the Optical Engineering Group at the Laboratory for Laser Energetics, University of Rochester. I would like to thank Terrance Kessler for offering me an opportunity at the Lab and encouraging me to pursue my thesis work with the Optical Engineering group. Without his guidance none of this would have been possible. I have to thank Steve Swales for his numerous ideas, help & suggestions which helped me out of so many tight corners throughout the course of this project; without his assistance a lot of this project could not have been designed, and Nitin Sampat for keeping the "*big picture*" in focus. I also thank Denise Ondishko of the Image Analysis Laboratory, with the system support and words of encouragement during many late programming blitzes.

Finally I would like to thank my family in India for supporting & encouraging me through the past two years in graduate school and making it all possible.

Table of Contents

Abstract 1

Acknowledgements 2

Chapter 1: Introduction to the User-Interface concept 3

§ 1.0 The need for an Interactive Computing Environment 3

§ 1.1 The User-Interface why, what and how ? 3

§ 1.2 Choice of programming language and environment 6

§ 1.2 X Network Transparent Window System 7

§ 1.3 Design Philosophy of the User-Interface 8

Chapter 2: Design of the XIP User-Interface 10

§ 2.1 Global functionality of the XIP User-Interface 10

§ 2.2 Functions Specification of Modules 12

§ 2.2.1 Error Handler Daemon (XERRWIN) 12

§ 2.2.2 Command Window Handler (XCOMWIN) 14

§ 2.2.3 Button Handling (XBUTTONS) 15

§ 2.2.4 Displaying and Saving Images (XDISPIMG & XSAVEIMG) 15

§ 2.2.5 Region of Interest Processing (XTRACTROI,XPROCROI,XDISPROI) 16

§ 2.3 Other functions performed by XIP 18

§ 2.4 Screen Organization and Management 18

§ 2.5 Limitations and restrictions in the XIP design 20

Chapter 3: The Error Window Daemon 23

§ 3.1 Introduction 23

§ 3.2 Design 23

§ 3.3 Implementation 24

§ 3.4 Caveats and Suggestions 25

Chapter 4: The Command Window Handler 26

§ 4.1 Introduction 26

§ 4.2 Design & Implementation 26

§ 4.3 Caveats and Suggestions 27

Chapter 5: The Button Handler 28

§ 5.1 Introduction 28

§ 5.2 Design & Implementation 28

§ 5.3 Caveats and Suggestions 29

Chapter 6: Image Conversion and Display 30

§ 6.1 Introduction 30

§ 6.2 Image Format 30

§ 6.3 Image Conversion in XIP 34

§ 6.4 Dithering, Color Maps, and Look-up tables 36

§ 6.5 Design and Implementation of Image Display program 38
§ 6.6 Caveats and Suggestions 38
Chapter 7: Region of Interest Processing 39
§ 7.1 Introduction 39
§ 7.2 Extraction 39
§ 7.3 Resizing 40
§ 7.4 Insertion 41
§ 7.5 Processing 41
§ 7.6 Caveats and Suggestions 42
Chapter 8: Menu Building and Generation 44
§ 8.1 Introduction 44
§ 8.2 Design and Implementation 44
§ 8.2.1 Tokens and reading the default file 45
§ 8.2.2 Menu Generation and Command Building 47
§ 8.3 Caveats and Suggestions 47
Chapter 9: Operation and Results 49
§ 9.1 Introduction 49
§ 9.2 XIP User Interface Control 49
§ 9.3 Results 50
§ 9.4 Conclusions 50
Appendix A: XIP User Interface header files ..... A
Appendix B: XIPMENU Menu Structure Definitions ..... B
Appendix C: Results C
Appendix D: Selected Statistics ..... D
Bibliography ..... D

Chapter 1: Introduction to the User Interface concept

1. The need for an Interactive Computing Environment

The rapid advent of Digital Image Processing in the past few years has led to a plethora of software systems being available in the commercial market as well as in academia. Each of these systems has almost the same functionality. Since digital image processing is a very computationally intensive process, it has usually been the case that portability, generality, and ease of learning and use have been sacrificed in favor of speed and efficiency. However it is very crucial that a system of substantial complexity and size be designed with the factors which are usually ignored.

A simple example will serve to illustrate the level of computation required. Consider a grey-level 512 X 512 pixel image, with 12 bits/pixel resolution. If an operator performs an operation on the 8 neighbors of each pixel over the entire image, over 32 million operations would be performed per image. Moreover each operation could itself be quite complicated. Even though this magnitude may seem overwhelming, with the current hardware available, such computations are relatively easy to perform . The cost and size of fast memory is no longer a major restriction.

That brings us to outlining the design of an image processing environment which addresses the issues of software engineering, code modularity, portability, device independence, flexibility, and ease of use. This proposed system while focusing on these issues does not necessarily compromise speed and efficiency. As Kasik states: "*The role of interactive computing has continually increased as the cost of more powerful terminals and computing hardware has decreased.*".[KAS82] It is the goal of this project to demonstrate the significance of interactive computing in the case of Digital Image Processing.

1.1. The User-Interface - why, what and how ?

There already exist a variety of systems which claim to satisfy the criteria of portability, generality and ease of use. e.g. PIPS [HAV86] , FIPS [WES85] and HIPS. In the author's opinion all of these systems fall short in some facet of their functionality. The details are numerous and thus have been

avoided. As Addington [ADD84] states :

> *The real challenge in software design has become the development of user interfaces applicable to a particular application or scenario.*

This is demonstrated in the system SDCIPS discussed by Addington[ADD84] and Green.[GRE85] It is generally agreed upon by many authors of these Image Processing software systems, that the use of personal workstations as a host machine to provide an image processing environment to the image processing expert/analyst is highly efficient as opposed to providing them with a programmer's environment on larger minicomputers or mainframes.

Although portability is a primary design goal of this project it is very hard to completely isolate such an elaborate system from the operating system environment of it's host. The host operating system in this project is UNIX which is known by most users and developers as recluse and "expert friendly" in nature. Although it offers tremendous power in the form of pipelining, multiple options, command line macros, shell-scripts etc., it can be very inconvenient to the user. UNIX is also rather unforgiving to the user's errors. A simple string of commands to histogram equalize an image , remap it onto an output image and display it, looks something like this.

```
% eqlz.run -m neigh -n 2 input.img | remap.run -t  | disp.run -P color
```

A minor syntactical error in the beginning requires that the entire command be re-entered. Command line editing is primitive if not absent. There is no interactive and interpretive facility to build up complicated commands from simpler ones. Thus a novice user remains at his stage of ignorance until he can read the entire documentation or an expert can provide him the complex details.

One alternative is to design an interactive menu-system which runs on top of the underlying programs. In fact such a menu generator does exist, which interactively collects the command options and arguments and executes the program.[SWA86] Of course the pipelining capability of UNIX is not available at the menu level. However that does not entirely remove the process of dealing with the operating system, cryptic commands and error messages.

We have assumed so far that the programs are executing on a single machine using its resident file systems. In reality such environments comprise of a network of UNIX workstations with a

distributed file system like NFS‡. If a program resides on a host on the network it is not feasible for the end-user to have to remotely login onto the other machine to run his program. Besides a remote login will not provide transparent access to the user's data which may be anywhere on the network. Thus network transparency is also an issue to be considered in this project.

Keeping these issues in mind a need has been envisioned for a user interface which can provide a single user with a transparent image processing environment, on a workstation console. The user should have the ability to perform the following kinds of tasks at a session level as discussed by Westrup, Kegel, and Gras.[WES85]

- Access to all types of images (independent of their original format).
- Display/print images independent of devices.
- Process/analyze images independent of which machine the program resides on.
- Display any error messages in a separate window.
- Ability to access on-line documentation.
- Means to build more complex operations out of simple ones.
- Means to execute basic operations asynchronously.

Overall the interface should be able to provide an environment which is usable by the programmer and the end-user with sufficient flexibility . An appropriate description of what a user interface should do is also provided by Addington[ADD84] :

> *In the simplest version of a user interface, an experienced image processing analyst or software developer can execute program modules directly by transferring the appropriate commands to the desired software modules. A more complex user interface may involve a separate set of application software that supports a full-screen interactive menu driven interface, keyboard function keys, graphics tablet, mouse etc.*

Since the user interface is based on a distributed system it should be able to provide network transparency. To quote Densmore and Rosenthal[DEN87] :

> *The needs for portability, distribution, and standardization are encouraging the implementation of window systems for UNIX as network window servers rather than as extensions to the operating system kernel. These servers are user-level daemon processes; clients connect to them and make what are effectively remote procedure calls in order to create/destroy windows and draw in them.*

Using the X window system † on the host machine(s) provides us with exactly such a network window system. The image processing user interface proposed is window oriented which uses client processes to carry out various functions of the interface. These client programs are interacting with the

‡ NFS (Network File System) is a trademark of Sun Microsystems.
† Henceforth X window system is also referred to as X.

X server, and accessing/utilizing its resources i.e. screen, mouse, keyboard etc. All the options which are available to the user at the command line are available at the user-interface level using a menu system. Although only one user can access a console at a given time, the multiprocessing capability is not taken away. There is a single user interface process handling all communication between the user (screen) and the system, which spawns child processes to perform specific functions, or handle some subset of the *user-application* interaction as mentioned above.

The following sections address the *global* issues in the design of the Image Processing User Interface further described in Chapter 2. First the choice of a programming language and environment is explained, followed by a brief overview of X. It should be mentioned at this point that although the concept of device/machine independence is primary it has been observed that X is portable to most UNIX and some non-UNIX systems, and is becoming an industry windowing standard. Thus X is chosen as the base for development for this project. Finally the issues in the user-interface internals design have been discussed.

1.2. Choice of programming language and environment.

Since the user-interface involves substantial interaction with the applications that lie under the visual environment it is easy to narrow down the choice of programming languages to a limited set of languages which can interact with these applications. This discussion does not examine the relative merits and demerits of languages but only cites reasons for the choice already made. The language chosen is C which is general purpose enough to develop most applications, and powerful enough to use the machine with maximum efficiency. Another factor in its favor is that the X window system has been mostly implemented in C, and so has the image processing system for which this user interface was primarily designed and tested on. C provides tremendous power when it comes to dealing with system software like the UNIX operating system, which has itself been mostly written in C.

Having decided on C as the language choosing UNIX as the programming environment is logical. As with C, UNIX has been the host operating system for the evolution of X, and the image processing software. Thus our choice of C and UNIX leads to software system integrity, and flexibility.

1.3. X - Network Transparent Window System

This window system for UNIX runs on computers with bitmap terminals. The X server distributes user input to and accepts output requests from various client programs located on any machine in the Internet domain.†[LEF,TAN81] X supports overlapping windows, fully recursive subwindows, text and graphics operations within windows. It also has a capability to emulate most terminals *(xterm).*

X runs it's window manager as a client program, which is very convenient as it lets the user write or modify his own window manager. Since X imposes little access control this approach is elegant and lends itself to using other client software such as this user interface, concurrently. Some window systems such as Andrew[MOR86] attempt to enforce a consistent style of user interface across all applications that use them, however X provides only a low-level mechanism and does not specify any details of appearance or functionality of an application program's user interface. It is difficult to write each application from scratch, thus an additional layer above the basic window system is necessary. This additional layer is provided in the form of a "toolkit" which lets an application program easily and rapidly construct items such as windows, menus, and scroll-bars from the toolkit library. X Version 10 does have a toolkit but it is not portable to all implementations of X Version 10, hence the current development has been carried out using the lower level C library interface.[GET86] The Version 11 implementation of X comes with many toolkits and porting the applications across does not pose a problem.

1.4. Design Philosophy of the User-Interface

The user interface is the software which interacts between the user and the operating system as well as the application software. This indicates that the user interface definition must actually define two interfaces-the *user interface*: the higher level programs/routines by which the user and the interface interact; plus an *application interface* through which the interface communicates with the operating system and the application programs. The latter interface is transparent to the user. The user interacts with the system environment via one process, to be able to maintain session logs and to be able to maintain a basic level of consistency across all applications being accessed via the interface. Such an

† refer to Sec. 8.2 on Interconnection of Packet Switching Networks, in *Computer Networks* by Andrew Tanenbaum.

approach has also been suggested by Hayes.[HAY85] In the same article Hayes suggests specifying the user-application interaction at a level more abstract than the language the application was implemented in. Let us then agree that the interface should have a higher level of communication. Then there are 3 possible ways of implementing the user interface.

- as a package of subroutines which can be called from the application code.
- as a non-executable external description of an application's interface.
- as an executable external definition of the interaction required by an application

As mentioned in Sec. 1.3, the X window system protocol has a low level C language interface, on top of which additional libraries can be built or applications written. This C library can be used by the interface to interact with the window system via a stream connection.

Contrary to the idea that using subroutine libraries would mean recompilation and relinking the application source code, the user interface described in the next chapter is designed to be as independent of the image processing system as possible. There are "hooks" provided in the interface to allow execution of applications, but the *application interface* remains the same as far as possible. Thus the inconvenience of experimenting with variations in the user interface is no longer critical. The interference and interplay between the various aspects of the interaction (e.g. mouse and keyboard events versus events generated asynchronously by other processes) within the interface are not a disadvantage either as the interface has been relatively isolated from the application software system. The advantages gained by using the subroutine library approach are:

- the X window system is becoming an industry windowing standard, and so it's subroutine library will also be standardized much like the industry graphics standards were established.
- low level interaction details are better handled through subroutine packages.

The non-executable external description of the application interface approach is useful for comparisons but cannot be used for simulations of the final interface. It may also have trouble with dealing with more than one level of abstraction, which disqualifies it from our design approach.

The executable external interface does overcome the problems faced by the non-executable description. However the two methods described by Hayes, namely Form-Based Abstractions and Transitions Network-Based Abstractions are not tailor made to the Image Processing environment. There are

some key features to make note of, as some of them are incorporated in this user interface.

- A coarse grained command interaction with a finer grain when needed, to obtain missing command parameters and arguments etc.
- Graphical representation of the format in which the required information needs to be entered.
- The interface system interpreting the user input keeps track of the current value of each field in the window format.
- Correction of erroneous input, interactively.
- Integral on line-help and documentation facility.
- Command looping in the interface using highlighted icons and previously stored parameters to build up the loop.

The Recursive Transition Network based interface abstraction uses the notion of interface modes between which transitions take place as a result of user actions. Hayes, however, states that since this involves distinct states, and transitions are determined strictly by user input, their suitability to graphical interfaces is a gray area. This is because interface feedback events are not always associated with a mode change. Also enforcing restrictions on transitions may rob the user interface of some of its flexibility.

Having decided on a specific approach subsequent chapters discuss the design and implementation of the user interface, the data structures needed, the approach to achieving independence from application programs, handling different image formats, graphics devices etc. Variations from the original design objectives as implementation proceeded is also addressed to illustrate the difference between design goals and actual implementation.

Chapter 2: Design of the XIP User Interface

This chapter covers the actual design of the XIP user interface in accordance with the design criteria highlighted in the previous chapter. The acronym XIP stands for X Image Processing and will henceforth be used to refer to the user interface. Since the user interface has a great deal of interaction with different parts of the system and its devices, the details are broken up into appropriate logical subsections. The software system hierarchy is shown in figure 2.1. Notice that although the user interface sits on top of the X server it does not necessarily imply isolation from the underlying system.

2.1. Global functionality of the XIP User Interface

All X related operations are handled through the X server running on that host or the remote host to which the request is made. The C library *Xlib* provides the capability to make requests to the server to allocate/modify/deallocate resources under the control of X. Thus the first category of operations discussed are related to user interaction with the underlying system, followed by the functions of the user interface which are operating system related. Actually since the user interface has been modelled as a distributed system the differentiation between the two is not clearly defined. The lower end of interaction is with the image processing system. The hierarchy of the system apparently dictates placing the image processing system above the OS level. However the application programs run under the UNIX operating system and they operate on image data which is stored in the UNIX file system. Thus the image processing software system lies in the same strata as the file system. Another issue to be addressed is the image data format conversion scheme adopted by the user interface to support portability.

As can be seen in figure 2.2 the XIP user interface has eight modules associated with it. At this level of abstraction a module implies either a process spawned by the main program or a function call. The functionality of each module is summarized in figure 2.2.

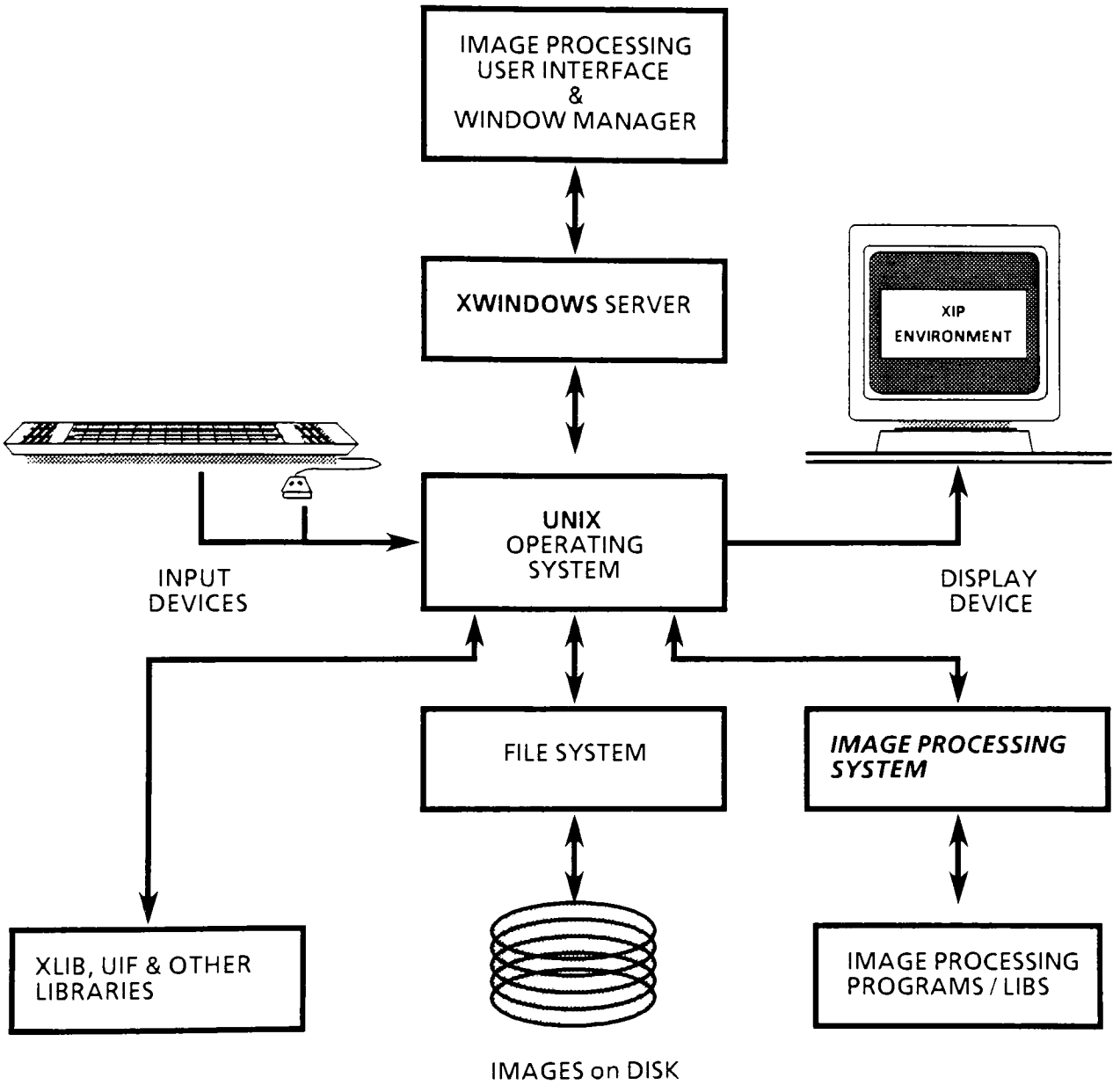


Fig. 2.1 Software System Hierarchy

2.2. Functional Specification of Modules

At the prototype level of each module the description entails the module type, its functional behavior, and its mode of communication with the main process as shown in figure 2.2. A more detailed breakdown of the underlying module interaction is provided in figure 2.5 at the end of the chapter. Notice that all the module names have been abbreviated and have the character X as a prefix. This has been done to be compatible with the client program naming convention in the X environment.

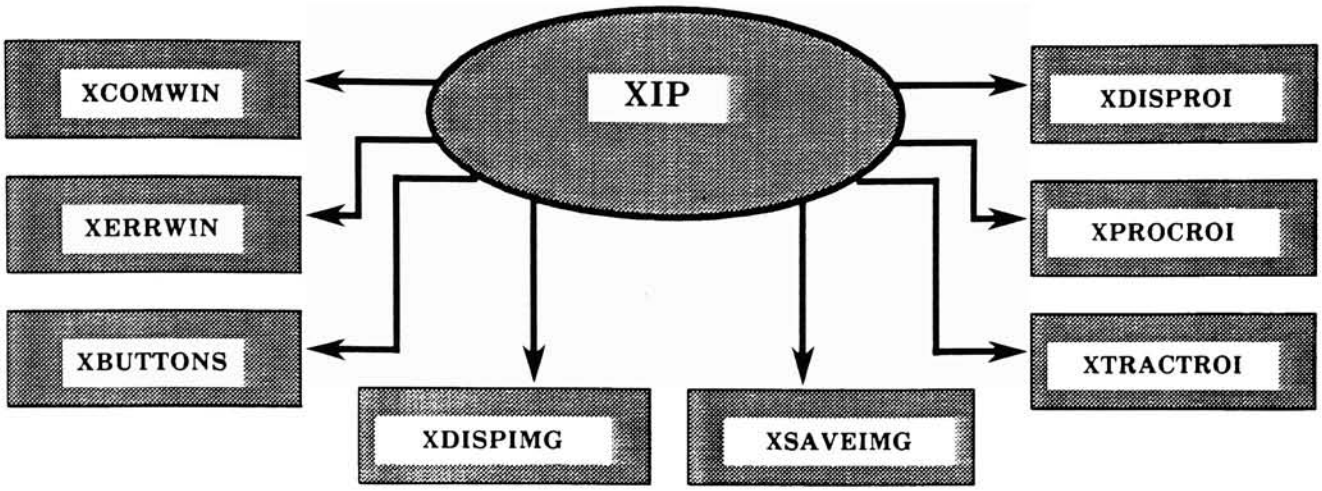
2.2.1. Error Window Daemon (XERRWIN)

Since error messages in UNIX are a source of much strife to the end-user it is critical to provide a separate error handling capability. This is provided by means of a daemon process called *xerrwin*.

Even though we are able to keep track of the error condition in UNIX† it is important to keep all the underlying details transparent from the user. This is easily achieved by using a separate text window to display error messages and any appropriate diagnostic messages. Although this does not guarantee any extra robustness it does allow the end-user to continue without being confused or cause an unclean termination of the user interface.

