Part Number 800-1122-01
Revision: A of 7 January 1984
For: Sun System Release 1.1

SunWindows Programmer's Guide

An Introduction to

the Sun Window System

Sun Microsystems, Inc.
2550 Garcia Avenue
Mountain View
California 94043
(415) 960-1300

Abstract

The *SunWindows Programmer's Guide* is a tutorial supplement to the *Programmer's Reference Manual for SunWindows*.

Revision History

Rev	Date	Comments
A	7 January 1984	First release of this programmer's guide.

Table of Contents

Table of Contents

List of Figures

Preface

Welcome to SunWindows, the Sun window system. This *SunWindows Programmer's Guide* provides a tutorial introduction to the overall SunWindows structure, to writing a tool, and to handling graphics applications in a window. It also points you to additional information in the *Programmer's Reference Manual for SunWindows*.

Who Should Read this Guide

This guide is intended for the programmer who wants tutorial instructions on how to write an application program using the SunWindows facilities. We assume a working understanding of UNIX, I/O concepts, and the C programming language and some knowledge of signals. We also assume that you are using a Sun workstation with a mouse so you can experiment with the examples.

How to Use this Guide

We recommend that you sit down at your workstation with this guide and try the examples as you read about them. Be sure to read the explanations in the order presented so you understand the terminology and basic concepts that we build on in subsequent chapters.

At times we provide rather sketchy descriptions to give you the general idea, but not the details. When you are ready to study the details of the window system, refer to the *Programmer's Reference Manual for SunWindows*. You may want to glance through it and return to the appropriate chapters as the need arises. Use the index to find detailed accounts of item glossed over in this guide. There are examples of window system application code in the appendices of the reference manual that you may also want to use.

For an elementary introduction to the general use of the mouse, the SunWindows pop-up menus, and the facilities provided with SunWindows, see the *Beginner's Guide to the Sun Workstation* and the *suntools*(1) entry in the *User's Manual for the Sun Workstation*.

Guide Contents

This guide contains the following:

Chapter 1 — *An Introduction to Windows* — describes what a window system is and what it is good for from the user's perspective; provides definitions of concepts and terms; and presents the SunWindows features. Novices should start here.

Chapter 2 — *SunWindows Implementation Overview* — describes the layers and basic concepts of SunWindows from the programmer's perspective.

Chapter 3 — *Applications — Tools and Canvas Programs* — differentiates between types of SunWindows programs; discusses high-livel implementation approaches; and describes existing SunWindows programs.

Chapter 4 — *Writing a Simple Tool* — provides an in-depth explanation of how to write a simple tool.

Chapter 5 — *Writing a Simple Canvas Program* — details how to write a simple graphics application for SunWindows.

Chapter 6 — *Writing a More Sophisticated Tool* — provides a detailed explanation of a more complex tool.

Chapter 7 — *Writing a More Sophisticated Canvas Program* — details how to write a more complex canvas program that receives input.

Chapter 8 — *Additional Topics* — explains how to implement a subwindow package and describes the *suntools* program that initializes and terminates a window environment.

Appendix A — *Glossary* — provides definitions of terms used in this manual.

Appendix B — *Bibliography* — an annotated list of references for additional reading on window systems and graphics applications.

Index — a quick reference to terms and code.

Note: The code examples show the proper case of letters for the names of macros, procedures, arguments, flags, and so on. In the text surrounding the code examples, the first letter in a sentence is capitalized as a courtesy to English, although the word may not then be technically correct.

Chapter 1

An Introduction to Windows

This chapter introduces you to window systems in general and to SunWindows in particular. It looks at what a window system is and describes the benefits of using windows from a user's point of view. If you are already familiar with a window system, skip the first section and read the second on what distinguishes the SunWindows system.

1.1. The What and Why of a Window System

A user of a window system views his environment through a collection of several rectangular display *windows*, each of which can correspond to a different task or context. A window system minimizes the context that the user must remember; it remembers the state of the user's partially completed tasks while the user is working on a different task. He manipulates the windows and their contents by a combination of keyboard inputs and pointing operations. The technique of using different windows for different tasks makes it easy to manage several simultaneous tasks and contexts, such as defining programs, testing programs, editing, asking the system for assistance, sending and receiving electronic mail, and switching back and forth between these tasks at his convenience.