Since error conditions can arise anywhere and in many numbers, making the error handler a separate process is a clean solution. There is a problem however. It is very possible for more than one process which may have been spawned by the XIP user interface to raise an error condition. A single process, whose communication is handled using the *pipe*‡ system call under UNIX is efficient but prevents more than one process to write to the pipe. The first implementation of *xerrwin* was done using pipes, but was discarded for this reason. The alternative is to use sockets. There are relevant merits and demerits of using pipes versus sockets. Pipes are simply a pair of connected stream sockets [LEF] and hence are limited in comparison. However they are in the author's experience easier to progam with, and more reliable than sockets.

† refer to the *errno(3)* manual page in the UNIX Programmers Reference Manual.
‡ refer to the manual pages on the *pipe(2), read(2), & write(2)* system calls in the UNIX Programmers Reference Manual.



XCOMWIN:
Type: asynchronous process, started by XIP process.
Function: reads keyboard events, echoes or returns formatted line.
Communication: duplex pipes .

XERRWIN:
Type : asynchronous daemon process. started by XIP process.
Function: displays appropriate error code & message in scrolled text window
Communication: spooled queue and message files.

XBUTTONS:
Type: asynchronous process, started by XIP process.
Function: monitors mouse click events in button windows and returns button code.
Communication: pipe .

XDISPIMG:
Type: asynchronous process, started by XIP process.
Function: convert, dither & display a foreign image in Image window.
Communication: parameters passed on command line

XSAVEIMG:
Type: subroutine call made by XIP process.
Function: convert UIF image back to a foreign image and write to disk.
Communication: parameters passed via subroutine call.

XTRACTROI:
Type: subroutine call made by XIP process.
Function: interactively extract Region of Interest from current image, return co-ordinates
Communication: parameters passed via subroutine call.

XPROCROI:
Type: asynchronous process, started by XIP process.
Function: convert UIF image buffer to foreign image , process and return UIF image buffer to be displayed in Region of Interest window.
Communication: duplex pipes .

XDISPROI:
Type: asynchronous process, started by XIP process.
Function: display current region of interest buffer in Region of Interest window.
Communication: parameters passed on command line.

Fig. 2.2 Global Structure of the XIP User-Interface

Given the above implementation problem, the approach adapted is more general. The *xerrwin* daemon process is started up at a high level and runs in the background. Error messages are spooled to a queue, which is a shared resource protected by a simple semaphore mechanism. The window daemon dequeues them reads and displays the error message files, which are subsequently removed. Thus the daemon is not aware of how many processes are enqueueing error messages, and when it has access to the queue it treats all messages in the same fashion.

As a further extension of the error window implementation it is also possible to scroll the error messages in the error window and retrieve old error messages using the mouse. This is useful since error message usually are queued in bursts when a number of functions fail, as a result of the last one raising an error flag.

2.2.2. Command Window Handler (XCOMWIN)

The user interface is a visual interactive environment and hence there is no explicit command line where commands are executed from. A lot of processing depends on user input, and all input cannot be acquired using visual tools like buttons, and scrollbars without a sophisticated toolkit. Hence a command window which prompts the user for input, informs him of any intermediate action, and collects input from the user is vital. This functionality is provided by the command window handler called *xcomwin*.

Under X the keyboard is a device, which generates unique events for each keystroke (under all supported terminal emulators). These events are queued in the X server, and the application program can take appropriate action. It is the application program which has to dequeue and parse the input. These collected keyboard events can be formatted into a command line, and passed along to the calling program/subroutine, while the same can be echoed in the command window. This implementation also provides the application program the choice of ignoring any keyboard input. The command window is scrolled like the error window, but since the implementation of *xcomwin* is a slave terminal‡ the scroll capability is a function of the terminal emulator *xterm(1)*.

‡ see *pty(4)* in the UNIX Reference Manual.

The command window is also a separate process which communicates with the parent process through pipes. Since full-duplex communication is being provided, two pipes are used, one for each direction. The command window handler suffers from a similar problem as the earlier implementation of the error window handler. When a child of the main user interface process spawns the command window handler it causes the parent to hang up while reading or writing to the pipe. This problem however does not directly affect the execution of the XIP user interface.

2.2.3. Button Handling (XBUTTONS)

The primary actions that the XIP User Interface performs are those provided in the form of buttons. In the current implementation eight buttons are supported, each of those having a unique action associated with them. Management of these buttons is handed down to a separate process called *xbuttons*.† Associated with each button is a code. When a button is clicked it indicates a response and returns the code to the parent process via a pipe. The parent process can examine the code and take any necessary action. Thus the button handler provides a clean interface between the user and the controlling process.

The task of adding buttons needs the creation of an icon using a bitmap editor like *bitmap(1)*. The icon files are read in by the button manager which has to create the window and manage input from it. Though this process is simple it does require the recompilation of the user interface software.

2.2.4. Displaying and Saving Images (XDISPIMG & XSAVEIMG)

The primary functions of the user interface are: the ability to display an image in a variable geometry†† window independent of the foreign image format or the type of monitor (monochrome, grey-scale or color) available, and the ability to save the image after all processing has been completed. The modules *xdispimg* and *xsaveimg* carry out these functions.

---

† refer to the *xbuttons(1X)* manual page in the XIP User Interface Reference Manual.

†† the term *geometry* in the X environment refers to the co-ordinates of the origin of the window, and the size of the window. All co-ordinates are in pixels and hence are limited to the size of display device.

The image conversion details are hidden under the image display routine. Note that these routines operate on entire images, as opposed to regions of interest which are subsets of images. The routines that operate on a subset of the image are discussed in the following section. After retrieving an image from disk the image is kept in a UIF image buffer. The buffer is the data-structure which is most frequently accessed and modified and hence its structure is important.‡ The image window has a variable geometry and it takes the size of the image. This is done to prevent visual distortion of the actual image data as well as limiting the region of interest co-ordinates to the image dimensions. Selected regions of interest are, however, enlarged to expose the features of the image sub-section being studied. The scaling operation is examined in the next section.

After all processing has been completed the current image buffer can be saved to disk. The image conversion is now reversed and once again the implementation is taken care of by lower level library routines. In this version of the user interface the cross-conversion between foreign image formats has not been considered but it is a feature that could be easily added.

2.2.5. Region of Interest Processing (XTRACTROI, XDISPROI, & XPROCROI)

This section examines the three region of interest operations performed in the user-interface namely: extraction, display, and processing, which are performed by the *xtractroi, xdisproi, and xprocroi* modules respectively.

A region of interest is defined as a contiguous block of image data which is a subset of the entire image. Once the entire image has been retrieved from disk and displayed in the image window a section of displayed image can be interactively extracted by *xtractroi* from the image window and stored in a separate buffer similar to the image buffer. This buffer could very well be the entire image itself or a small section of it.

Having extracted the region of interest, it is displayed in a separate region- of-interest window, which has a fixed rectangular geometry, by *xdisproi.* The window size is usually fixed at 512x512 and the

‡ see Appendix A for details on the image data structures

extracted image data is resized to approximately that size.‡ The row and column factors are stored in the image header so that the image can be reduced when it is inserted back into the original image after processing. The resizing algorithm used is very simple for reasons of efficiency: pixel replication for zooming and pixel sub-sampling for reduction.

Finally we consider the image processing operations to be performed on the displayed region of interest, by *xprocroi*. In accordance with our design philosophy of transparency, modularity, and portability all image processing operations are carried out external to the user interface by the underlying image processing system. This definitely causes the overall performance to be slowed down, but the benefits of isolating the system are more significant in this design. Every image processing operation requires the conversion of a region of interest to a foreign image format, and the reverse operation after the program has completed. A black-box application, *xconvert* performs this task†.

The pipelining feature of the UNIX operating system is taken advantage of, by specifying concurrent operations to be performed in a pipeline using the same syntax as UNIX tools. To the underlying image processing programs in the pipeline the incoming data appears to be in their resident format which after being processed is fed to a pipe, oblivious of what will happen to it afterwards. The conversion program returns the processed data in the user interface image format to the calling function. The calling function displays the new buffer back on screen. The obvious advantage of adapting such a scheme is that the user interface can now be used with any other image processing system just by adopting its syntax and by porting the conversion program and the image conversion library.

Displaying the region of interest buffer is exactly the same as displaying an image, as the image is converted and written to disk, the display is program executed on that data file. The loss in efficiency due to conversion is evident but is forsaken in favor of modularity. The issue of dealing with color images, color and RGB dithering, look-up tables will be addressed in the following chapters.

‡ Due round-off and truncation error while computing the resized image dimensions.
† refer to *xconvert(IX)* manual page in the XIP User Interface Reference Manual.

2.3. Other functions performed by XIP

The major functions outlined above do broadly cover most of what is needed from the user interface. However there are a few other features which may be performed just as easily by the main process.

One such feature is the ability to **undo** an operation. Since the region of interest buffer is converted to the foreign image format before being processed, a copy of the buffer can be easily saved in a history buffer. Undoing the last operation thus implies displaying the history buffer in the region of interest window. This is a vital feature to iterative processing where human judgement is being applied continuously. The ability to **repeat** an operation is also provided, and it involves storing the previous command pipeline which can be subsequently re-executed on the current region of interest buffer.

2.4. Screen Organization and Management

Along with the underlying functions performed a description of the screen organization and its management is necessary. It has been decided that the user interface use variable geometry windows within fixed limits and to disallow overlapping image windows . This may seem a very restrictive approach, but it should be noted that this user interface is geared towards providing a specific kind of environment which should appear the same every time the program is invoked. An escape from this limiting feature is provided to the expert user.† The user interface is running under the X environment and thus uses the resident window manager *uwm(1)*, for window reorganization and management. This is elegant and separates the task of image processing from window/interface management. The major windows are the image and region of interest windows as well as the error and command windows which are text windows. Other accessory tools can also be used as they are already a part of X. However the more tools the user starts the slower the performance of the user interface will become. The present screen configuration is shown in figure 2.3.

All output data goes to the image windows, all errors are redirected to the error windows, and all user-input is acquired from the command window. According to figure 2.3 processing capability is

† refer to the *xip(1)* manual page in the XIP User Interface Reference Manual.

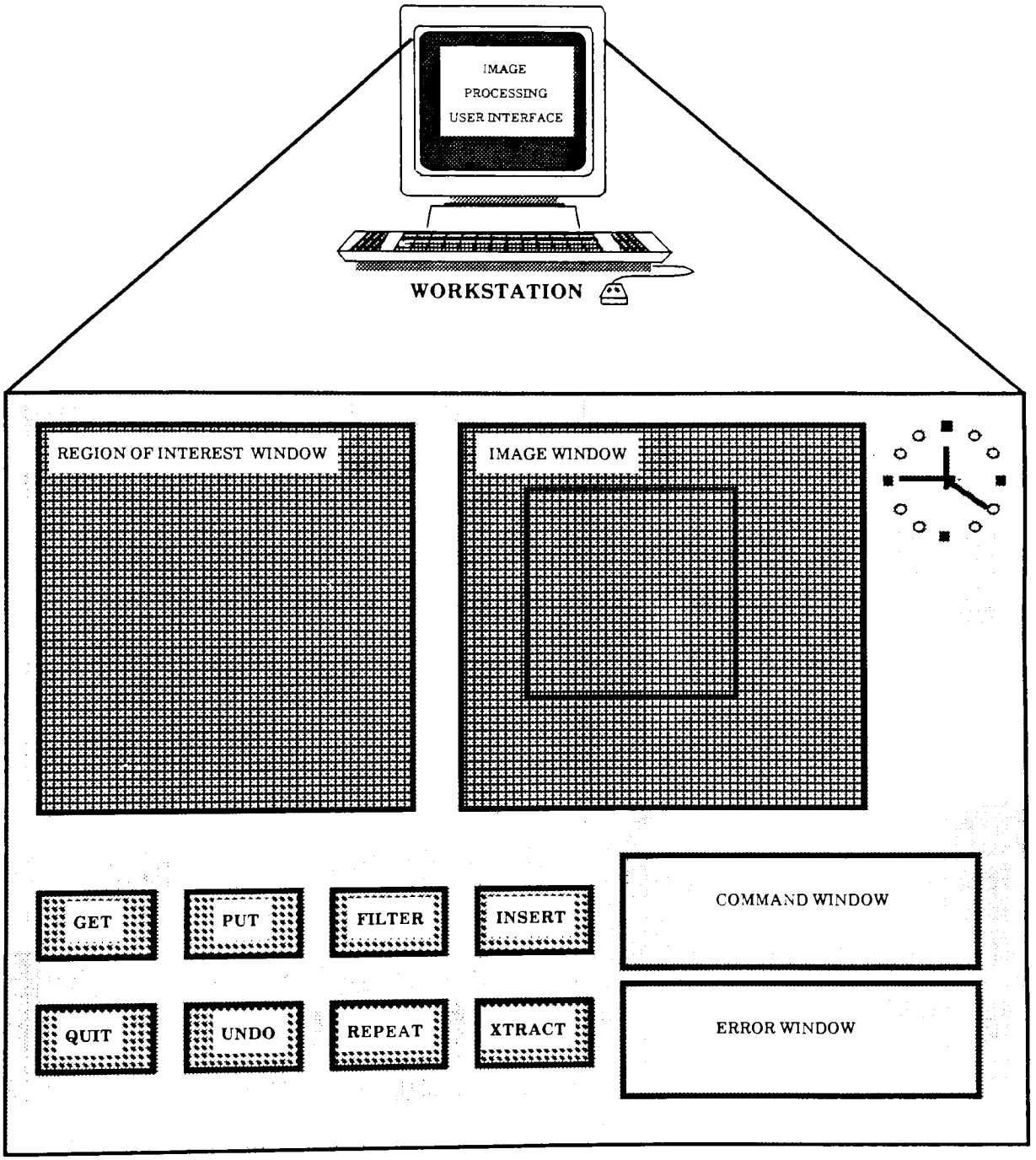


Fig. 2.3 Screen Organization & Management

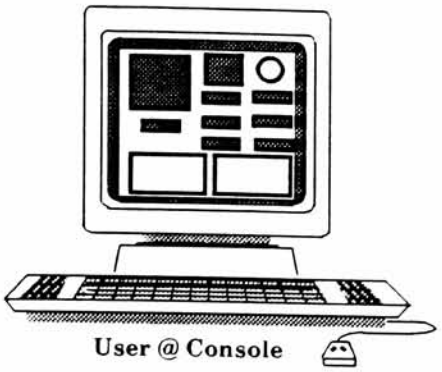
provided using small button windows which take action when selected. The scenario of a button selection where the user interactively selects and enters options is outlined in figure 2.4 and will be discussed in detail in the following chapter.

2.5. Limitations and restrictions in the XIP design

The limitations and restrictions of the current design, are now summarized below. The list may grow or shrink depending on future design changes.

- Image format conversion is performed continuously and slows down the system.
- Error recovery with pipe communication has not been addressed, but the user interface is robust enough to be aborted and restarted.
- Limiting window geometry seems inconvenient but is important to screen organization.
- Entire image cannot be implicitly processed, instead the region of interest is stretched over the entire image window.
- There is some dependency on UNIX utilities.
- Only one image can be processed at a time, which is overcome by switching from *novice* to *expert* level.

The following chapters outline in greater detail the underlying data-structures, algorithms, routines and implementation details of each module of the XIP User Interface



This is a simple example of the sequence of operations a user goes through to process an image. The sequence below illustrates what the user would typically see on the screen.

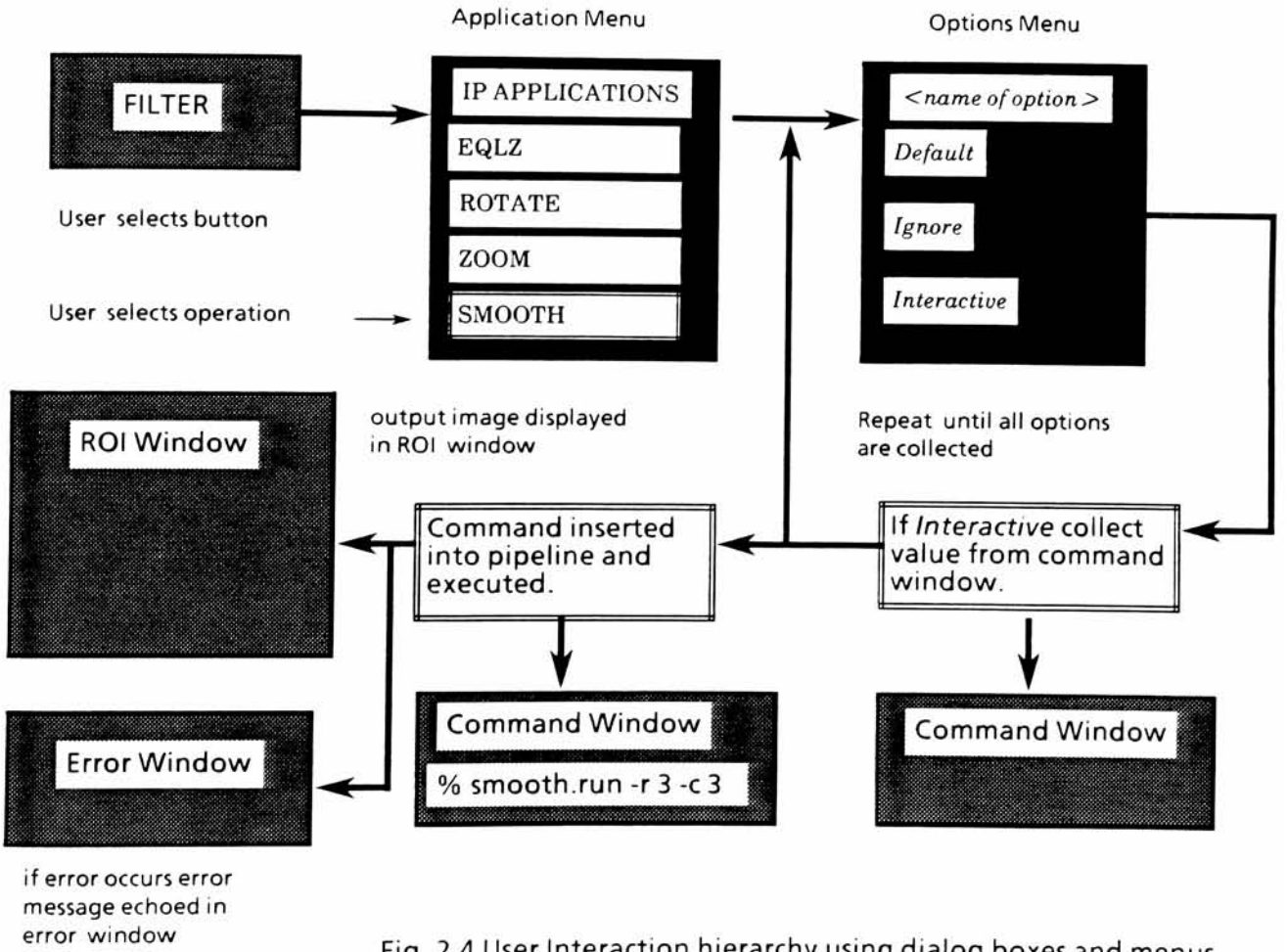


Fig. 2.4 User Interaction hierarchy using dialog boxes and menus

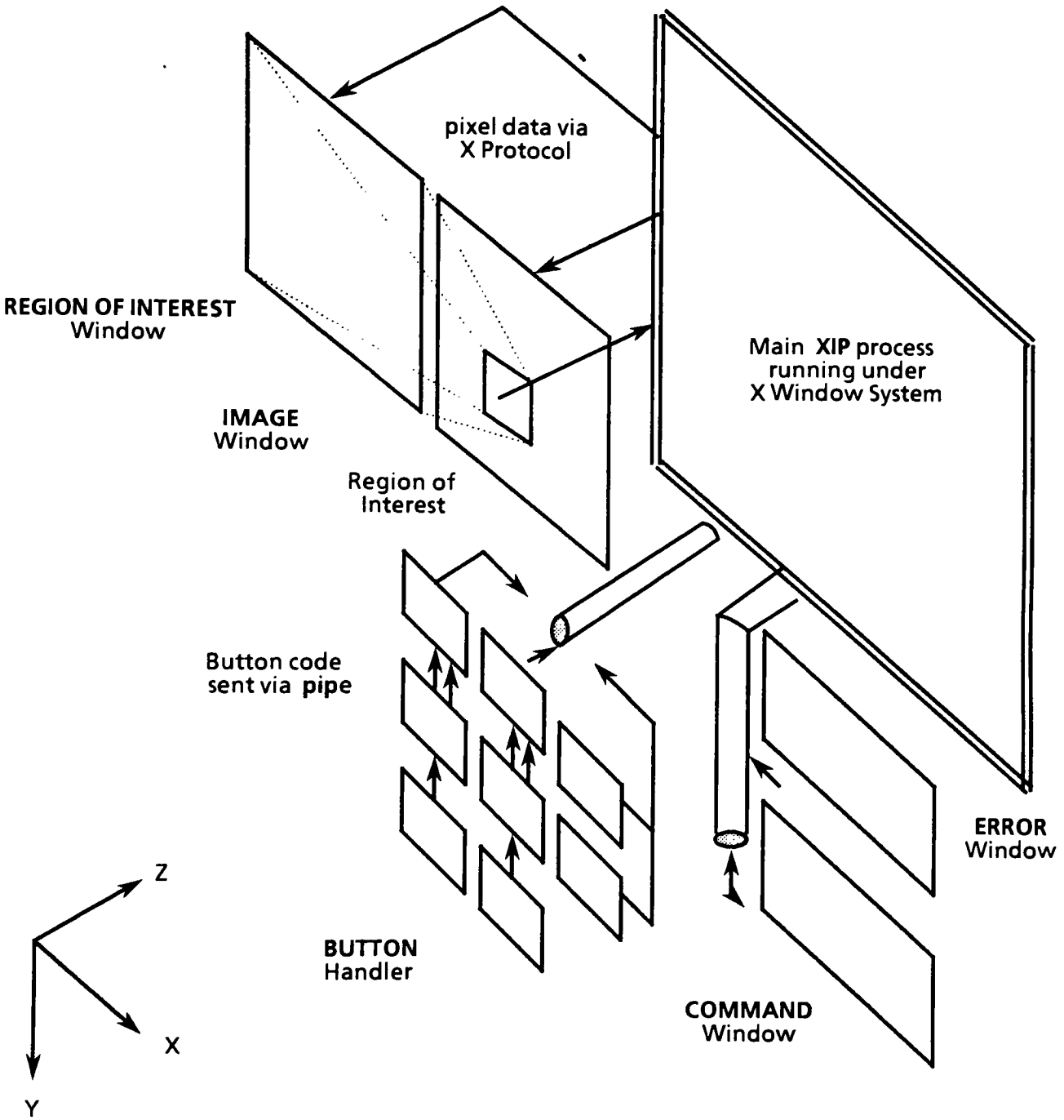


Fig. 2.5 **Front End System Layers and Interaction**

Chapter 3: The Error Window Daemon

3.1. Introduction

An error condition can arise from a system or a subroutine call. It may not be desirable to make the error condition fatal; it is preferable to trace the reason for the error and try to recover gracefully instead of an unclean termination of the user's program. The global variable *errno*† can be used to index into the global error table *sys_nerr* which has a corresponding error message for each valid error code. However there may be error conditions specific to X or to the user interface and these error codes need to be identified separately from the system error codes. X has a function (*Xerror()*) which handles any internal error conditions. In the present implementation, specific errors in the XIP user interface have not been identified, but there are hooks provided to build an error table which can be referenced by the error routine.