A window system divides the area available on the display screen into multiple regions, usually rectangles. Each region provides a window through which the user and the computer interact. At any time the user can change the size of a window, allotting more of the display space to those windows currently containing interesting information, while ignoring the other uninteresting windows. Some window systems support *tiling*; a window fits into the available screen real estate so all windows are visible at once. Other systems support *overlapping windows*; windows can overlap one another, in the same way that pieces of paper can overlap one another. In this case, windows that are underneath can be brought to the top of the stack and vice versa. Overlapping windows increase the user's effective working space, and contribute to the notion that the user is working on an electronic desk top. The user interacts with the *window manager* of a window system to move, change the size of, and modify the overlap of a window.

The figure below shows a sample window display.

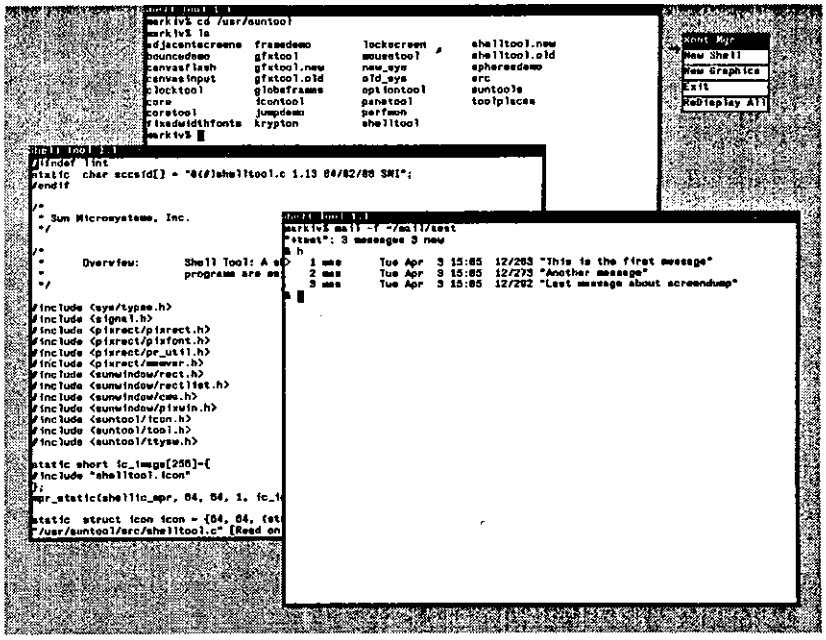


Figure 1-1: Sample Window Display

The figure shows that the user is reading his mail is one window, editing a file in another, and listing the files in a directory in a third. Output in a particular application's window can be tailored so it clearly coincides with the application and controls the user input. The user can collect all the information specific to an application in a single window, so each window corresponds to a different task.

With multiple windows the user can work on multiple tasks concurrently as the figure shows. Each task has a visual representation on the display. To stop working on one task and start working on another task, all the user need do is move his attention from one window to another by pointing at that window with the pointing device. All of the context of the other tasks is preserved in their windows. Moreover, while interacting with the user on the second task via the second window, the computer can be processing the first task in the background. Thus, the user can interact simultaneously with multiple processes.

Most window systems provide some form of interaction via *menu* input. A menu is a list of commands and/or parameters displayed to the user in a window. When the user chooses an item in the menu, the system starts an operation or changes its internal state. For instance, the figure also shows a simple menu with four items. A menu can be permanent, in the sense that it is always displayed, or it can be temporary, only appearing in response to a specific user input and disappearing when the user is finished with it. Menus present lists of options from which the user can choose. In addition, menu items can usually be chosen without having to use the standard typing keys, thereby permitting these keys to keep a standard interpretation.