3.2. Design

As discussed in the previous chapter, the error message facility needs to be isolated from normal processing. Whenever an error condition is raised the user program/function can choose to print out an error message and either exit or return to its calling function.

The best way to isolate error handling is to use a separate process. Using a separate process automatically implies use of a inter-process communication mechanism to pass down formatted error message buffers. In the previous implementation of the error window handler the *pipe(2)* system call under UNIX was used, however that prevents many client processes from writing to the same server process.

Having discarded the client-server model, the most elegant solution to this problem is to follow the example of the UNIX *spooler-daemon model.* In this model there is a daemon process which runs continuously and waits for messages to be spooled on the queue. An application which has an error condition calls the function *errwin(3X)*, which spools the messages. The queue is a shared resource

† refer to *errno(3)* manual page in the UNIX Programmers Reference Manual

between the spooling process and the daemon, and needs to be accessed in a mutually exclusive fashion. On termination/abortion the client program cleans up the queue, and removes any error message files, to prevent false messages from being displayed when the daemon is started the next time.

3.3. Implementation

The client application program needs to start the error window daemon from a sufficiently high level, usually the first thing to be done in the program. Once started there is no more direct communication between the two. The routine *errwin()*‡ is the interface between the client process and the daemon. When called, it formats the four arguments and writes it out to a temporary message file. The file name is enqueued onto the spool queue. Mutually exclusive access is guaranteed to the queue by using a single semaphore.† The daemon process when started creates a text window and waits for either of three events: queued messages waiting to be displayed, mouse button events in the error window which let the user scroll up and down the history of error messages, or window exposure events.

It has been observed that error messages arise in bursts and are handled as soon they are queued. When there are messages on the queue all of them are removed from the queue and displayed. This has two advantages. It saves the overhead of reading the queue, displaying the first error message, and rewriting the remaining messages id's back to the queue, and it also releases the semaphore (lock file) faster. Of course it implies that it blocks the semaphore while it is reading all the messages, but that is a smaller cost to pay. The main disadvantage of using a single semaphore is that of *lock-step* synchronization. Since we are not dealing with a critical resource, fairness need not be guaranteed.

X event handling is slightly trickier since most calls to verify the existence of an event queued by the server are blocking in nature. This can be dangerous as the user may not ever generate a button or exposure event and hence not see any messages, while a potential fatal condition may have arisen. Before fetching the next event a non-blocking function call has to be made to check if there is an event waiting in the queue. If it is an event the program is expecting, then it takes appropriate action otherwise it ignores it.

‡ see *errwin(3X)* in the XIP User Interface Reference Manual.
† see *semops(3X)* in the XIP User Interface Reference Manual.

Once all the queued messages are printed out, the user can choose to backtrack up through the past error messages. The history list is only 100 messages long after which it wraps around.

3.4. Caveats and suggestions

The main flaw in the asynchronous event handling mechanism is that an extra button event is needed the first time the history list is traversed. This is due to the fact that an extra event is needed to get the attention of the event handling block of code. With an improved C interface library for X, this problem should disappear, using cleaner non-blocking verification.

One feature that can be added to the error window mechanism is the storing of the error message list in a file, which can be subsequently printed out if desired.

When a sufficiently comprehensive list of XIP error messages has been compiled the error message need not be passed as an argument, but referenced from the internal error table.

Chapter 4: The Command Window Handler

4.1. Introduction

The need for a command window to allow controlled user input and output has already been identified. In the case of the command window, a client-server model has been proven to perform efficiently. There are four functions performed by the command window handler: displaying messages, collecting user input, executing a command under the window, and terminating the window handler.

The level of interaction is somewhat primitive and an enhancement could be to run a menu interface under the command window to acquire user input.

4.2. Design & Implementation

The command window handler is a separate process spawned at a high level in the client application program. Once again the inter-process communication is provided by using pipes. Since full-duplex communication is desired, a pair of pipes are used, one for each direction. The pipe descriptors are passed to the child by the parent as command line arguments.

A client program communicates with the command window handler *xcomwin* using the function *cmdwin()*† The protocol is extremely simple, using 4 opcodes for each of the four functions performed in the command window namely: SEND, RECV, EXEC, and QUIT.

Unlike the error window which has only output redirected to it, the command window interface needs to be more sophisticated i.e. look like a terminal interface. Under UNIX there are two kinds of terminal drivers, a general terminal interface *tty* and a pseudo terminal interface *pty*. The *pty* driver provides support for a pair of devices called a *pseudo-terminal*, which is comprised of a *master* and a *slave*. The *slave* is identical to a *tty*, but instead of having a hardware interface (via a port) , the functions of the slave terminal are implemented by another process manipulating the master end. The master side is the client program which creates a *TTYWindow*, and makes requests to the server using the *Xtty*‡

† refer to the *xcomwin(1X)*, and *cmdwin(3X)* manual pages in the XIP User Interface Reference Manual.
‡ refer to the *Xtty(3)* manual page in the X Reference Manual.

library. Since the slave terminal is equivalent to a regular terminal window (*xterm*), the 3 standard file descriptors can be redefined which is useful for running applications in the command window. The client can choose to provide support for the *curses* library, and hence run truly menu driven window applications.

The window geometry properties can be modified by changing the entries for *xcomwin* in the *Xdefaults*† file. It should also be mentioned that the command window is treated by the window manager like any other window and thus can be customized by the user.

4.3. Caveats and Suggestions

The primary problem with using processes and pipe communication is the overhead of the system calls. However, once started the command window response time is the same as any terminal window.

A command window process killed inside a client process and restarted does not always get reinitialized correctly. The alternative to using pipes, is to use shared buffers with a semaphore mechanism.

Finally, the most visible feature which could be upgraded is the interface inside the command window itself. As mentioned above the slave terminal supports the *curses* library and can be used to customize the command window menus.

† The *Xdefaults* file is placed in the home directory.

Chapter 5: The Button Handler

5.1. Introduction

All actions taken by the XIP user interface are directed by the use of buttons. Buttons are small windows which act like physical buttons in that when clicked using a mouse they cause some action to be taken.

The actual implementation of the button-like entity on the screen is done by means of the button handler process *xbuttons*. There are 8 buttons and each of them has a specific name which is indicative of the action to be taken by the application which uses this client program.

5.2. Design & Implementation

The button is a standard *Window* created under X, except that its dimensions are small enough to make it look like an icon. Each window has an icon associated with it, which is created using the X bit-map editor *bitmap(1)*. The icons are used to fill the windows after they have been created and mapped to the screen. The buttons supported are:

GET IMAGE
PUT IMAGE
FILTER IMAGE
INSERT REGION OF INTEREST
QUIT BUTTON
UNDO OPERATION
REPEAT OPERATION
EXTRACT REGION OF INTEREST

When clicked the buttons flash in response. The flashing action is achieved by inverting the pixels and refilling the window. The flash time is controlled by a delay function. This simple mechanism provides the user with an effective response. At the same time as the window is flashed, the *xbuttons* process sends the code associated with the selected button to the parent process via a pipe.

Once the button windows are displayed, the button handler's only task is to manage events associated with those windows. The two events supported are exposure events and mouse button events. Since event handling is the only task to be performed at this stage the program can check for events in

blocking mode. The event handling problem that occurs with the error window daemon is thus avoided. Exposure events result in the window being refreshed. This is important since the entire user-interface operates asynchronously and it is possible that the user will carry on more than one unrelated task with overlapping windows which could cover the buttons. Mouse click events simply result in the button code being written to the pipe.

This combination of flashing buttons and codes is thus effectively used by the controlling client program like *xip* to interface with the user.

5.3. Caveats and Suggestions

The first problem with the button handler is the limiting nature of the program which provides a limited number of buttons that cannot be changed, deleted, or added easily. This is because creation of a new button amounts to the following steps:

- editing an icon file, and creating the icon bitmap,
- including the icon file in the program,
- modifying the program to create, map and handle events in the new window,
- defining and adding a new code for the button and
- modifying the parent client program to respond to the new button.

The above procedure is explicit and should be a trivial programming task. It is important to mention that another approach has been tried in button management using individual processes for each button. Although this separates each button and alleviates the task of creating new buttons slightly it can pose a limitation to the parent process. UNIX processes are allowed to have 20 open descriptors†. Each button requires 2 pipe descriptors, leading to a total of 16 open descriptors in the parent. This prevents the parent application from doing any other non-trivial I/O, hence a single button handler process has been used.

Another aesthetic feature which could be modified is the ability to create colored buttons for color consoles. Since the icons are bitmaps this implies mapping the pixel using a different intensity value. This feature offers benefits for future implementations of the user interface as buttons can then be color coded by class of functions and integrated into panels as sub-windows.

† Under 4.2 BSD Sun Release 3.2.

Chapter 6: Image Conversion and Display.

6.1. Introduction

The XIP User Interface is a visual interactive tool and it is vital to have a generalized, consistent and flexible approach to image conversion and display. In fact displaying images is the most often executed operation by the user interface controller process, *xip*. The first issue to be addressed in this chapter is the image data structures used. It is important to have an internal image format for the user interface since a lot of image conversions are actually taking place, and also to maintain independence from the image processing system the user interface is interfaced with.

Image conversion to and from the user interface should also be dealt with in a simple and generalized fashion to achieve efficiency and portability. Since the image conversion library routines are the only image processing system dependent parts of the user interface, it is important to address them independently from the issues of image format and display.

Finally a section is devoted to operations internal to the image display program: dithering, scaling, look-up table generation, and color.

6.2. Image Format

To be consistent with our philosophy of portability and generality, it is necessary to define a standard image data structure to be used in all application software written for the user interface.

It is commonly known that images have large memory requirements. Depending on the demands placed on image storage/retrieval, image display generation or image processing may vary for example, in an image database storage/retrieval is of prime importance, whereas in a pattern recognition application image generation is more important. It is also obvious that the human eye/brain is extremely quick at feature/data extraction from a 2-dimensional image. On the other hand a computer must proceed in a logical and iterative fashion to do the same. According to Pavlidis[PAV82] all data structures used for images are graphs, where the nodes are pixel values, and the branches connect nodes to neighboring

pixels. He also outlines other graphs (quad trees, binary image trees) and related graph traversal algorithms used for images. The key point to note in Pavlidis' discussion, from the perspective of the data structure used in the user interface, is that in reality most of the graphs are represented as x-y coordinates and their intensity values. Thus the problem is how to optimally access the value of a given pixel. After outlining the details of the image data structure an example is given on how image data access is handled efficiently in the user interface.

In the XIP user interface, image data is represented as a 2-dimensional array of integers with 16 bits of precision. Since 16 bit integers yield 65535 possible intensity values, it is unlikely that any device will need more precision to represent image data. This brings up the question of truncation and round-off errors in computing. However it has been observed that increasing precision does not provide drastic improvement, while it certainly increases the memory/disk-space used as well as the processing time. It is also sufficient that all data be limited to integers. With visual image processing capabilities it is unlikely that any non-integer data will need to be displayed. In the event that it is necessary to do so, the real or complex image data can be converted to images with integer data.

An issue not addressed above is how to store color images. Color images are represented as 3 *frames* of red, green and blue values respectively, and the final display is a superposition of the 3. They can be processed as 3 planes and for display purposes a look-up table can be used to generate pixel values, for color monitors. Although this gives each color 16 bits of resolution which is far more than most devices can display, it is perhaps better than using 8 or 16 bits distributed unequally among 3 colors (3,3,2 or 6,6,4 bits/color) in one frame. When displaying color data the image can be scaled to as many bits as can be displayed. To display color images on grey-scale or monochrome displays or to display grey-scale images on monochrome displays, dithering is performed. This is discussed later on in this chapter.

The ability to just store and retrieve image data is not sufficient. Very frequently it is necessary to use other information about the data being processed. Thus the image structure is separated into image header and image data. The header and image structure are outlined below. Notice that the header has a sub-header which contains the co-ordinates of the region of interest of that image being processed.

The reason for inclusion of the sub-header is best explained by the following scenario.

The user at the console requests a particular foreign image to be displayed in the image window. The foreign image is converted to user interface UIFIMAGE format and stored in an image buffer, which gets displayed. The user may then choose to extract a region of interest from the image and display it in the region of interest window. As mentioned earlier this method of interactively selecting regions of interest is equivalent to zooming the image. This algorithm is discussed in the next chapter. Now we have two buffers, one a subset of the other. This subset is dynamic in the sense that it may change iteratively until the user is satisfied with the result and then the entire image may be saved. The sub-header is needed to store the current co-ordinates of the region of interest extracted from the current image buffer. The user interface also has a capability to undo an operation performed, i.e. it has a history of one event. Assume that the region of interest buffer is being smoothed repeatedly using some constant row and column factors and being displayed. Once the image has been smoothed it may be required to compare the last buffer to the current buffer. Once again the sub-header is needed to store spatial information of the history buffer.

It may seem like overkill to design a whole set of structures just to maintain a few buffers. However they provide tremendous convenience in terms of generality of image formats and ease of converting image buffers continuously between user interface image format and foreign image formats. In the present implementation only two foreign image formats are supported. They use the same library functions, and the task of conversion is easy. Modifying the image structure to handle multiple foreign image formats is relatively trivial since usually the same information is required of most images. Thus the user image structure will remain the same for all user interface applications irrespective of the image it is reading or the image it will write to disk. Access to information in the header and sub-header is also greatly simplified because of the macro capability of the programming language C. The files containing the image structure and macros are listed in Appendix A. Listed below is the user interface UIFIMAGE structure, which can be modified as the user interface expands to handle other image formats.

```
typedef struct user_img_struct {
        UIFHEADER * hdr;        /* pointer to header structure */
```

```
        UIMG_WORD ** image;      /* pointer to image data */
}UIFIMAGE;

typedef struct uif_header_struct {
        UROIHEADER roi;     /* region of interest sub-header */
        FILE *file;     /* source FILE */
        short rows;             /* image dimensions */
        short cols;
        short bits;             /* image resolution */
        short minpixel;         /* min & max intensity values */
        short maxpixel;
        short ncolors;      /* # of colors & frames (1 for monochrome/grey-scale */
        short frames;
}UIFHEADER;

typedef struct uroishdr_struct {
        short lx;               /* region of interest co-ordinates */
        short ly;
        short rx;
        short ry;
}UIFIMAGE;
```

Note that image data is represented by a user-defined type (**typedef**) UIMG_WORD. This is to allow different machines to treat image data in the same fashion. It is a short integer (16 bits), but short integers may vary across various machines and hence UIMG_WORD may be redefined for different hardware without affecting the implementation of the image I/O routines.

Returning to the issue of accessing data in the image structure, the example below shows how data in a 2-dimensional array is usually accessed.

```
UIMG_WORD **image;
/* allocate memory for image and initialize */
for ( i = 0 ; i < rows ; i++ )  /* print out pixel values */
    for ( j = 0 ; j < cols ; j++ )
        fprintf(stderr,"PIXEL VALUE = %d",image[i][j]);
```

If the number of rows and columns in the image are 512 each, then there will be 524,288† total memory references, as each pixel value is accessed by computing a row and a column offset. However, using the scheme below, the total number of memory references is dramatically reduced.

```
UIMG_WORD **image;
UIMG_WORD *imgrow;
/* allocate memory for image */
for ( i = 0 ; i < rows ; i++ ) { /* print out pixel values */
    imgrow = image[i];  /* compute row pointers */
```

† 512*512*2 = 524,288

```
    for ( j = 0 ; j < cols ; j++ )
        fprintf(stderr,"PIXEL VALUE = %d",imgrow[j]);
}
```

In this case there are 512*(512+1) = 262,656 total memory references, i.e. 512*512 for the pixel values and 512 references to assign the row pointers. Note that there is about a 50% saving in time for accessing image data.

6.3. Image Conversion in XIP

As discussed above the user interface is designed to deal with heterogeneous image formats which are transparent to all upper level functions which deal with only one data structure (UIFIMAGE). Therefore a library of subroutines to perform image conversion and other related processing is needed. Conversions are being performed at two stages in the user interface: region of interest processing, and complete image retrieval and storage before and after processing. Figure 6.1 shows that there are two levels at which conversion is performed: application level and subroutine level. The need for application level conversion is because the region of interest processing routine executes image processing commands in a pipeline, feeding one end of the pipeline with the current region of interest buffer and reading off the processed data off the other end. Thus the application level program *xconvert* reads data from standard input and writes converted data to standard output acting like a real filter. The *xconvert* program uses the lower level conversion functions.

Every conversion needs allocation of memory for the image structure, as well as initialization of the fields in the image header structure. The two library routines provided are:

- *fimg_to_uimg()* and *uimg_to_fimg()* to convert images,
- *makeuifimage()* and *makeuifheader()* to allocate UIFIMAGE structures.

Data read into a UIFIMAGE structure can have at most 16 bits of precision. Therefore all image data written out into foreign images will have that precision. These routines read in only the header information from the foreign image relevant to the UIFIMAGE structure. The only additional information determined at conversion time is the maximum and minimum pixel values of the image. Since the data is read in pixel by pixel, determining the two values does not pose a substantial overhead. Foreign

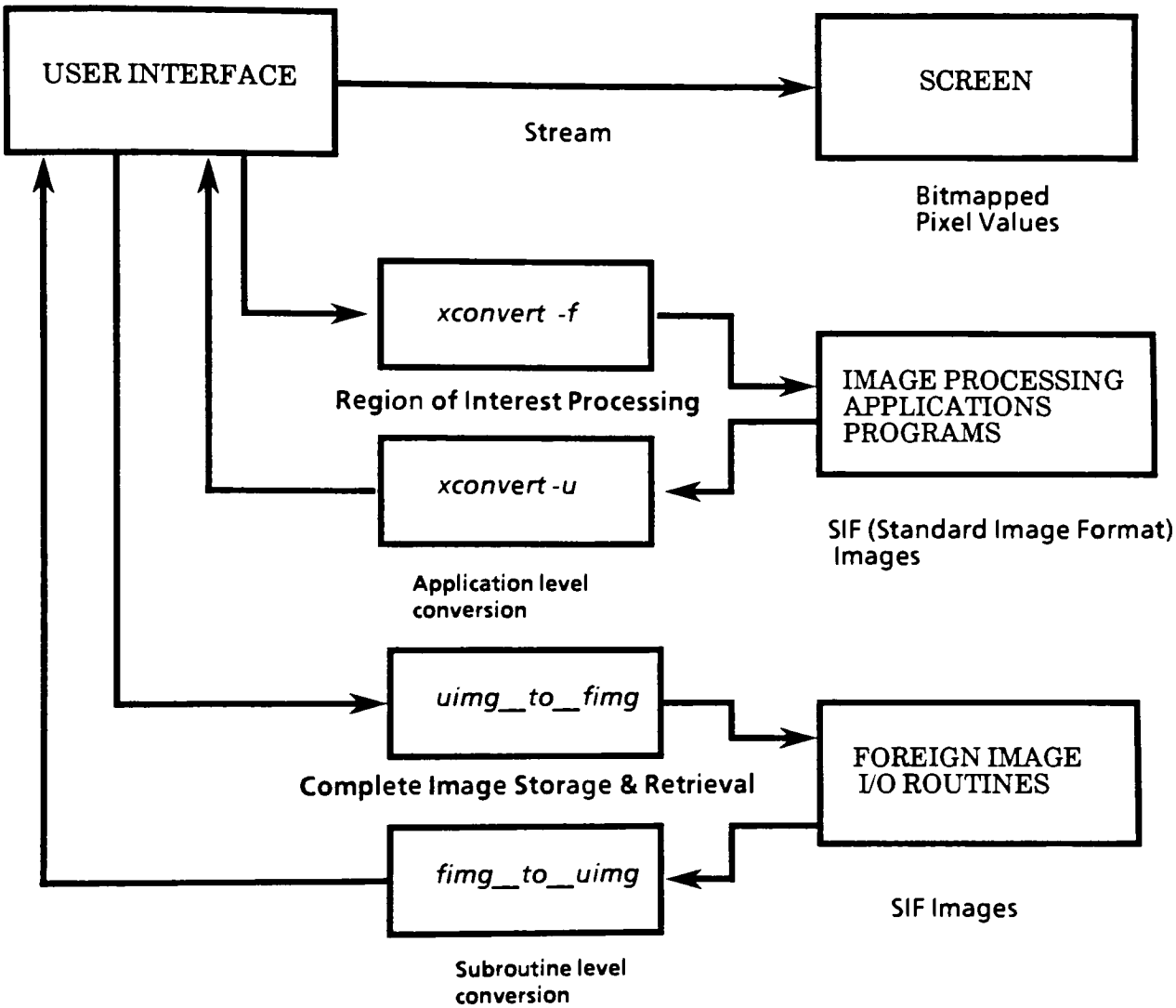


Fig. 6.1 Image I/O between
User Interface and System

images generated from the user-interface may not resemble their predecessors but in fact their contents are of the same type, only their headers and trailers if any, are different.

6.4. Dithering, Color Maps, and Look-up tables (LUT)

Having defined an image structure and converted image data to the internal image format, the data needs to be displayed. Display devices could be monochrome, gray-scale or color. In case of monochrome and color displays, dithering is used. Since the image data available at present is only gray-scale, the pixel value on monochrome displays can only take two values and in the case of color displays a dithering technique is needed to distribute all intensity values across the color map to cause pseudo-coloring. Dithering can be performed by a number of techniques namely ordered dither, constrained average, dynamic threshold, and minimized average error as shown by Jarvis, Judice, and Ninke.[JAR76] A static thresholding technique was initially used to provide a working prototype of the XIP user interface. Thresholding is used for displaying continuous tone images on monochrome (bi-level) displays. In reality using two tone images can obliterate most of the actual information stored in the image. Thresholding was very convenient at the prototype level. The thresholding algorithm is given below.

```
/* define a 50% threshold */
threshold = (max. pixel value - min. pixel value) * 0.5
for ( i = 0 ; i < rows ; i++ )
    for ( j = 0 ; j < cols ; j++ )
        display pixel value = ( pixel value < threshold ? 0 : 255)
```

In the final implementation the user interface is capable of displaying continuous tone images on a monochrome, grey-scale or color displays. The algorithms needed are:

- dither grey-scale images for monochrome monitors,
- transform color images for monochrome monitors and
- colorize grey-scale images for color monitors.

The dithering algorithms differ primarily in the method by which they produce the threshold values for comparison with the picture element intensities. All the algorithms examined by Jarvis et.al. are deterministic and compute the dithered values based on only the current pixel value. From the results presented by Jarvis [JAR74] *ordered dither* was an obvious choice as it was simple to implement and pro-

vides an excellent rendition of the test subjects displayed in his paper. The algorithm generates a bi-level representation of grey-scale images by comparing pixel values to a set of thresholds contained in a n x n dither matrix. The matrix element $D^n{}_{ij}$ is selected and the pixel value $P(x,y)$ generated as follows:

$$\text{if } I_{xy} > D^n{}_{[x\,\%n][y\,\%n]} \quad \text{then } P(x,y) = 255 \text{ (8 bits)}$$
$$\text{else } P(x,y) = 0$$

Thus the dither matrix is repeated over the entire image. The critical factor is the dither matrix itself. Historically the first dither matrix D^2 was proposed by Limb[JAR76] ,

$$D^2 = \begin{bmatrix} 0 & 2 \\ 3 & 1 \end{bmatrix}$$

A n-th order dither matrix can be generated from this 2 X 2 matrix as shown by Jarvis et.al. using the following recursion matrix formula:

$$D^n = \begin{bmatrix} 4D^{n/2}+D^2{}_{00}U^{n/2} & 4D^{n/2}+D^2{}_{01}U^{n/2} \\ 4D^{n/2}+D^2{}_{10}U^{n/2} & 4D^{n/2}+D^2{}_{11}U^{n/2} \end{bmatrix}$$

Dither matrices up to 16x16 have been experimented with, but there is a tradeoff between speed and quality. Results presented by Jarvis et.al [JAR76] show that beyond a 8x8 dither matrices the displayed output is not distinctly different. The objective of dithering is to provide as many grey-levels on monochrome displays. The algorithm chosen for converting color images to monochrome is very simple. It is better known as the NTSC transform and is outlined below. [FOL82]

```
for ( i = 0 ; i < rows ; i++ )
    for ( j = 0 ; j < cols ; j++ )
        BW value = 0.35*RED + 0.55*GREEN + 0.10*BLUE
```

The BW value needs dithering before being displayed. To pseudo-color a grey scale image for color displays a linear or non-linear look-up table can be used, however this does not guarantee aesthetically perfect results. The table used in XIP contains three cycles evenly distributed. X supports a palette of 216 colors at a time, which can be changed dynamically if necessary. Having scaled the data to 8 bits (255 levels) the pixel value is dithered, before the table look-up. The color dither matrix is similar to the one used above except that a maximum value of 215 is necessary to generate a pixel value within the range of the color map. Experimentation in the area of true color involves entering the realm of color science and that is not the objective of the design of this software system.

6.5. Design & Implementation of the Image Display program.

This section covers the few implementation issues not addressed above. The sequence of operations for image display is as follows:

- read in and convert foreign image.
- determine nature of display; generate dither matrix and color look-up tables if needed.
- scale image to 8 bits precision.
- dither, map and display each pixel value in the image. The displayed pixel value is stored in a buffer.
- wait for exposure events and refresh entire window.

The scaling operation uses the maximum and minimum pixel values which are determined at conversion time, and redistributes the intensities over 8 bits of precision.

The overhead computation time required to determine which parts of the image window were overlapped and need refreshing is substantial, and it has been observed that complete refresh works extremely fast and is thus used to update the image. Since the update operation terminates only when the process is killed, it was necessary to make *xdispimg* and *xdisproi* separate processes, which have to be terminated before being executed again.

6.6. Caveats and Suggestions

The display tool has been tested on monochrome and color displays using gray-scale images. Thus testing with other kinds of data and color maps would be required before a final solution is arrived at.

Possible methods of speeding up the program include reading in a static dither matrix from a file, and reading in a static color map from a file. These two operation take up a substantial amount of time. However for experimentation, generation of dither matrices and color maps was found to be more convenient albeit at the cost of efficiency.

Chapter 7: Region of Interest Processing.

7.1. Introduction

region of interest processing includes 3 operations: extraction, processing and display. They are performed by *xtractroi, xprocroi and xdisproi* respectively. Displaying a region of interest is identical to image display which has been discussed in the previous chapter. The issues addressed in this chapter are extraction, processing, resizing and insertion, which take place at a higher level in controlling client process.

It should be noted that both *xtractroi* and *xprocroi* are functions not separate programs. In fact these are the only high level operations of the XIP user interface which are functions.

7.2. Extraction

To extract a region of interest from an image buffer the location of the region of interest in the original image is needed. The location, specified by the upper-left and lower-right co-ordinates of the region of interest, gets returned when selected by the user. The mouse is warped† into the region where the image window is located. Thus the *Window* structure of the image window can be obtained by querying the mouse. The default region of interest appears initially at the top of that window, and the action associated with each mouse button is echoed in the command window.

Once the default box has been resized and moved to the desired section of the image the user clicks the mouse to return the co-ordinates. The co-ordinates are clipped to the window boundaries to prevent erroneous extraction of image data.

With the co-ordinates available it is simple to read in the desired pixel values into a UIFIMAGE structure. The region of interest co-ordinates are stored in the sub-header of the parent image, for insertion after processing. It is important to mention that the image co-ordinates use the lower left corner as the origin whereas X uses the upper left corner as the origin, and thus the y co-ordinates have to be

† *warp*, refers to the act of moving the mouse/cursor to a specified location, not in the context of the same window.

reversed while extracting the data.

7.3. Resizing

Although the image window assumes the dimensions of the image, the region of interest needs to be resized to enhance the feature of the image just extracted. A maximum size of 512x512 pixels is allowed and the image is resized to fit an image into as large a window as possible within the limits imposed. The operation to be performed is a simple zoom. The x and y enlargement factors are determined as shown in the following example:

Region of interest window size = 512 x 512 pixels
Region of interest selected from image window = (10,10) to (110,110)
Region of interest buffer size = 100 x 100
Enlargement factors for region of interest window = 512/100 = 5

The dimensions are not exact because of rounding off caused when the row and column factors are computed. Having obtained the appropriate enlargement factors the zooming algorithm can create the display buffers for the region of interest window. Simple and effective algorithms for zooming and reduction are outlined below.

Zooming (Reduction/Enlargement) algorithm

```
UIFIMAGE ** reduce (image, xfact, yfact)
UIFIMAGE **image;
int xfact, yfact;
{
/* variable declaration & initialization */

/*
*check if reduction factors divide into image size;
*if not, ignore excess rows/columns
*/
    rrows = rows/yfact; /* yfact defines vertical reduction */
    rcols = cols/xfact; /* xfact defines horizontal reduction */
/* create empty image structure */

/* reduction by eliminating every yfact pixel */
    for ( i = 0, k = 0; i < rows; i += yfact, k++) {
        imgrow = image[i];
        rimgrow = rimage[k];
        for ( j = 0, l = 0; j < cols; j += xfact, l++ )
            rimgrow[l] = imgrow[j];
    }
        ...
        return (rimage);
```

```
}

UIFIMAGE ** enlarge (image, xfact, yfact)
UIFIMAGE **image;
int xfact, yfact;
{
/* variable declaration & initialization */

    erows = rows*yfact; /* yfact defines vertical expansion */
    ecols = cols*xfact; /* xfact defines horizontal expansion */

    for (i = 0, k = 0; i < rows; i++) {
        imgrow = image[i];
        for (rowcnt = 0; rowcnt < yfact; rowcnt++, k++) {
            eimgrow = eimage[k];
            for (j = 0, l = 0; j < cols; j++) {
                for (colcnt = 0; colcnt < xfact ; colcnt++, l++)
                    eimgrow[l] = imgrow[j];
            }
        }
    }
    return (eimage);
}
```

The reduction algorithm is applied when the processed region of interest buffer needs to be inserted into the original image. Region of interest insertion is discussed in the following section.

7.4. Insertion

When the region of interest is resized, the computed row and column factors are saved as they are needed to reduce the processed region of interest before insertion into the parent image.

Meanwhile if the parent image has been changed it is possible to "paste" this region of interest into the new image. This *cut and paste* feature has been retained in the user interface to allow flexibility in manipulating image data if desired by the user.

The image reduction algorithm is outlined above and the factors used are the same as those used for zooming the image. Once again the insertion function has to recognize the fact that the origin of the image and the co-ordinates are not the same and make the appropriate correction.

7.5. Processing

Region of interest processing is the most computationally intensive of all the operation and hence

the slowest. It has to be able to construct the command syntax of the image processing application to be executed on the region of interest buffer, and process it transparent to the application. The *xprocroi* function can be divided into two logical subsets, command line generation and image processing and retrieval.

There are two levels of expertise allowed in the XIP user interface : *novice* and *pro*. The levels are set by using environment variable† XIP_LEVEL. If the variable is set to *novice xprocroi* executes the menu tool *xipmenu*‡ The menu tools interactively build a command line which is inserted into the pipeline:

xconvert -f | <*application name*> <*options*> <*arguments*> | xconvert -u

which is then executed. The standard descriptors of the command pipeline have been duplicated with two pipe descriptors. The UIFIMAGE buffer is written to the front end of the pipeline and read off the back end after all data has been processed. Although, elegant and simple, the overhead of writing, conversion, processing and reading the image back is substantially high. It does however, isolate the image processing application program completely from the user interface. The *xconvert* programs provide the interface required to achieve that isolation by taking care of the necessary image conversion.

7.6. Caveats and Suggestions

The primary cause of delays in the processing is due to the pipe I/O overhead and is the cost to be paid for such a degree of independence.

Strict parent child relationships are not verified at the time of insertion, however this can be easily imposed if it is desired. It would involve maintaining a threaded list of images and verification if a processed buffer belongs to the image where it is being inserted.

In terms of visual improvement, thicker lines have been tried with X Version 10, however they are not recommended due to the limitations of *Xlib*. A feature which could be added is the ability to grid an image window which would make region of interest selection much easier at the cost of slowing

† see *setenv(2)* in the UNIX Programmer's Reference Manual.
‡ see *xipmenu(1X)* in the XIP User Interface Reference Manual

down the extraction process.

It should also be possible to print the actual co-ordinates of the region of interest in a sub-window of the Region of Interest window, this would inform the user of the actual data which is being displayed/processed.

Chapter 8: Menu Building and Generation

8.1. Introduction

When operating at the *novice* level the XIP user interface uses an interactive menu system to collect all the parameters and options needed to execute a selected image processing application on the current region of interest.

This action is executed by the menu tool *xipmenu*. As we are dealing with a large range of input values for any application program there is a limitation on the amount of high-level interaction that can be performed. The user has to enter the value via the command window, if the current default values are not acceptable.

8.2. Design and Implementation

It is not practical to build all the information about an application program into the user interface at compilation time as it leads to inherent dependencies on the underlying image processing system. The alternative is to gather the information at run time, and use that with a less cryptic method to prompt a user with the possible options.

X provides an elegant solution to this problem. The function *XGetDefault()* provides the ability to determine the value of a named parameter of a given program at run time. The options are stored and read from a file *~/.Xdefaults*† in the user's home directory and returned to the calling program.

The menu tool borrows from the same concept storing the formatted information for all image processing programs called by the user interface in the *~/.xipdefs* file. The image processing options are needed often since the programs could be invoked many times. Just storing parameter values is rather wasteful. However it is possible to store the options, a default value for each option and a range of values acceptable for each option. This scheme, although lengthy, provides flexibility and time saved in the long run. Adding a new application program *equalize* implies adding the following information to

† refer to the X(1) manual page in the X Reference Manual, or the Xlib document.

the ~/*xipdefs* file.

```
equalize.name:      eqlz.run
equalize.opts:      3
equalize.1:         -m
equalize.1.name:     mapping
equalize.1.prereq:   NONE
equalize.1.check:    YES
equalize.1.def:      NONE
equalize.2.range:    cumul, mean, rand, neigh, hyper
equalize.2:         -n
equalize.2.name:     neighborhood
equalize.2.prereq:   -m neigh
equalize.2.check:    NO
equalize.2.def:      1
equalize.2.range:    1,4
equalize.3:         -s
equalize.3.name:     slope
equalize.3.prereq:   -m hyper
equalize.3.check:    NO
equalize.3.def:     -0.5
equalize.3.range:    -0.5,0.5
```

8.2.1. Tokens and reading the default file

The specifications above seem very voluminous for a single application program. They are just an example of the many instances of the tokens which are described below with their associated text.

```
<program>.name       - name of the application program
<program>.opts       - total number of options in the application program

<program>.i          option ( number range from 1 onwards )
<program>.i.name       option name, needed for menu/scroll-bar
<program>.i.prereq     pre-requisite option to be set for this option
<program>.i.check      check if option is mandatory on the command line
<program>.i.def        default value of this option
<program>.i.range    - range of values for the option
```

A program can have *i* options each of which has the *name, prereq, check, def, and range* fields. Tokens are associated with each of the fields and returned by the lexical analyser† along with the following text. Using the tokens and associated text, a two-dimensional linked list is built as shown below. This list consists of menus and an option list attached to each menu. The structures for the menu lists are defined in Appendix B. Access to the various fields in the structures are provided by macros in the same manner as the image display macros.

† refer to the lex(1) manual page in the UNIX Programmer's Reference Manual.

The list is returned to the calling routine which can traverse, modify, or destroy the list.

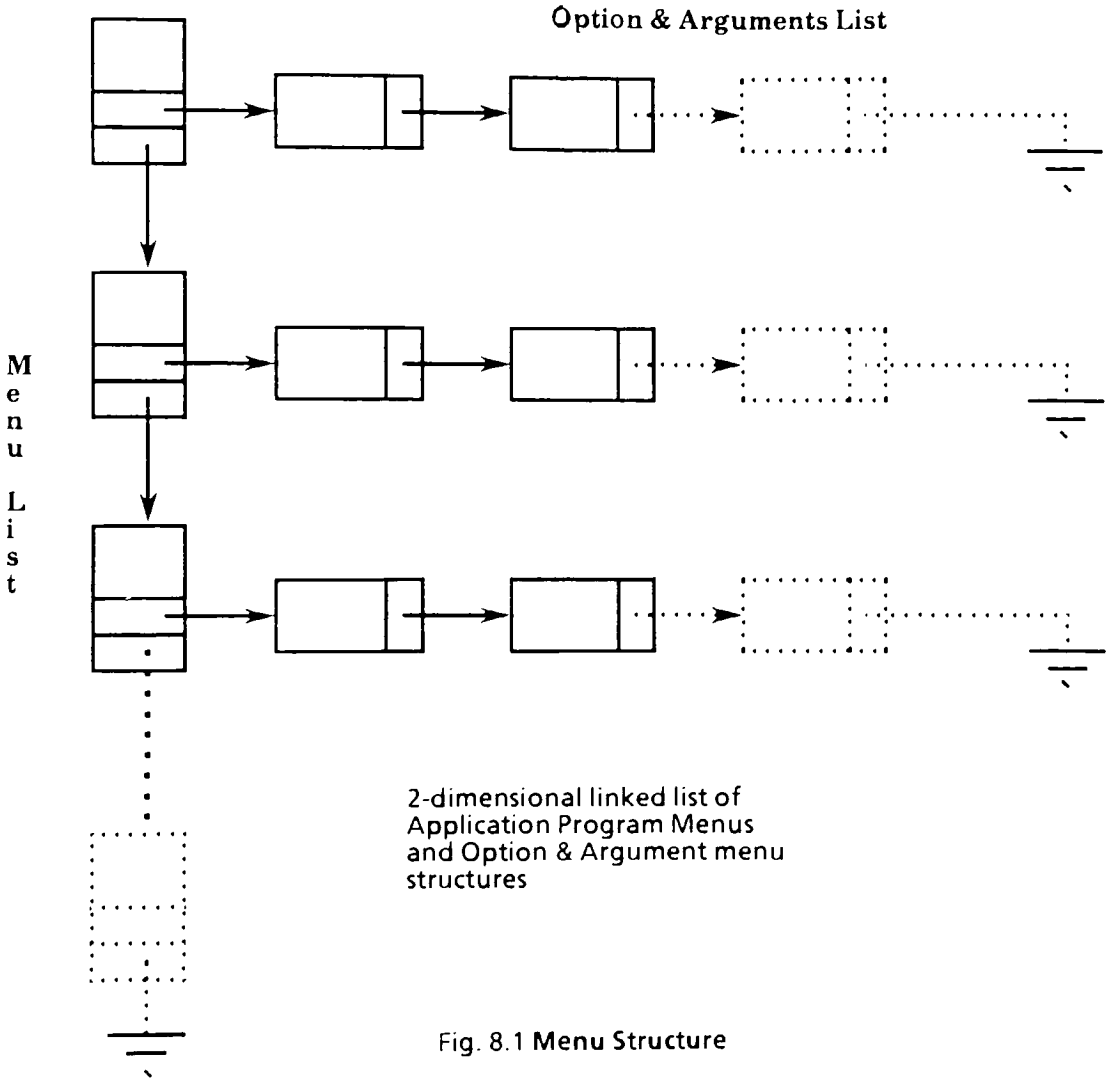


Fig. 8.1 **Menu Structure**

8.2.2. Menu Generation and Command Building

The *XMenu* deck of cards system under X provides client programs with a library of functions to build multi-pane, multi-selection menus. From the information in the menu list, menu panes and selections are constructed. Referring back to figure 2.4, there are two basic panes in the *xipmenu* tool: the application menu and the options menu. The application menu is specific and it contains a list of all possible applications which are in the menu list. Selections are labelled with the name of each application. The options menu on the other hand is generic. The reason for making a generic options menu is because an application can have potentially endless options each of which having a very large range of valid values. Using scroll bars or other higher level graphic items would allow the use of specific panes for each option, but it is more efficient to relabel a generic menu for each option, since the actions allowed in an options menu are always the same: *Default, Ignore* or *Interactive*. The *Default* and *Ignore* options are self-explanatory. The *Default* option uses the default value associated with that option, *Ignore* option skips that option if it is not mandatory. The *Interactive* option results in the user being prompted in the command window for the input values needed for that option.

Thus the generic menu is relabeled and its dependencies recomputed for each option saving the time needed to create new panes. Once the application is selected the associated options need to be examined. For each selection/action, the associated data field is inserted into a formatted UNIX command line which is written to standard output. The parent process which starts the menu tool reads that formatted command line and inserts it into a pipeline of chained commands as explained in the previous chapters.

8.3. Caveats and Suggestions

The lexical analyser is the most syntax sensitive (least robust) of all modules in the *xipmenu* tool. Using wrong syntax in the *~/.xipdefs* file results in a incorrect menu list being built. The cost of building in syntax verification was more than necessary for a menu generation tool. Thus an incorrect defaults file could result in the program aborting if a list could not be built or an erroneous list built which will show up only when the menu panes are displayed.

The menu tool could be a function instead of a program but using it as a stand-alone program is useful since it can be run inside of programs, shell-scripts or as an instructional tool to the user learning to use the XIP user interface.

In the present implementation the list is built every time the client program is executed; it may be desirable to store the list once the user interface is running and update the list by moving items last accessed to the head of the list. This *drift-ahead* policy can reduce list traversal substantially as menu items used most often will tend to stay up front.