Window systems use a ***bit-mapped display*** and a pointing device, called a ***mouse***, to simplify and speed up user interaction. A bit-mapped display is a display device which has independent control for each displayed point. A displayed point is called a *pixel* for *pic*ture *el*ement. A one-to-one mapping of each pixel to a portion of memory provides the control values. Note that for color displays, more than one bit is mapped to a pixel. The bit-mapped display allows windows to contain arbitrary images in addition to simple text. In fact, text is just graphics with familiar shapes. The bit-mapped display's graphical capability significantly improves the ability of a

program to present information to the user in ways that the user can quickly grasp.

The user points with the mouse to direct the window system's attention to particular regions of the display. A special image known as the *cursor* tracks this direction and the movement of the mouse on the display. In the figure, the cursor is the small arrow near the upper righthand corner. Typical regions that the user can point to with the mouse are individual windows, items inside a menu, characters or collections of characters inside windows displaying text, lines or other geometric primitives inside windows displaying graphics, and even individual pixels. The value of the pointing device is that it allows very quick movement, usually much faster than that available from keys on the keyboard, while at the same time allowing very precise positioning, usually with single pixel resolution.

1.2. What is SunWindows?

SunWindows is the Sun window system. It supports rectangular windows on both monochrome and color displays. SunWindows accepts input from the keyboard and from the Sun workstation's pointing device, which is usually a mouse.

The mouse is used to point at text and graphics displayed on the screen. The user directs his input to a window by moving the cursor into that window. The user can also call up menus with the mouse by pressing one of its buttons and choosing the menu items. The user can point at most text that is displayed on the screen and treat it as input, exactly as though it were typed. Text can be chosen with the mouse and copied within and between windows.

The user tells SunWindows to change a window to an *icon* when he is not concerned with the contents of a window. An icon is a small picture that represents the application area of the window. For example, the icon for the window that continuously displays the date and time as text is a clock face with hour and minute hands as shown below next to its window.

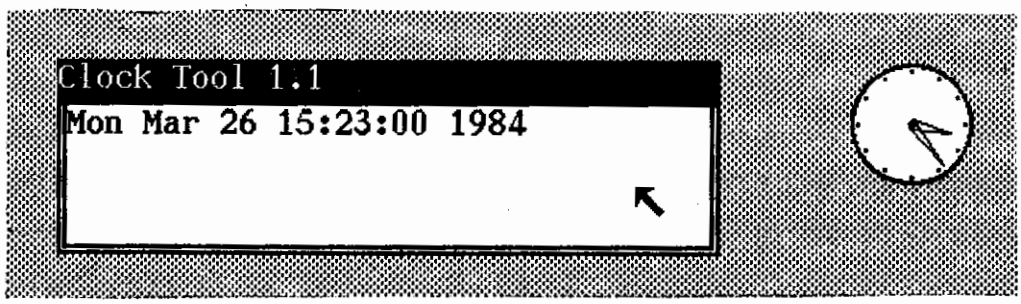


Figure 1-2: Clock Tool and Icon

SunWindows encourages the construction of application-oriented windows. We use the term *tool* to describe such a program focused on a particular task and written for execution in the SunWindows environment. A tool is not a window; it is a program which creates and destroys windows on the display during its execution. Most SunWindows tools have at least two windows. Looking at the following figure, you see that one window acts as the frame, defining the maximum area of the display that the tool controls. It usually displays the name of the tool in a stripe at the top and displays a border all around its region.

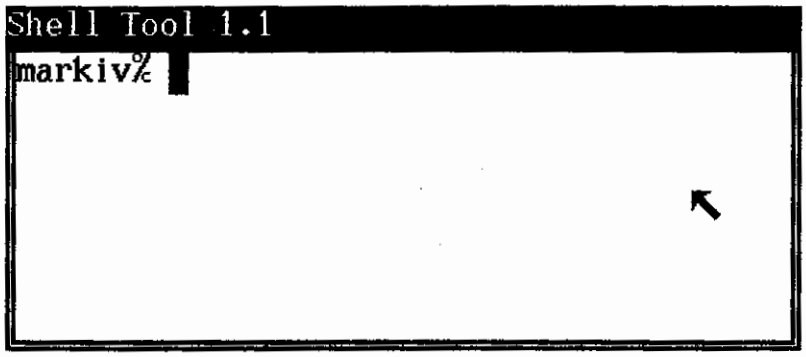


Figure 1-3: SunWindows Tool Window

Here, this name stripe shows "Shell Tool 1.1." The other window(s) occupy areas inside this frame, obscuring most of it. The inner windows are called *subwindows*, because they are subordinate to the frame. In particular, if the frame is moved, all of the subwindows move with it, keeping their positions relative to the frame. Subwindows are tiled over the surface of a tool.