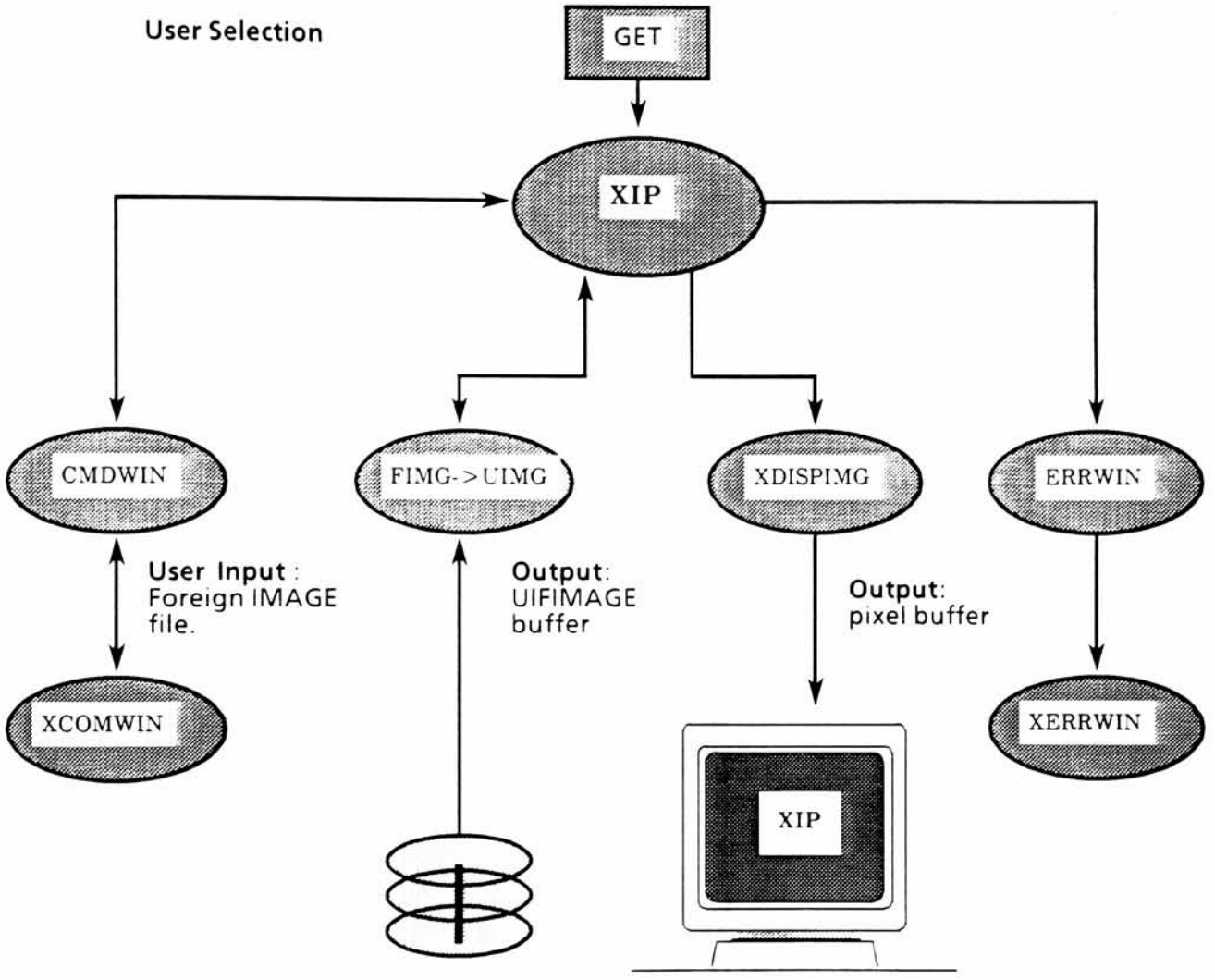


Fig 8.1 **Get Image Operation**

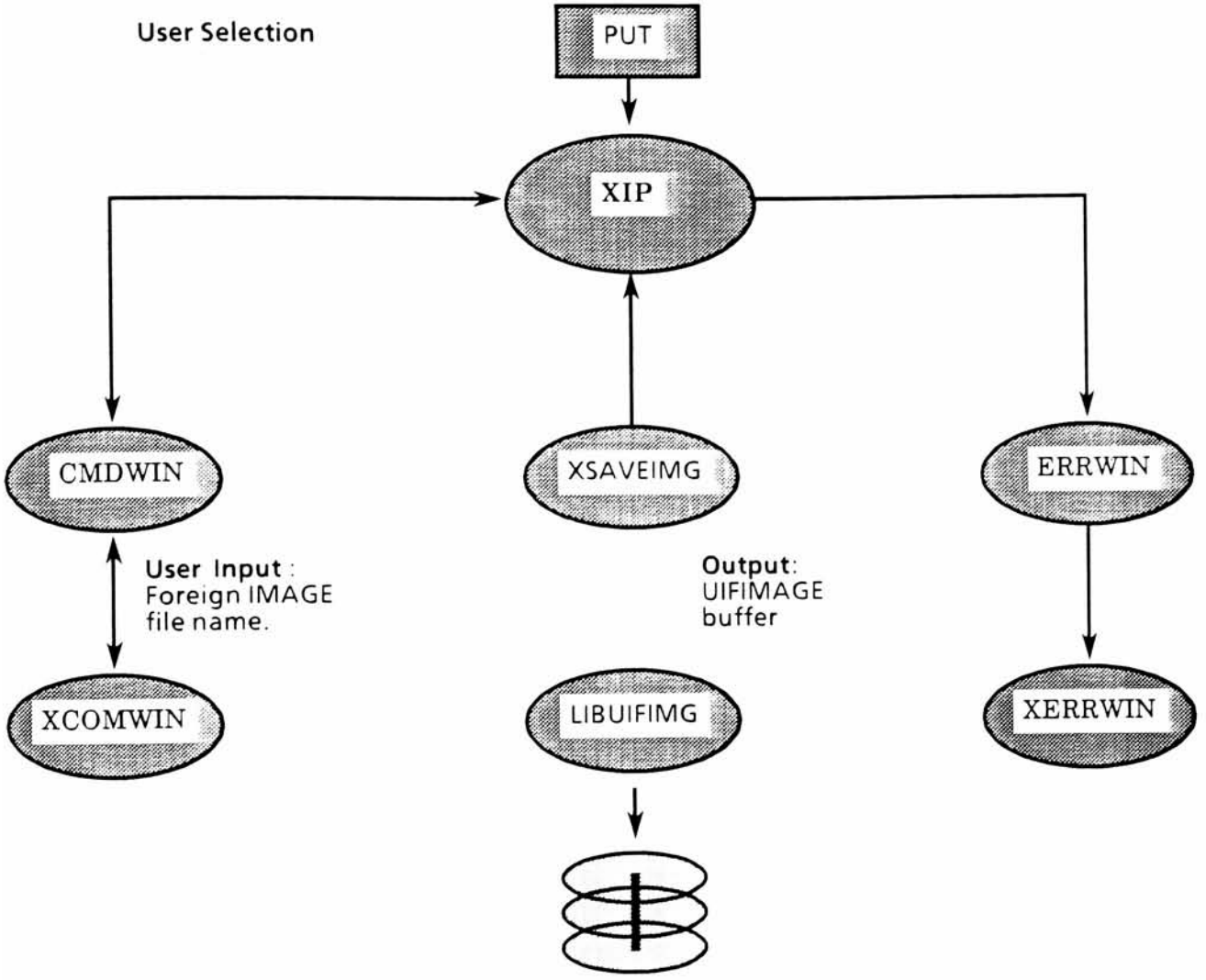


Fig 8.2 **Put Image Operation**

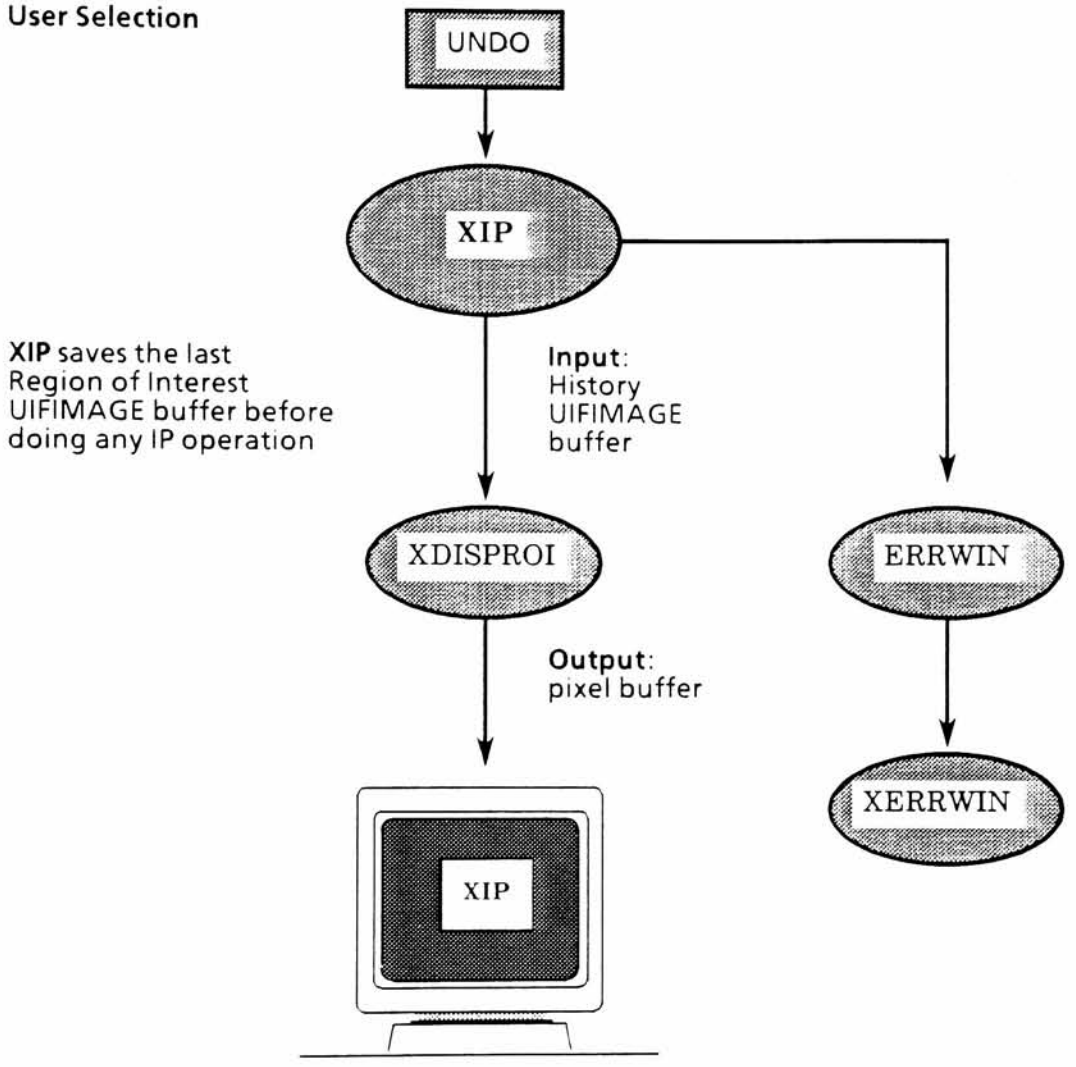


Fig 8.3 **Undo Last Operation**

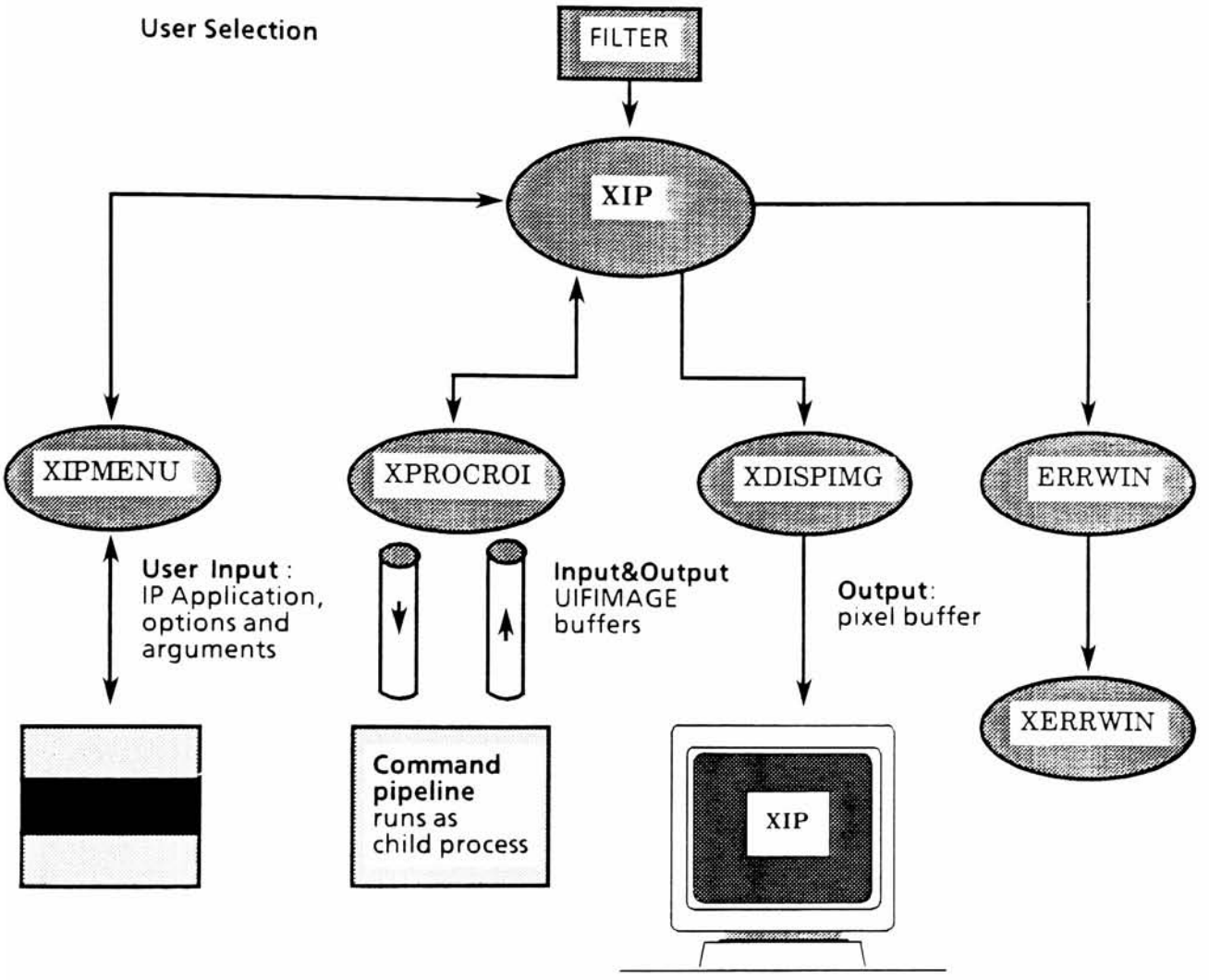


Fig 8.4 **IP Filter Operation**

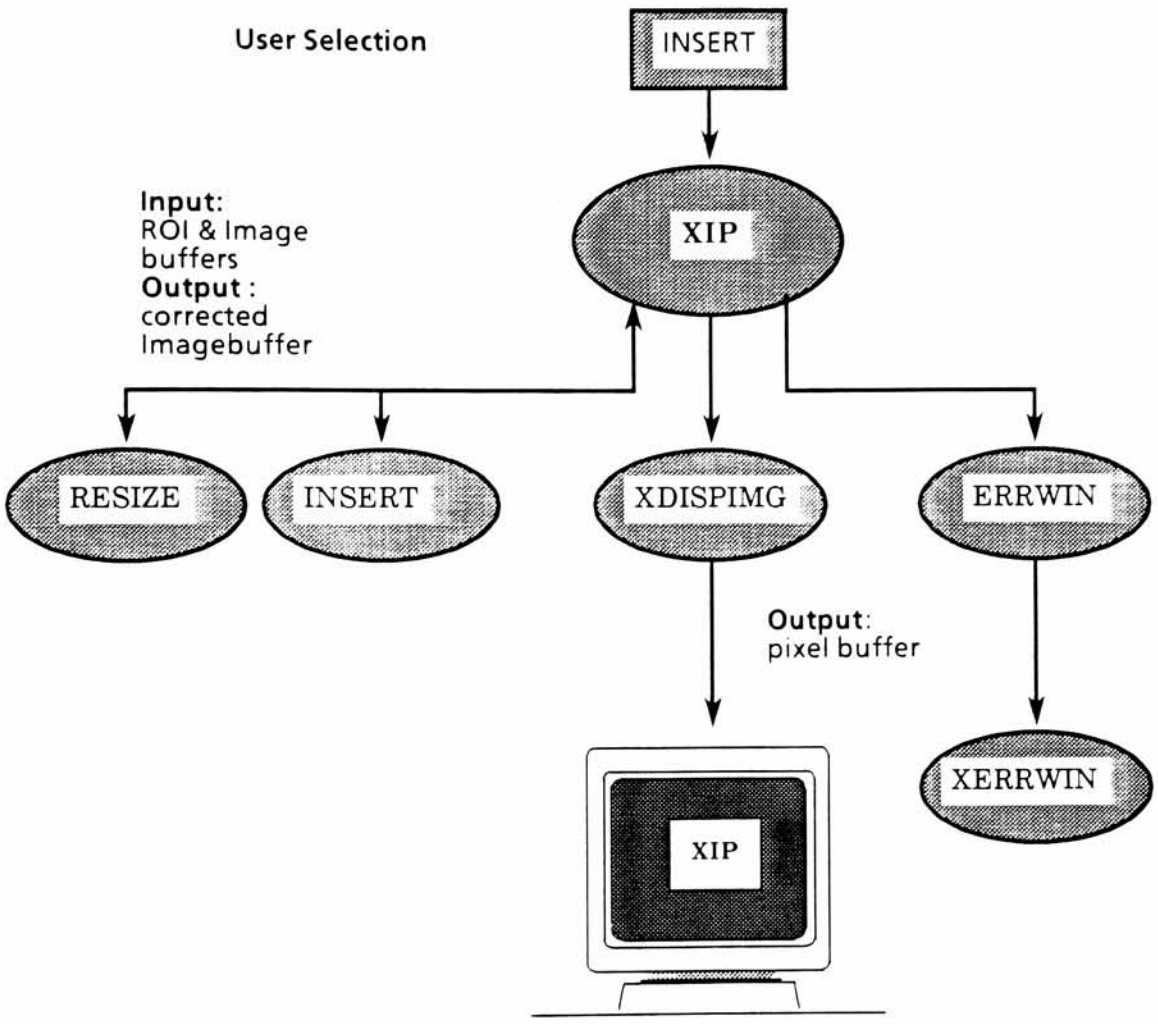


Fig 8.5 **Insert Region of Interest Operation**

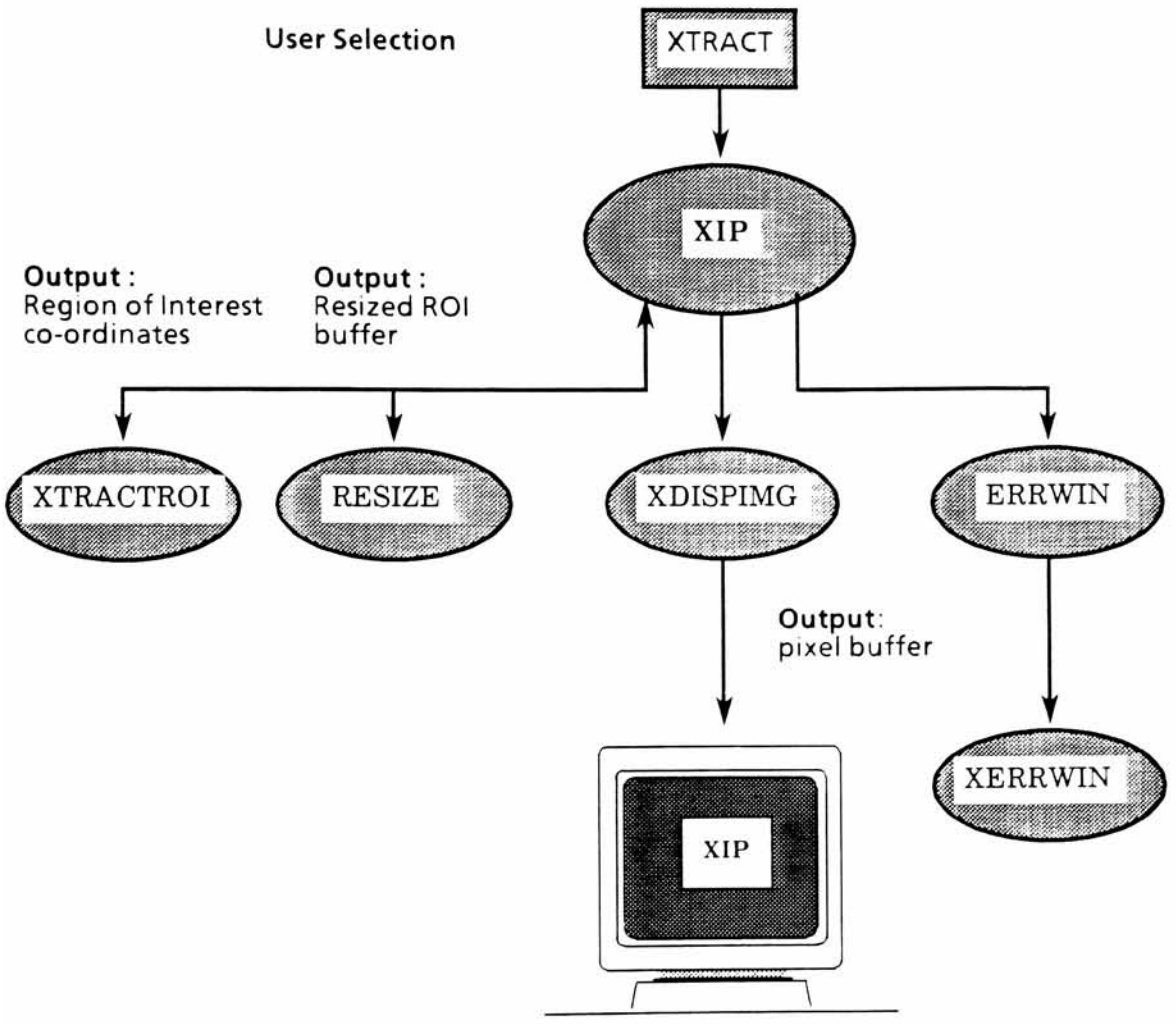


Fig 8.6 **Extract Region of Interest Operation**

Chapter 9: Operation and Results

9.1. Introduction

The previous chapters have outlined the design and implementation of the modules which comprise the XIP User Interface. The philosophy emphasised in the early chapters was to modularize, and isolate all the functionally different parts of the software. This exercise has not just been one in graphics and image processing but also in software engineering. In the demands for faster processing many of the cardinal rules of software development are broken, and it has been one of the goals of this project to exemplify those rules and at the same time provide a working software system which meets the specifications outlined in the first chapter.

The following sections examine the top level control and operation of the XIP User Interface, as well as provide results of the software system in execution.

9.2. XIP User Interface Control

Dividing the various tasks performed in the user interface to conceptually lower level programs/routine provides the controller process *xip* with a lot of flexibility and also makes it very easy to write and modify.

The dialog between the user and *xip* is carried out using the *xbuttons* button handler and the *xcomwin* command window handler. Each button causes an action to be taken and these are described pictorially, in chapter 8.

This reduces the main program to starting the daemons, handlers etc., initializing the trap routine, and waiting for button codes and taking action for each code, making the task of adding new buttons to the user interface very easy.

It should be mentioned here that the functions to extract, resize, insert and process region of interest are linked and called at this level. Thus there are two levels over which the operation of the user interface is distributed: the main control level and the underlying client application level.

Termination of the user interface causes all temporary files, error message files, and error message spooler queue to be deleted. It also results in the termination of all the initially spawned sub-processes. The user can exit the user interface using the QUIT button or by generating an interrupt in the window where the user interface was started. The interrupt (ctrl-C) results in the same effect as the QUIT button, but is an escape when the user interface gets blocked while processing due to abnormal conditions.

Session logs can be maintained by simply redirecting the output from the user interface to a file, or alternatively modifying the main process to create and write a log file every time the user interface is invoked.

9.3. Results

It is hard to demonstrate the success of an interactive visual system in a document, however the figures in Appendix D attempt to show the various stages of execution. Figures S1.1 to S1.6 show the various stages in a session of the XIP user interface. A photographic image is used to show the imaging capability of the user interface. All output was displayed on a 1152x900 monochrome and color consoles of a SUN 3/160 workstation. The SUN 3/160 is a MC 68020 based, 16 Mhz machine. Appendix E lists selected statistics of certain modules in execution to demonstrate the actual efficiency of the user interface.

Another set of examples is provided in Figures S2.1 to S2.3 where the use of the *xipmenu* tool is demonstrated. Finally there are a few images in monochrome and gray-scale to demonstrate the efficacy of the dithering technique used in the XIP user interface.

9.4. Conclusions

Each of the preceding chapters has had a section for suggestions specific to that part of the software system. This section only seeks to summarize the overall performance and to scope for improvement or further work on this software.

As mentioned at numerous instances throughout this document, speed has been the main drawback of the entire system. This may be a result of overdesigning the entire system or just abiding by the

rules of software engineering and sacrificing efficiency whenever there was a conflict between *programmer efficiency* and *program efficiency*. Speeding up of the display tools, and the region of interest processing module would cause substantial improvement.

At run time there are a few drawbacks that should be mentioned here. Image processing is very memory intensive thus it is very possible for the user interface to run out of disk space. Since temporary files have randomly generated names, it is hard to determine whether the file is in use or not However a polling routine could clean up temporary files once in a while.

A high load on the system automatically results in poorer performance. This user interface is targeted at single user systems and works effectively in that environment.

In terms of design extensions to the user interface, the most important is the addition of color image handling. Since color images are multiple frames, the extension is trivial. The other enhancement is the addition of multiple image formats. This could be achieved by using conversion programs to convert all foreign images to the current foreign image format supported by the user interface, or alternatively add conversion routines for each new format supported by the user interface. The former approach is recommended.

Finally, the use of more sophisticated graphics and icons would result in more transparent operation of the user interface.

Appendix A: XIP User-Interface header files

- xip_img.h ---------- image definitions.
- xip_img_macros.h --------- image macro definitions.
- xip_macros.h ---------- xip macro definitions

xip_img.h

```
/*
 *      FILE :          xip_img.h
 *      AUTHOR :        Sandeep Mehta, Optical Engg., LLE
 *      DATE:           07 /17 /1987
 *      PURPOSE:        header definitions for user interface images
 *                      image format (UIF)
 */

/* The SIF image consists of a header, & any number of sub-headers and IMAGE
 * pointers. Most of the header information in the SIF image is not relevant
 * to the user-interface image format (UIF), and hence the structure is
 * relatively simple. The image information available in the IMAGE is extracted,
 * saved, or manipulated by a set of subroutines & macros declared in
 * xip_img_funcs.h & xip_img_macros.h.
 */

/* The UIF image will be capable of handling color images using multiple frames
 * for the R, G, & B components. In case of B&W images there is a single plane.
 */

typedef short UIMG_WORD ;

/* The following is a definition of the user image region of interest
 * sub-header structure. The only information currently needed is region
 * of interest boundary co-ordinates.
 */
typedef struct uroishdr_struct {
        short lx;           /* upper left co-ordinates of region of interest */
        short ly;
        short rx;           /* lower right co-ordinates of region of interest */
        short ry;
} UROISHDR;

/* The following is a definition of the user image header structure.
 * The header currently has only one sub-header. The other information
 * is extracted from the SIF image, & is useful in image format conversion etc.
 */
typedef struct uifheader_struct {

        UROISHDR roi;           /* UIF image region of interest dimensions */
        FILE *file;             /* foreign image file */
        short rows;             /* SIF image dimensions */
        short cols;
        short bits;             /* SIF image resolution bits /pixel */
        short minpixel;         /* min, & max pixel values are useful */
        short maxpixel;
        short ncolors;          /* number of colors in image (= # frames) */
        short frames;           /* number of frames e.g. 3 for RGB */
} UIFHEADER;

/* This is the final user-interface image format structure. It has only one
 * header at present. Alterations will be needed later for complex images etc.
 */
typedef struct user_img_struct {
        UIFHEADER *hdr;
        UIMG_WORD **image;
} UIFIMAGE;
```

xip_img_macros.h

```
# define xip_putminpix(A,B) ((A)-> hdr-> minpixel = (B))
 /*
  * "write to " macros when only header is available
  * /
# define hdr_putname(A,B)   ((A)-> file = (B))
# define hdr_putrows(A,B)   ((A)-> rows = (B))
# define hdr_putcols(A,B)   ((A)-> cols = (B))
# define hdr_putbits(A,B)   ((A)-> bits = (B))
# define hdr_putcolors(A,B) ((A)-> ncolors = (B))
# define hdr_putframes(A,B) ((A)-> frames = (B))
# define hdr_putroi(A,B)    ((A)-> roi = (B))
# define hdr_putroilx(A,B) ((A)-> roi.lx = (B))
# define hdr_putroily(A,B) ((A)-> roi.ly = (B))
# define hdr_putroirx(A,B) ((A)-> roi.rx = (B))
# define hdr_putroiry(A,B) ((A)-> roi.ry = (B))
# define hdr_putmaxpix(A,B) ((A)-> maxpixel = (B))
# define hdr_putminpix(A,B) ((A)-> minpixel = (B))
```

```
/*
 *        FILE .              xip_macros.h
 *        AUTHOR .            Sandeep Mehta
 *        DATE:               07 /14 /1987
 *        PURPOSE:            general purpose macros useful with xip & associated
 *                            programs.
 */

/*
 * Useful string macros
 */
#define strneq(a,b) (strncmp ((a), (b), strlen(a)) == 0)
#define streq(a,b) (strcmp ((a),(b)) == 0)

/*
 * Useful argument vector macros
 */
#define isopt(optname) (strneq(*argv, optname))              /* does partial matching */
#define hasarg ( ( *(argv+ 1) != (char *) NULL) && (*(argv+ 1)[0] != '-') )
#define getarg (*++argv)

/*
 * Other useful macros
 */
#define bitmask(x) ( 1 << (x))    /* macro to create mask by left shifts */
#define MAX(x,y)  ( (x) > (y) ? (x) : (y) ) /* return max of x or y */
#define INTBITS 32                  /* number of bits in an integer */
```

Appendix B: XIPMENU Menu Structure Definitions

- xipmenu_defs.h ---------- menu definitions.
- xipmenu_macros.h ---------- menu macros.

xipmenu_defs.h

```
/*
 *      FILE :          globals.c
 *      AUTHOR :        Sandeep Mehta, Optical Engg., LLE
 *
 *      DATE:           Thu Sep 10 17:24:13 EDT 1987
 *      PURPOSE:        global definitions for xipmenu
 *
 *      DESCRIPTION:    this file contains the definitions for structures
 *                      used by xipmenu.
 *      FUNCTIONS:
 *      LIBRARIES:
 *      $Log$
 */

/*
 * A typical application program running under UNIX is executed from the
 * command line using the following syntax:
 *      % program [ -option < argument> ] [ -option2 < argument> ] ...
 * Some options are mandatory for the program to execute correctly.
 * If an option is specified it may or may not need an argument. The argument
 * if needed may be allowed to have a fixed range of values. If the value
 * specified is not within the range a default value is usually assumed
 * inside the application program.
 *
 * Thus a structure to build interactive menus for application program
 * should have at least the following information:
 *      + program name, & number of options,
 *      + options & the name associated with it, and if any arguments are needed
 *      + a prerequisite option if needed,
 *      + flag to check if the option is mandatory,
 *      + a default value, and a range of values for the argument
 * Given the above information the following typedef can be constructed
 * which provides a consistent medium for handling all application
 * programs. Since the XIP user-interface is primarily geared towards
 * image processing applications which are limited in number when compared
 * to all applications existing under UNIX, a list/tree of menus for all
 * image processing applications which themselves consists of lists of
 * options and associated arguments is feasible.
 *
 * Input to and output from menu structures is controlled by the use of
 * macros, which keep underlying details hidden from upper level routines
 * constructing menus to be displayed on the screen. Note that all fields
 * in the structures below are statically allocated, to make allocation/deallo-
 * cation of memory easier. Of course the pointers to the list of options
 * other option structures are needed for dynamic extension/contraction
 * of these lists.
 */

/*
 * PROG_OPT :    structure definition for an application program's options
 */
typedef struct progopt {
    char option[16];        /* option is at most 16 chars */
    int args;               /* flag specifies if option has arguments */
    char optname[32];       /* option name is at most 16 chars */
    char prereq[16];        /* prerequisite option */
    int check;              /* flag to check if option is mandatory */
    char def[32];           /* default value for option */
    char range[80];         /* range of values allowed in option */
    struct progopt *next;           /* pointer to next option */
} PROG_OPT;
```

```
/*
 * PROG_MENU     structure definition for an application program's menu
 */
typedef struct progmenu {
     char name[80];              /* program name's max size is 80 chars */
     short numopts;              /* number of options in program's menu */
     PROG_OPT *optlist;          /* pointer to first option structure */
     struct progmenu *nextmenu;          /* pointer to next menu in a class */
} PROG_MENU;

/*************************************************************************
 *
 * THIS EXTENSION TO XIPMENU MAY BE ADDED ON LATER WHEN IT IS POSSIBLE TO
 * CLASSIFY APPLICATION PROGRAMS BY TYPE OR ACTION. UNTIL THEN THE MENU
 * WILL BE BUILT FROM A LIST OF PROG_MENU'S AS DEFINED ABOVE. HOWEVER A
 * STRUCTURE HAS BEEN DEFINED TO ACCOMODATE THAT FEATURE.
 *
 ************************************************************************* /

/*
 * XIP_MENU ·     structure definitions for all application menu's under XIP
 */
typedef struct xipmenu {
     char class[32];                    /* class of application programs */
     PROG_MENU *classlist;              /* list of applcation menus under that list */
     struct xipmenu *nextclass;  /* pointer to next class of menus */
} XIP_MENU;
```

xipmenu_macros.h

```
/*
 *      FILE :          xipmenu_macros.h
 *      AUTHOR :        Sandeep Mehta, Optical Engg., LLE
 *
 *      DATE:           Thu Sep 10 18:59:43 EDT 1987
 *      PURPOSE:        macro definitions for xipmenu
 *
 *      DESCRIPTION:
 *      $Log$
 */

/*
 * "read from" macros for menu
 */
/*
 * class operations
 */
#define menu_getclass(A)                ((A)->class)
#define menu_getclasslist(A)            ((A)->classlist)
/*
 * menu operations
 */
#define menu_getprogname(A)             ((A)->name)
#define menu_getnumopts(A)              ((A)->numopts)
#define menu_getoptlist(A)              ((A)->optlist)
/*
 * option operations
 */
#define menu_getprogopt(A)              ((A)->option)
#define menu_getargflag(A)              ((A)->args)
#define menu_getoptname(A)              ((A)->optname)
#define menu_getprereq(A)               ((A)->prereq)
#define menu_getcheckflag(A)            ((A)->check)
#define menu_getdefault(A)              ((A)->def)
#define menu_getrange(A)                ((A)->range)

/*
 * "write to" macros for menu
 */
/*
 * class operations
 */
#define menu_putclass(A,B)              (strcpy((A)->class,(B)))
#define menu_putclasslist(A,B)          ((A)->classlist = (B))
/*
 * menu operations
 */
#define menu_putprogname(A,B)           (strcpy((A)->name,(B)))
#define menu_putnumopts(A,B)            ((A)->numopts = (B))
#define menu_putoptlist(A,B)            ((A)->optlist = (B))
/*
 * option operations
 */
#define menu_putprogopt(A,B)            (strcpy((A)->option,(B)))
#define menu_putargflag(A,B)            ((A)->args = (B))
#define menu_putoptname(A,B)            (strcpy((A)->optname,(B)))
#define menu_putprereq(A,B)             (strcpy((A)->prereq,(B)))
#define menu_putcheckflag(A,B)          ((A)->check = (B))
#define menu_putdefault(A,B)            (strcpy((A)->def,(B)))
#define menu_putrange(A,B)              (strcpy((A)->range,(B)))
```

Appendix C: Results

Session #1

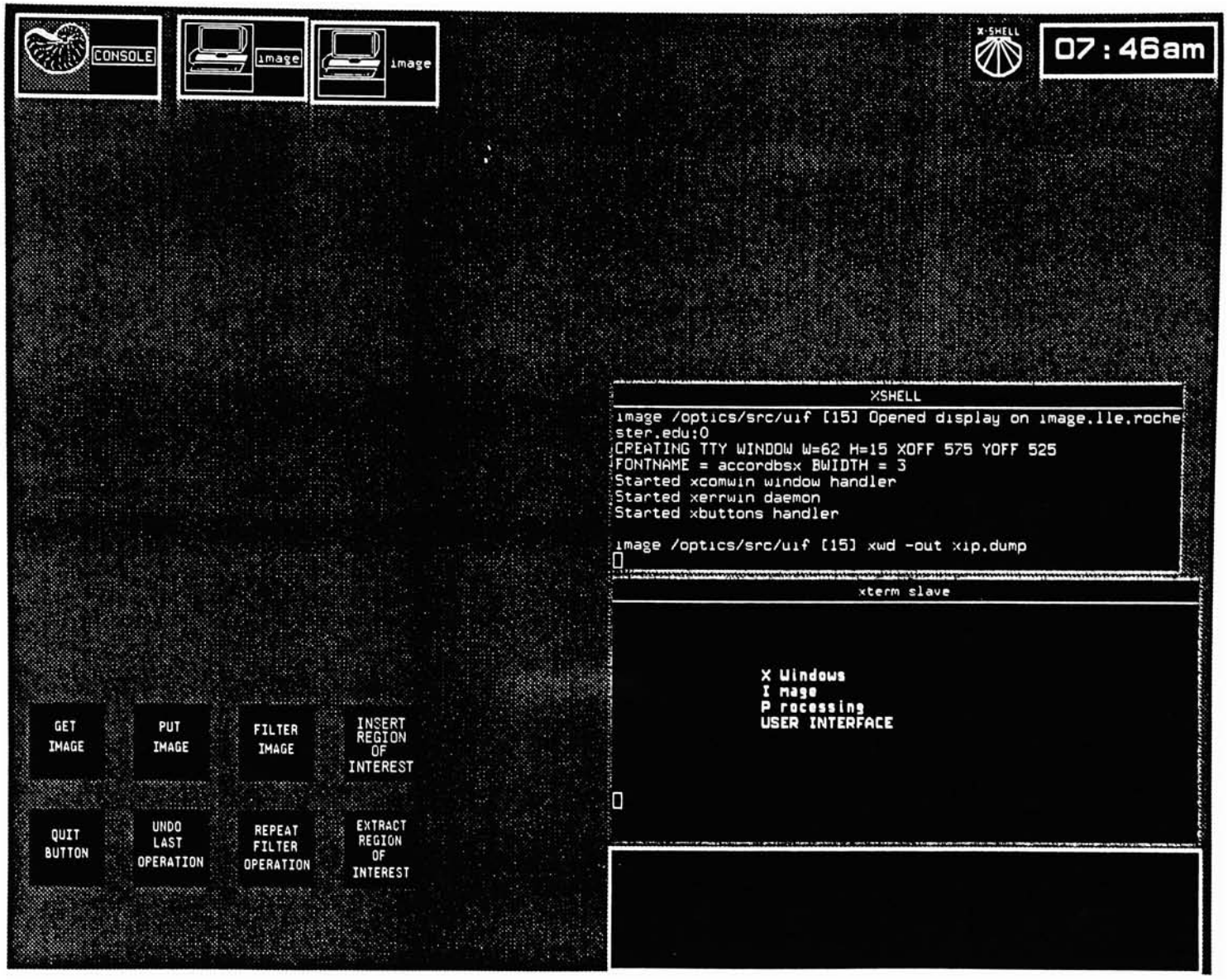
Figure S1.1 --------- Initial Screen Configuration.
Figure S1.2 --------- Image Window Displayed.
Figure S1.3 --------- Region of Interest Extracted.
Figure S1.4 --------- Region of Interest Processing.
Figure S1.5 --------- Region of Interest Inserted.

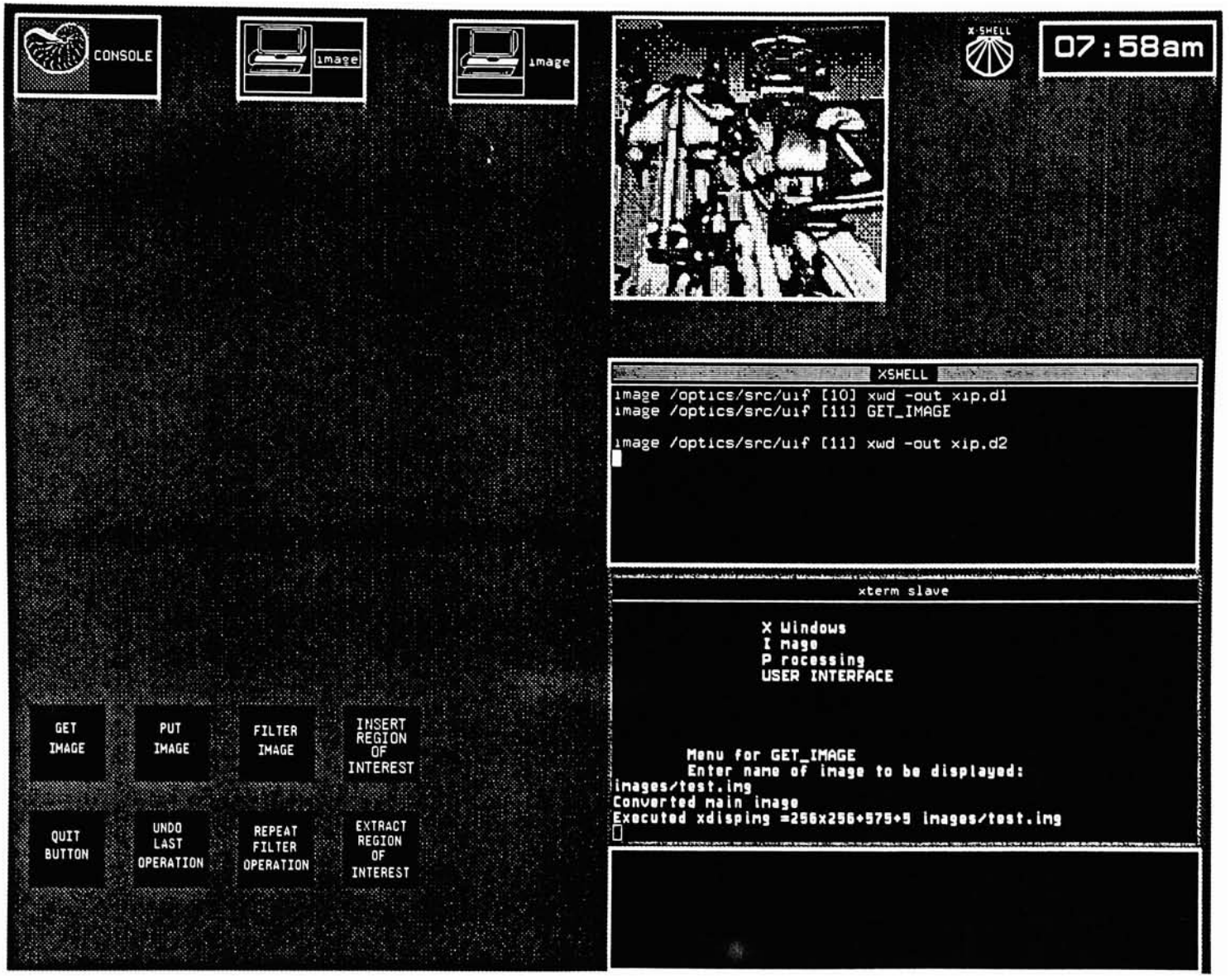
Session #2

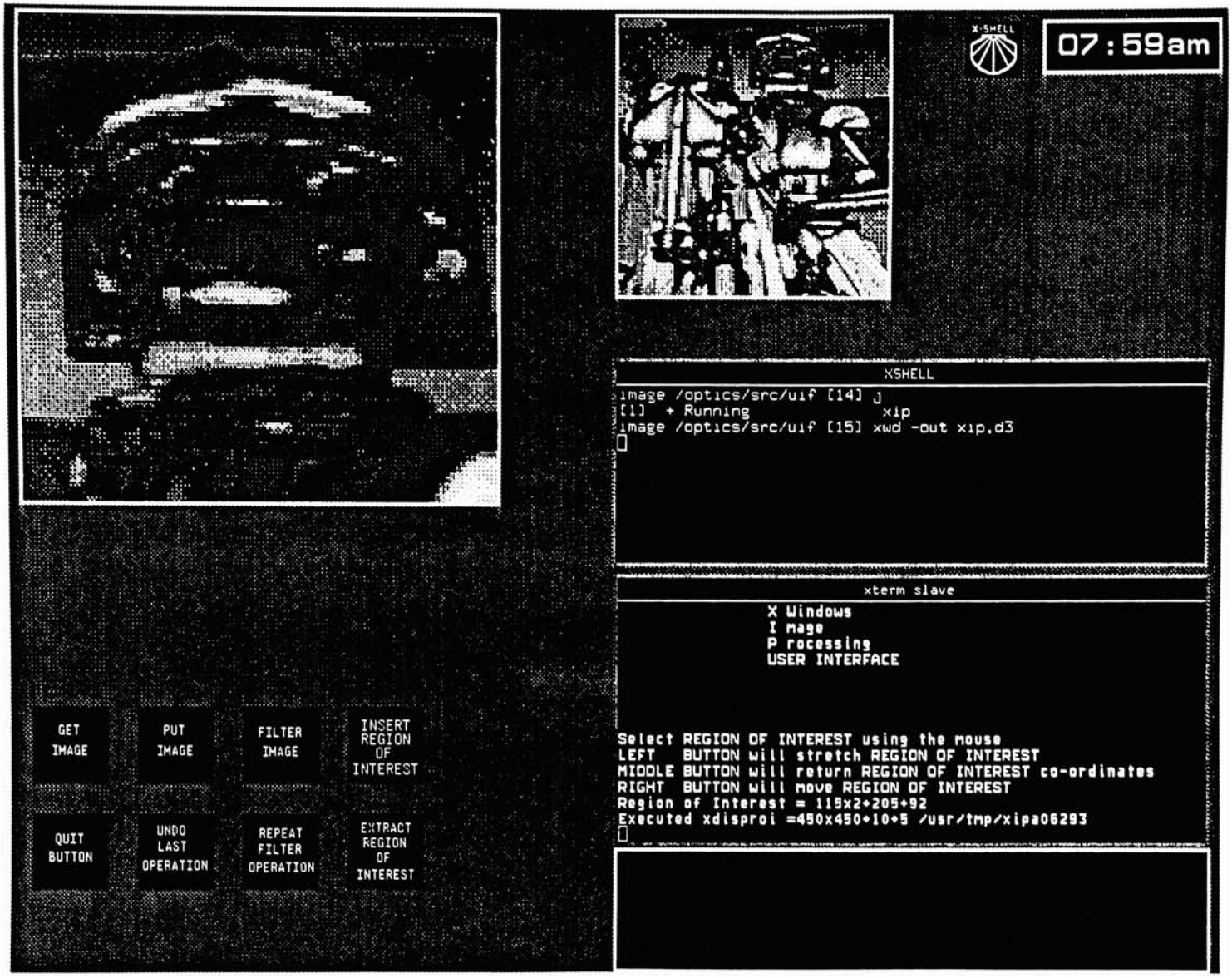
Figure S2.1 --------- Image Displayed
Figure S2.2 --------- Extracted Region of Interest
Figure S2.3 --------- Running Menu tool

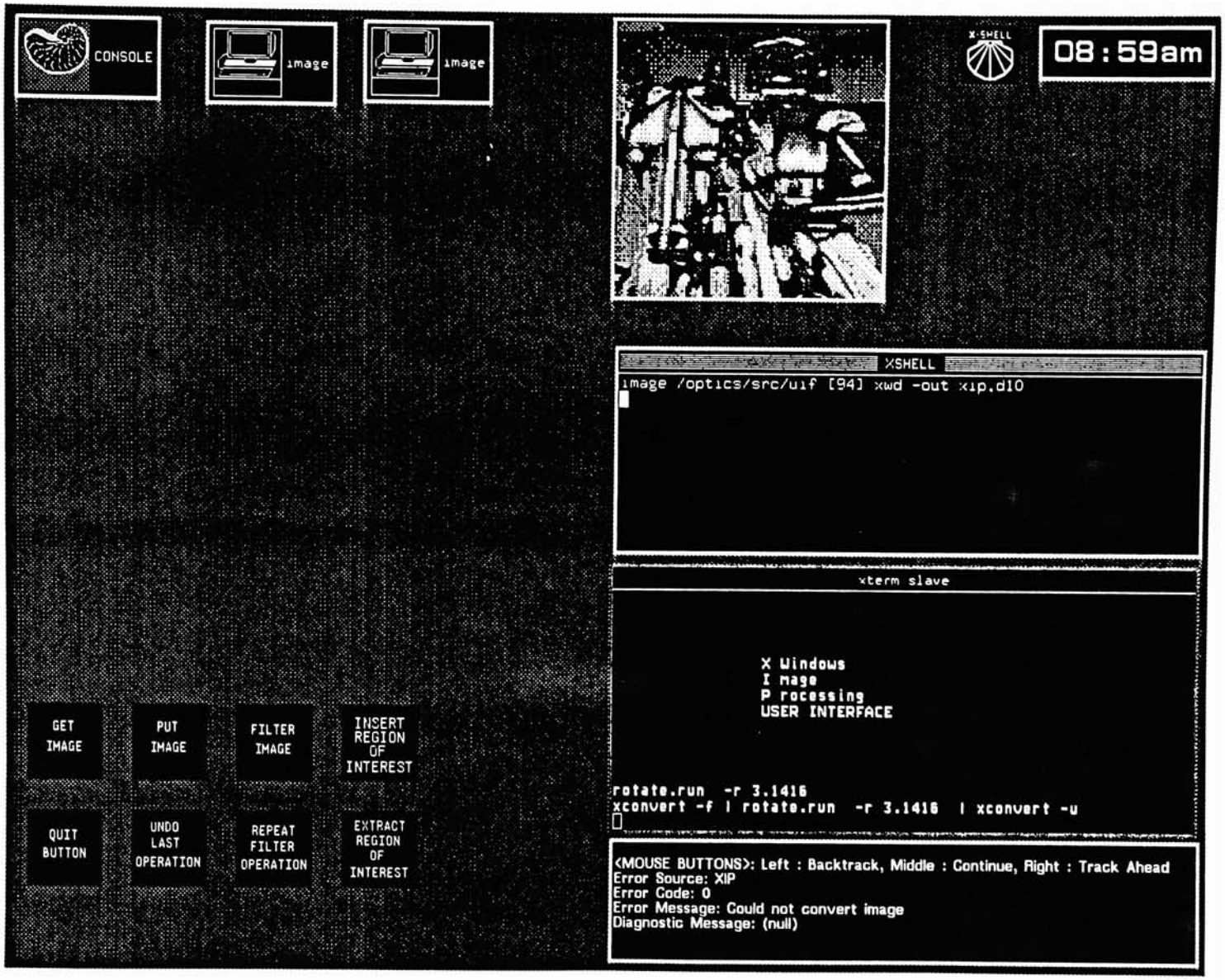
Miscellaneous Images

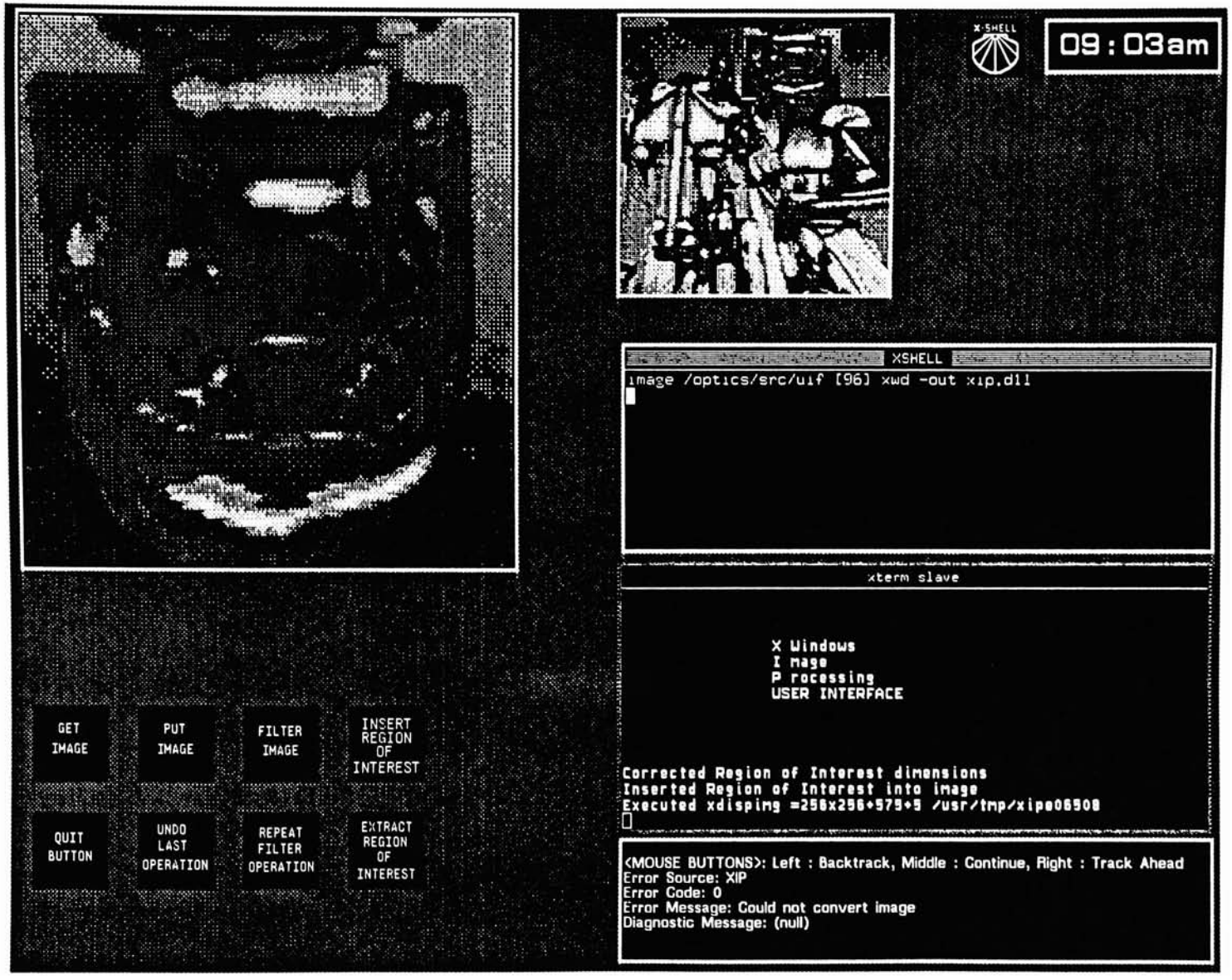
Monochrome Test Image: laserbay.img
Extracted Region of Interest
Monochrome Test Image: horse.rv
Gray Scale Test Image: horse.rv

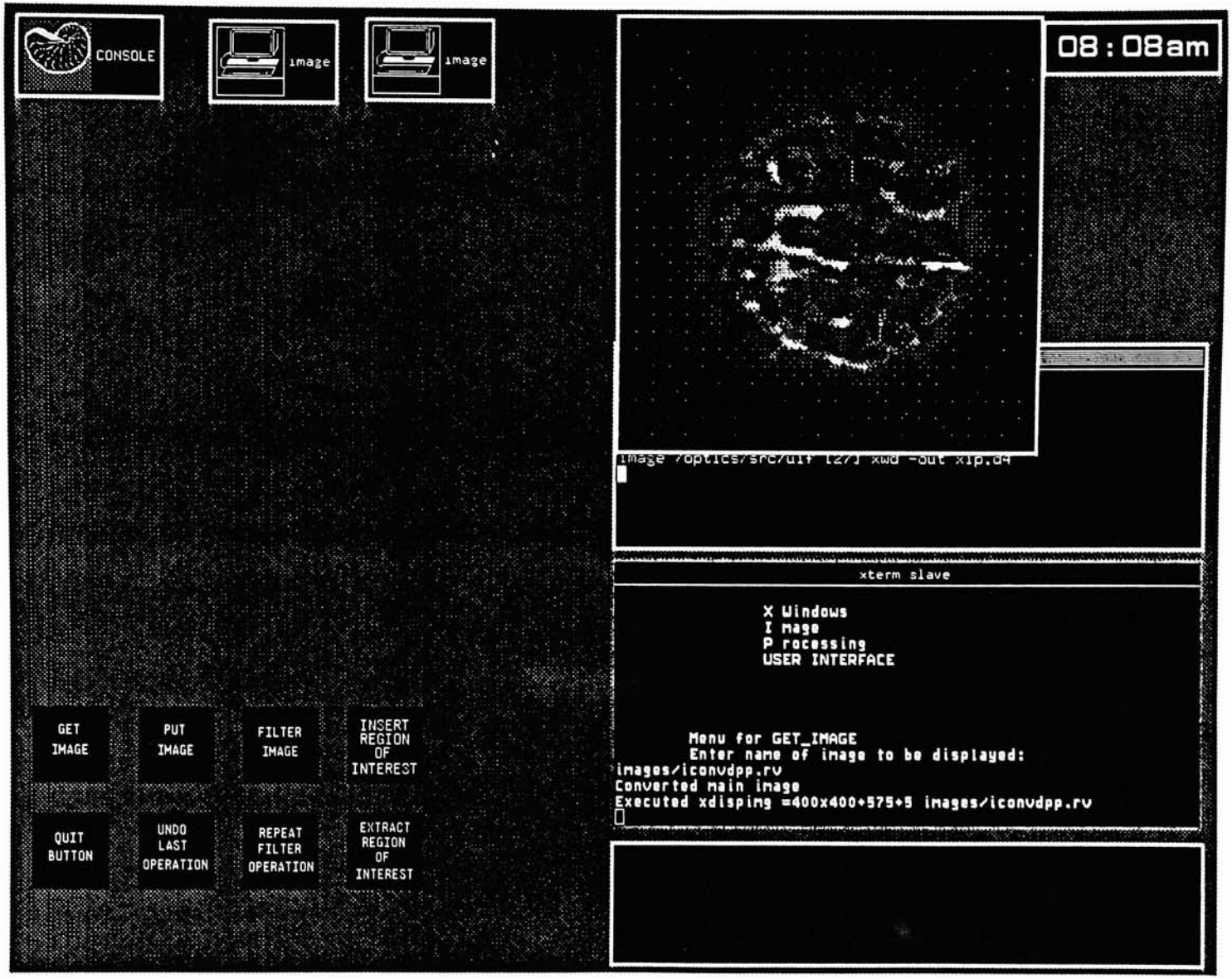

CONSOLE
image
image
X-SHELL
07:46am
XSHELL
image /optics/src/uif [15] Opened display on image.ile.rochester.edu:0
CREATING TTY WINDOW W=62 H=15 XOFF 575 YOFF 525
FONTNAME = accordbsx BWIDTH = 3
Started xcomwin window handler
Started xerrwin daemon
Started xbuttons handler
image /optics/src/uif [15] xwd -out xip.dump
xterm slave
X Windows
I mage
P rocessing
USER INTERFACE
GET IMAGE
PUT IMAGE
FILTER IMAGE
INSERT REGION OF INTEREST
QUIT BUTTON
UNDO LAST OPERATION
REPEAT FILTER OPERATION
EXTRACT REGION OF INTEREST

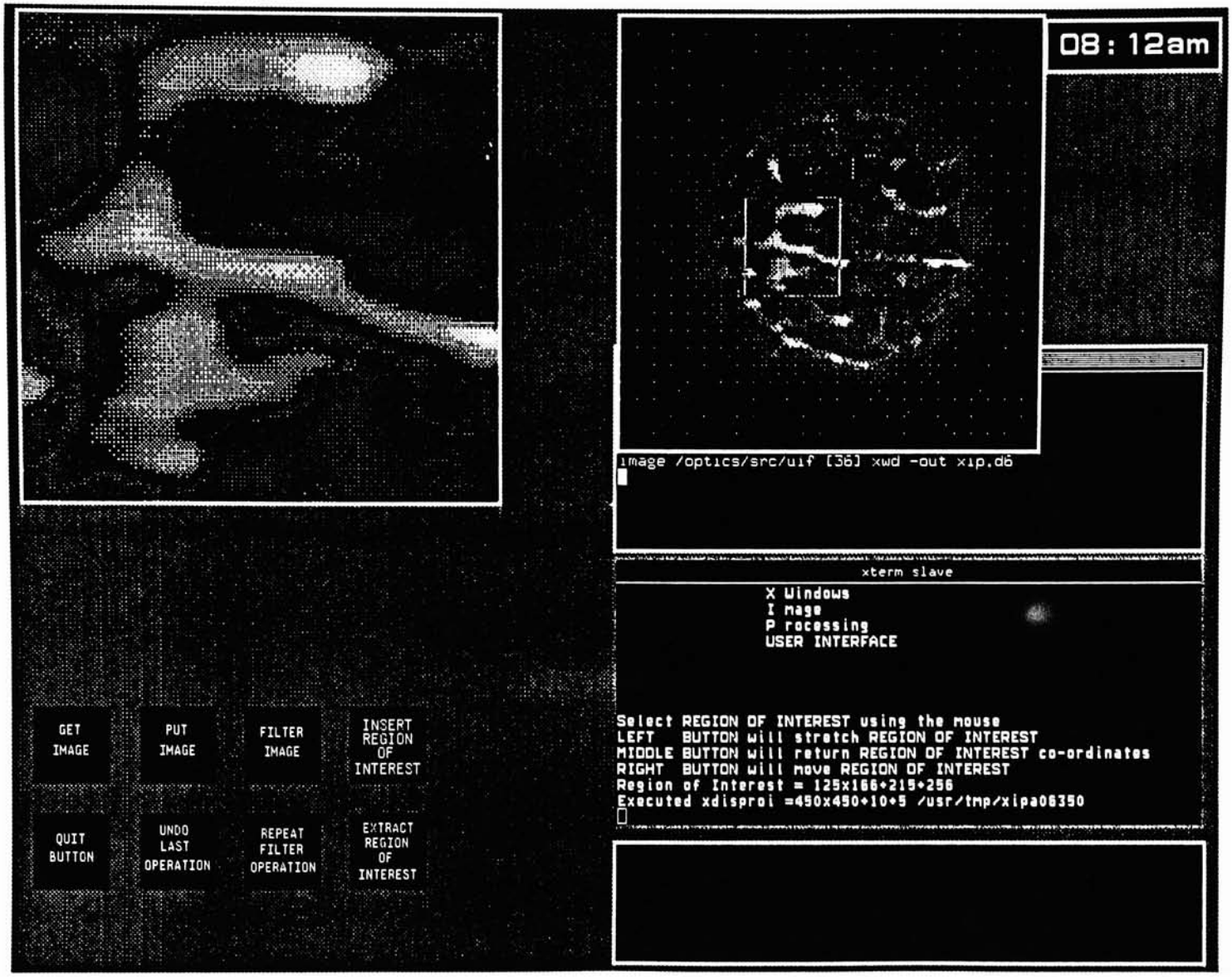

CONSOLE
image
image
X-SHELL
07:58am
XSHELL
image /optics/src/uif [10] xwd -out xip.d1
image /optics/src/uif [11] GET_IMAGE
image /optics/src/uif [11] xwd -out xip.d2
xterm slave
X Windows
I mage
P rocessing
USER INTERFACE
Menu for GET_IMAGE
Enter name of image to be displayed:
images/test.img
Converted main image
Executed xdisping =256x256+575+5 images/test.img
GET IMAGE
PUT IMAGE
FILTER IMAGE
INSERT REGION OF INTEREST
QUIT BUTTON
UNDO LAST OPERATION
REPEAT FILTER OPERATION
EXTRACT REGION OF INTEREST

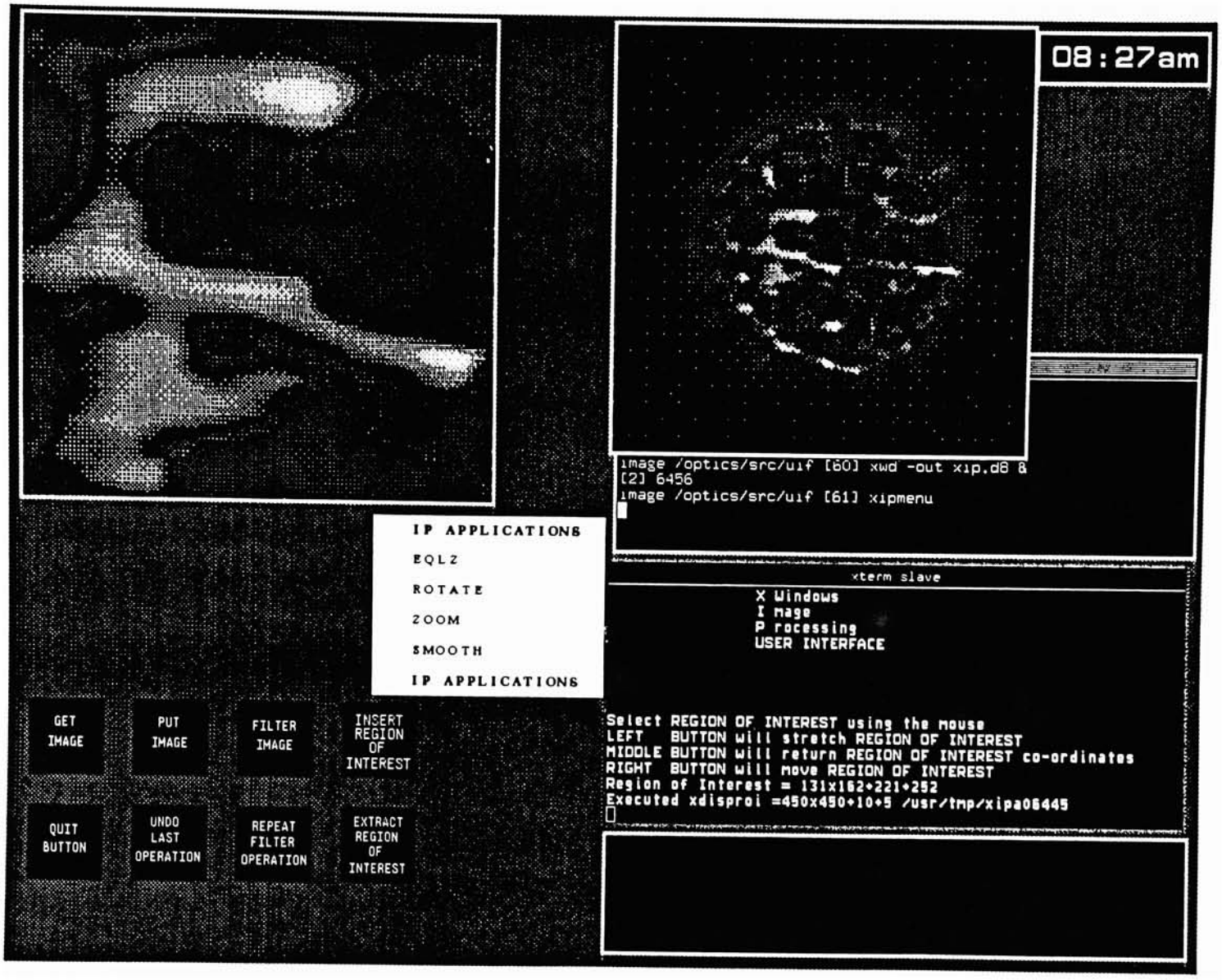

X-SHELL
07:59am
XSHELL
image /optics/src/uif [14] j
[1] + Running xip
image /optics/src/uif [15] xwd -out xip.d3
xterm slave
X Windows
I mage
P rocessing
USER INTERFACE
GET IMAGE
PUT IMAGE
FILTER IMAGE
INSERT REGION OF INTEREST
QUIT BUTTON
UNDO LAST OPERATION
REPEAT FILTER OPERATION
EXTRACT REGION OF INTEREST
Select REGION OF INTEREST using the mouse
LEFT BUTTON will stretch REGION OF INTEREST
MIDDLE BUTTON will return REGION OF INTEREST co-ordinates
RIGHT BUTTON will move REGION OF INTEREST
Region of Interest = 115x2+205+92
Executed xdisproi =450x450+10+5 /usr/tmp/xipa06293

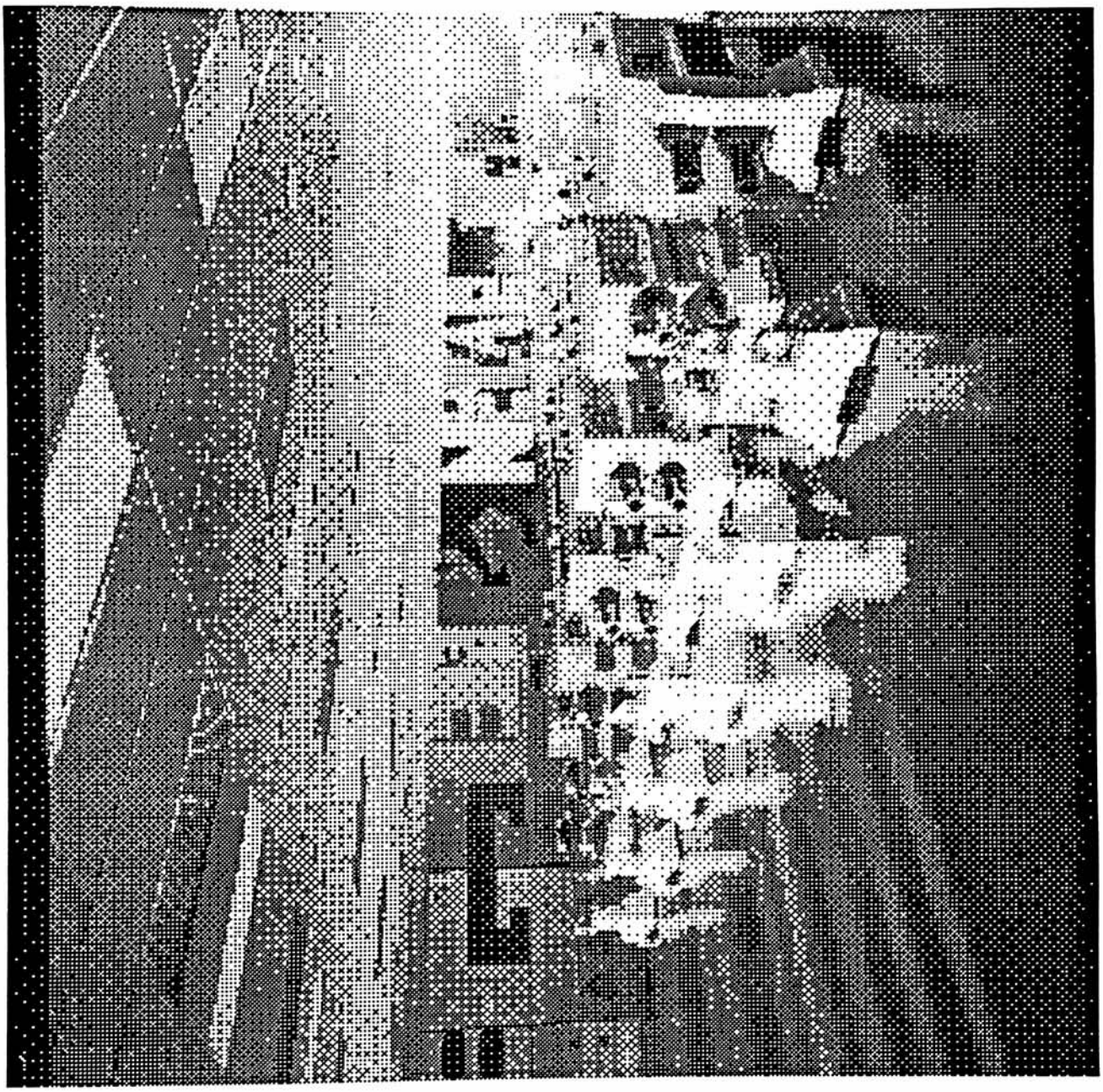

CONSOLE
image
image
X-SHELL
08:59am
XSHELL
image /optics/src/uif [94] xwd -out xip.dIO
xterm slave
X Windows
I mage
P rocessing
USER INTERFACE
rotate.run -r 3.1416
xconvert -f | rotate.run -r 3.1416 | xconvert -u
GET IMAGE
PUT IMAGE
FILTER IMAGE
INSERT REGION OF INTEREST
QUIT BUTTON
UNDO LAST OPERATION
REPEAT FILTER OPERATION
EXTRACT REGION OF INTEREST
<MOUSE BUTTONS>: Left : Backtrack, Middle : Continue, Right : Track Ahead
Error Source: XIP
Error Code: 0
Error Message: Could not convert image
Diagnostic Message: (null)

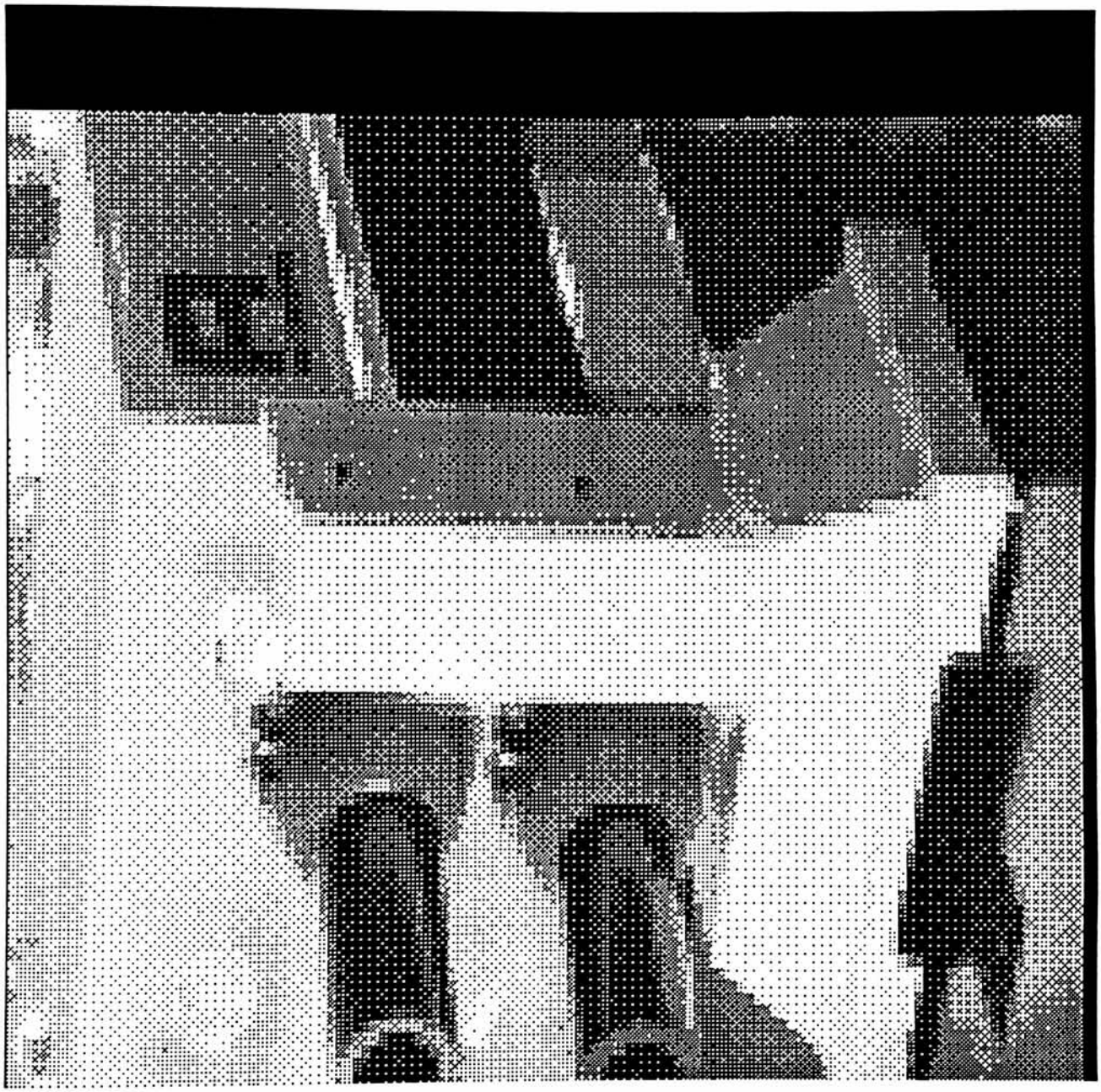

09:03am
XSHELL
image /optics/src/uif [96] xwd -out xip.dll
xterm slave
X Windows
Image
Processing
USER INTERFACE
Corrected Region of Interest dimensions
Inserted Region of Interest into image
Executed xdisping =256x256+575+5 /usr/tmp/xipa06508
GET IMAGE
PUT IMAGE
FILTER IMAGE
INSERT REGION OF INTEREST
QUIT BUTTON
UNDO LAST OPERATION
REPEAT FILTER OPERATION
EXTRACT REGION OF INTEREST
<MOUSE BUTTONS>: Left : Backtrack, Middle : Continue, Right : Track Ahead
Error Source: XIP
Error Code: 0
Error Message: Could not convert image
Diagnostic Message: (null)

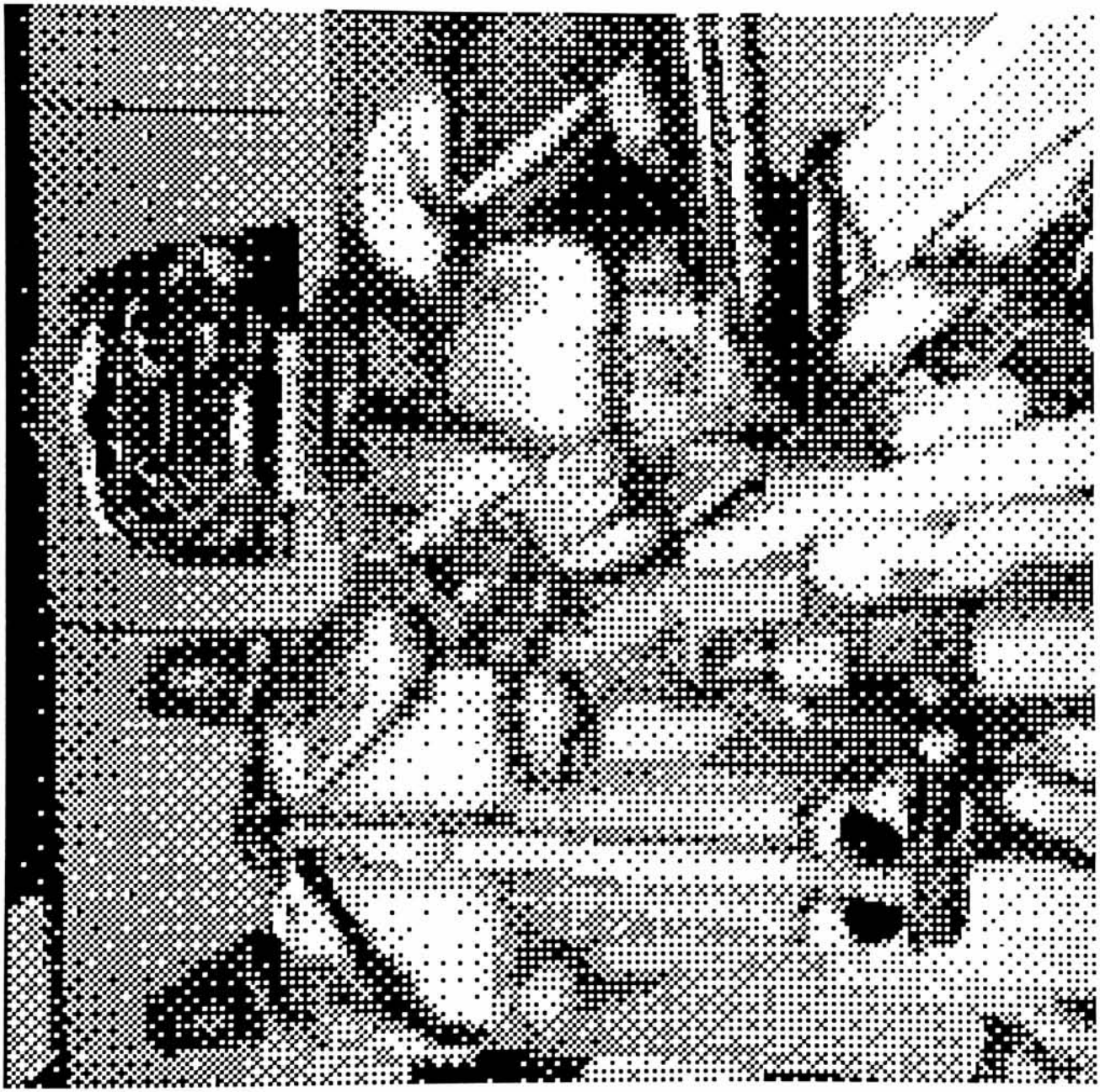

CONSOLE
image
image
08:08am
image /optics/src/ui+ [27] xwd -out xip.d4
xterm slave
X Windows
I mage
P rocessing
USER INTERFACE
GET IMAGE
PUT IMAGE
FILTER IMAGE
INSERT REGION OF INTEREST
QUIT BUTTON
UNDO LAST OPERATION
REPEAT FILTER OPERATION
EXTRACT REGION OF INTEREST
Menu for GET_IMAGE
Enter name of image to be displayed:
images/iconvdpp.rv
Converted main image
Executed xdisping =400x400+575+5 images/iconvdpp.rv

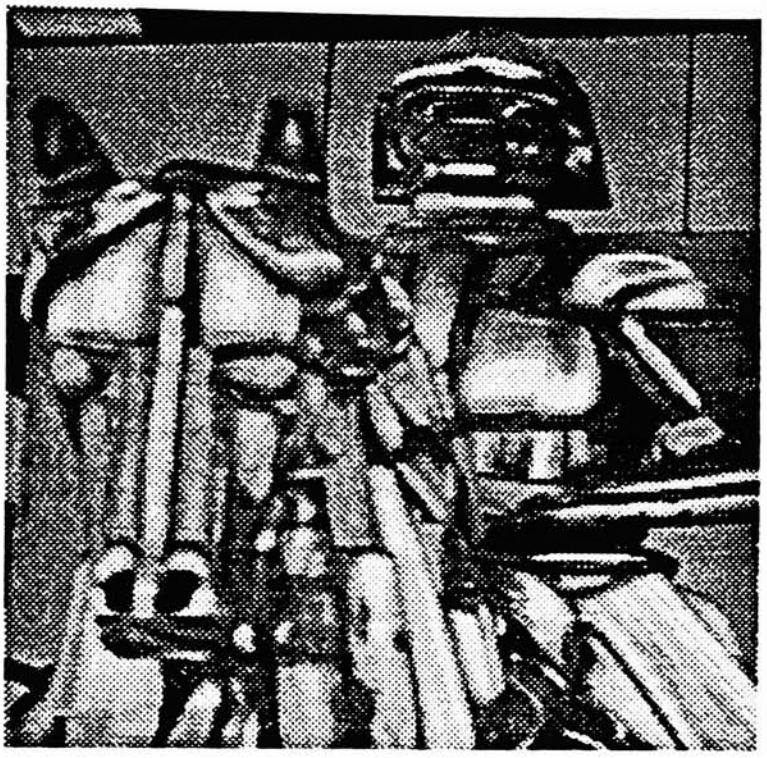

08:12am
image /optics/src/uif [36] xwd -out xip.db
xterm slave
X Windows
I mage
P rocessing
USER INTERFACE
Select REGION OF INTEREST using the mouse
LEFT BUTTON will stretch REGION OF INTEREST
MIDDLE BUTTON will return REGION OF INTEREST co-ordinates
RIGHT BUTTON will move REGION OF INTEREST
Region of Interest = 125x166+215+256
Executed xdisproi =450x450+10+5 /usr/tmp/xipa06350
GET IMAGE
PUT IMAGE
FILTER IMAGE
INSERT REGION OF INTEREST
QUIT BUTTON
UNDO LAST OPERATION
REPEAT FILTER OPERATION
EXTRACT REGION OF INTEREST


08:27am
image /optics/src/uif [60] xwd -out xip.d8 &
[2] 6456
image /optics/src/uif [61] xipmenu
xterm slave
X Windows
I mage
P rocessing
USER INTERFACE
IP APPLICATIONS
EQLZ
ROTATE
ZOOM
SMOOTH
IP APPLICATIONS
GET IMAGE
PUT IMAGE
FILTER IMAGE
INSERT REGION OF INTEREST
QUIT BUTTON
UNDO LAST OPERATION
REPEAT FILTER OPERATION
EXTRACT REGION OF INTEREST
Select REGION OF INTEREST using the mouse
LEFT BUTTON will stretch REGION OF INTEREST
MIDDLE BUTTON will return REGION OF INTEREST co-ordinates
RIGHT BUTTON will move REGION OF INTEREST
Region of Interest = 131x162+221+252
Executed xdisproi =450x450+10+5 /usr/tmp/xipa06445









Appendix D: Selected Statistics

Test Runs of Display, Menu, and Region of Interest Processing Programs.

STATISTICS FOR DISPLAY IMAGE OPERATION	
Compiled Flags : -gx -DDEBUG Command : test.img, 256x256, 16 bits Image : xdispimg =256x256+10+10	
real time	13.5 secs
user time	8.2 secs
sys time	1.3 secs

STATISTICS FOR DISPLAY IMAGE OPERATION	
Compiled Flags : -gx -DDEBUG Command : laserbay.img,512x512,16bits Image : xdispimg =512x512+10+10	
real time	43.6 secs
user time	31.0 secs
sys time	3.0 secs

STATISTICS FOR XIPMENU PROCESS	
Compiled Flags : -gx -DDEBUG Command : rotate.run -d180 -x180 -y180 Menu Mode : Interactive selection	
real time	22.2 secs
user time	1.1 secs
sys time	1.3 secs

STATISTICS FOR XIPMENU PROCESS	
Compiled Flags : -gx -DDEBUG Command : rotate.run -d180 -x180 -y180 Menu Mode : Interactive selection	
real time	24.9 secs
user time	1.1 secs
sys time	1.1 secs

STATISTICS FOR XIPMENU PROCESS	
Compiled Flags : `-O -s` Command : `rotate.run -d360 -x180 -x180` Menu Mode : `Interactive selection`	
real time	`24.3 secs`
user time	`1.2 secs`
sys time	`1.0 secs`

STATISTICS FOR XIPMENU PROCESS	
Compiled Flags : `-O -s` Command : `rotate.run -d180 -x256 -x256` Menu Mode : `Default Selection`	
real time	`10.6 secs`
user time	`1.0 secs`
sys time	`1.1 secs`

STATISTICS FOR REGION OF INTEREST PROCESSING	
Compiled Flags: `-gx -DDEBUG` Command: `smooth.run -r 3` Type: `using temp files` Image: `laserbay.img 512x512, 16 bits`	
real time	`46.2 secs`
user time	`23.0 secs`
sys time	`6.1 secs`

STATISTICS FOR REGION OF INTEREST PROCESSING	
Compiled Flags: `-gx -DDEBUG` Command: `smooth.run -r 3` Type: `pipeline loop` Image: `laserbay.img 512x512, 16 bits`	
real time	`42.8, 42.5, 42.7 secs`
user time	`23.0, 23.3, 23.4 secs`
sys time	`7.0, 6.9, 7.4 secs`

STATISTICS FOR REGION OF INTEREST PROCESSING	
Compiled Flags: `-gx -DDEBUG` Command: `smooth.run -r 3` Type: `pipeline loop` Image: `test.img 256x256, 16 bits`	
real time	`12.9, 14.8, 13.2 secs`
user time	`6.5, 6,6, 6,5 secs`
sys time	`2.3, 3.1, 2.8 secs`

STATISTICS FOR REGION OF INTEREST PROCESSING	
Compiled Flags: `-gx -DDEBUG` Command: `smooth.run -r 3` Type: `using temp files` Image: `test.img 256x256, 16 bits`	
real time	`13.5 secs`
user time	`6.4 secs`
sys time	`2.6 secs`

References

ADD84.

J. D. Addington, "A Device & User-Interface Independent Image Processing System," *Proceedings of the SPIE, Int. Soc. Opt. Engg. (USA)*, vol. 504, pp. 112-15, SPIE, 1984.

BAS85.

L. J. Bass, "Generalized User Interface for Application Programs," *Communications of the ACM*, vol. 28, No. 6, pp. 617-27, ACM, 1985.

BRU82.

J. D. Bruner and A. P. Reeves, "An Image Processing System with Computer Network Distribution Capabilities," *Proceedings of IEEE Conference on Pattern Recognition & Image Processing*, pp. 447-50, IEEE, New York, June 1982.

CHA83.

S. K. ChangI. Jurkevich, A. Petty, and C. C. Yang, "Software design for Image Information Systems," *EASCON, '83: 16th Annual IEEE Electronics & Aerospace Systems Conference & Exposition.* , pp. 115-20, IEEE, New York, 1983.

CHA83.

Shi-Kuo Chang, E. Jungert, S. Levialdi, G. Tortora, and T. Ichikawa, "An Image processing langauge with icon-assisted navigation," *IEEE Transactions on Software Engineering (USA)*, vol. SE-11, No. 8, pp. 811-19, IEEE, Aug. 1985.

COT84.

Y. R. de Cotret and A. Schubert, "A user interface for photographic image processing based on a photgraphic kernel system," *ESPRIT '84. Status Report of ongoing work*, pp. 355-64, North-Holland, Amsterdam, Netherlands, Sept. 1984.

DAW87.

Benjamin M. Dawson, "Introduction to Image Processing Algorithms," *BYTE Magazine*, pp. 169-186, March 1987.

DEN87.

Owen M. Densmore and David S. H. Rosenthal, "A User-Interface Toolkit in Object Oriented POSTSCRIPT," *Computer Graphics Forum*, vol. 6, pp. 171-180, North Holland, 1987.

DRA84.

S. W. Draper and D. A. Norman, "Software Engineering for User Interfaces," *IEEE Proceedings on Software Engineering*, pp. 214-20, IEEE, 1984.

FRE77.

Werner Frei, "Image Enhancement by Histogram Hyperbolization," *Computer Graphics & Image Processing*, vol. 6, pp. 286-94, Academic Press, Inc. , 1977.

FUK84.

T. Fukushima, S. Miura, Y. Kobayashi, K. Hirasawa, M. Takatoo, and T. Usui, "Interactive Picture Evaluation software package," *1984 IEEE Computer Society Workshop on Visual Langauges*, pp. 143-8, IEEE Computer Society Press, Silver Spring, MD, USA, 1984.

GET86.

Jim Gettys, Ron Newman, and Tony Della Fera, *Xlib C Langauge X Interface Protocol Version 10*, MIT Project Athena, Nov. 1986.

GRE85.

W. B. Green, "Image Processing System Interfaces," *Proceedings of the SPIE Int. Soc. Opt. Eng. (USA)*, vol. 528, pp. 103-9, SPIE, Los Angeles, Jan. 1985.

HAR85.

H. Rex. Hartson, *Advances in Human-Computer Interaction*, 1, Ablex Publishing Corp. , Norwood, New Jersey. , 1985.

HAV86.

W. Havens, "PIPS: A portable image processing system," *IEEE Computer Society Workshop on*

Visual Langauges. , pp. 125-35, IEEE Computer Society Press, Washington DC, 1986.

HAY85.

Philip J. Hayes, "Executable Interface Definitions Using Form-Based Interface Abstractions. ," *Advances in Human-Computer Interaction*, vol. 1, pp. 161-189, Ablex Publishing Corp. , Norwood, New Jersey. , 1985.

HEC82.

Paul Heckbert, "Color Image Quantization for Frame Buffer Display," *Computer Graphics*, vol. 16, No. 3, pp. 297-307, ACM, July 1982.

JAR74.

J. F. Jarvis, C. N. Judice, and W. H. Ninke, "Using Ordered Dither to Display Continuous Tone Pictures on an AC Plasma Panel," *Proceedings of the S. I. D.* , vol. 15/4, pp. 161-69, SID, 1974.

JAR76.

J. F. Jarvis, C. N. Judice, and W. H. Ninke, "A Survey of Techniques for the Display of Continuous Tone Pictures on Bilevel Displays," *Computer Graphics and Image Processing*, vol. 5, pp. 13-40, Academic Press, 1976.

JOY85.

K. I. Joy, "Graphics interface tool development in a problem solving environment," *Visual Comput. (Germany)*, vol. 2, No. 2, pp. 63-71, Computer Graphics, Tokyo, 1985.

JOY85..

K. I. Joy, "A model for graphics interface tool development," *Proceedings of Graphics Interface, 1985*, pp. 159-65, Canadian Inf. Process. Soc. „ Toronto, Ont. , Canada, May 1985. .

KAS82.

David J. Kasik, "A User-Interface Management System," *Computer Graphics*, vol. 16, No. 3, pp. 99-105, ACM, July 1982.

KER78.

Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice Hall Software Series, Englewood Cliffs, NJ., 1978.

LEF. Samuel J. Leffler, Robert S. Fabry, and William N. Joy, *A 4. 2BSD Interprocess Communication Primer*, Computer Systems Research Group, University of California, Berkeley.

LIN83.B. Lindskog, B. Linnander, F. Bergquist, and P. Syven, "The B3 Image Processing System," *Proceedings of the 3rd Annual Scandinavian Conference on Image Analysis*, pp. 418-21, Studentlitteratur, Lund, Sweden, July 1983.

LIN84.S. Linnainmaa, "Interactive Picture Processing using ICEPIC user interface," *7th International Convference on Pattern Recognition*, vol. 2, pp. 684-86, IEEE Computer Society Press, Silver Spring, MD, USA, 1984.

MOR86.

James H. Morris, Mahadev Satyanarayanan , and Michael H. Conner, et. al. , "Andrew: A Distributed Personal Computing Environment," *Communications of the ACM*, vol. 29, No. 3, pp. 184-201, ACM, March 1986.

PAL83.

J. Palme, "A Human-Computer interface encouraging user growth," *Designing for Human Computer Communication*, pp. 139-156, Academic Press, London, 1983. edited by M. E. Sime, M. J. Coombs

PAV82.

Theo Pavlidis, *Algorithms for Graphics and Image Processing*, p. 100,122, Computer Science Press, 1982.

RAM85.

E. Ramo, "Digital Image Processing Workstation - a system concept," *Proceedings of Image Science '85.* , Helsinki, Finland, June 1985.

RIT84.P. R. Ritter, A. Kraugers, and A. J. Travlos, "Design on computer software for geographic image processing," *The 9th W. T. Pecora Remote Sensing Symposium : Spatial Info. T Tech. for Today & Tomorrow.* , pp. 33-9, IEEE Computer Soc. Press. , Silver Spring, MD, USA, Oct. 1984.

SHE83.

B. Sheil, "Power Tools for Programmers," *Datamation*, vol. 29, pp. 131-144, Feb. ,1983.

SWA86.

Stephen Swales, "Menu - A Language Compiler for Generating Interactive Menus," *LLE Internal Technical Report*, 1986.

TAN81.

Andrew Tanenbaum, *Computer Networks*, Prentice Hall Software Series, 1981.

WAS85.

Anthony I. Wasserman and David T. Shewmake, "The Role of Prototypes in the User Software Engineering (USE) Methodology," *Advances in Human-Computer Interaction*, vol. 1, pp. 191-199, Ablex Publishing Corp. , Norwood, New Jersey. , 1985.

WES85.

T. D. Westrup, W. Kegel, and J. Gras, "User interaction withan environment for Image Procesing and Graphics," *Computer Graphics Forum (Netherlands)*, vol. 4, No. 3, pp. 187-202, Eurographics Conference (UK). , Bath, England, Sept. 1985.

PRI77.D. H. Pritchard, "U.S. Color Television Fundamentals A Review," *IEEE Transactions on Consumer Electronics*, vol. CE-23 (4), pp. 467-478, November 1977.

SMI78.

A. R. Smith, "Color Gamut Transform Pairs," *SIGGRAPH 1978 Proceedings*, vol. 12(3), pp. 12-19, August 1978.