Now consider the following figure. We see in this sample display that the user has five tool windows.

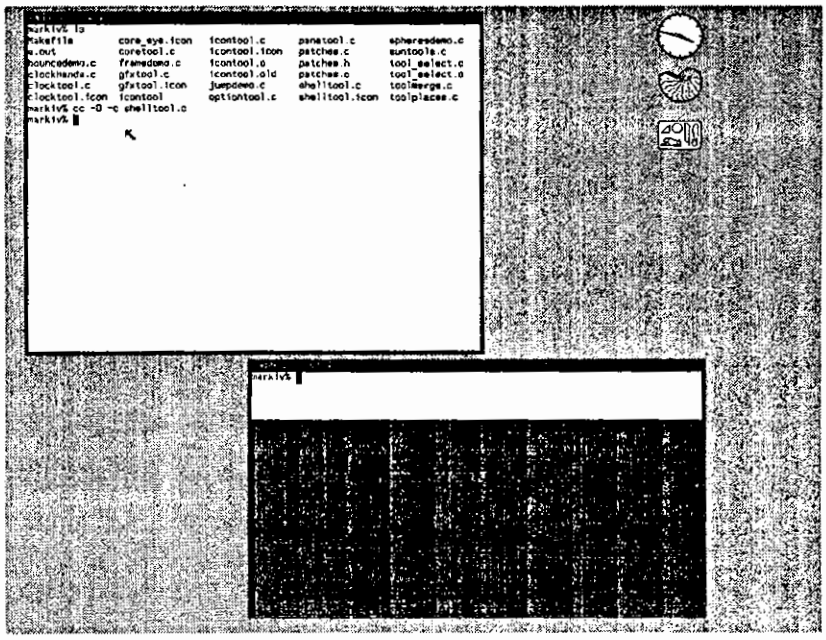


Figure 1-4: Sample SunWindows Screen Display

The tool with the large window on the left is a terminal emulator which is running the C-Shell. This tool is known as the shell tool. Its terminal emulation facility provides backward compatibility with traditional terminal-oriented UNIX utilities. We see that the user has made a listing of the current directory and then compiled a C program. Note the arrow that points upward and to the left in this window. This is the cursor that tracks the mouse.

The tool on the right that has the white and gray subwindows within the window is a graphics tool. The white subwindow is also a terminal emulator, albeit for a terminal with a smaller

screen. The gray subwindow is a blank canvas available to graphics programs. In *Writing a Simple Tool*, we examine code that implements the graphics tool.

The three small windows on the far right edge of the display belong to three different tools. Each tool has changed its window's presentation to an iconic form. Thus, from the top down, we see icons for a clock tool, for another shell tool, and for another graphics tool.

This figure also shows a window that underlies the rest, and is partially obscured by the other windows. This window belongs to the *suntools* program, the user-level program that initializes the SunWindows environment. It is the window that covers the entire display with background color. Because it is the window on which the others are displayed, it is called the *root window.*

The SunWindows system supports overlapping windows; one window can cover up another partially or completely. The user can stack windows however he wants. He can bring one window to the top of the stack, hide one underneath, and make a window as large as the screen. The user can manipulate the physical area of the screen to accommodate his needs. Note that all windows can be active at the same time, not just the one on top. Moreover, multiple display devices can be attached, with windows simultaneously displayed on each. SunWindows also supports both monochrome and color screens.

The next figure shows two examples of menus that use the SunWindows menu package. The menu on the left is available in the root window. The Tool Manager menu on the right is available in a portion of every tool window. SunWindows menus are called *pop-up* because they appear quickly out of the background, while the user is pressing one of the mouse buttons.

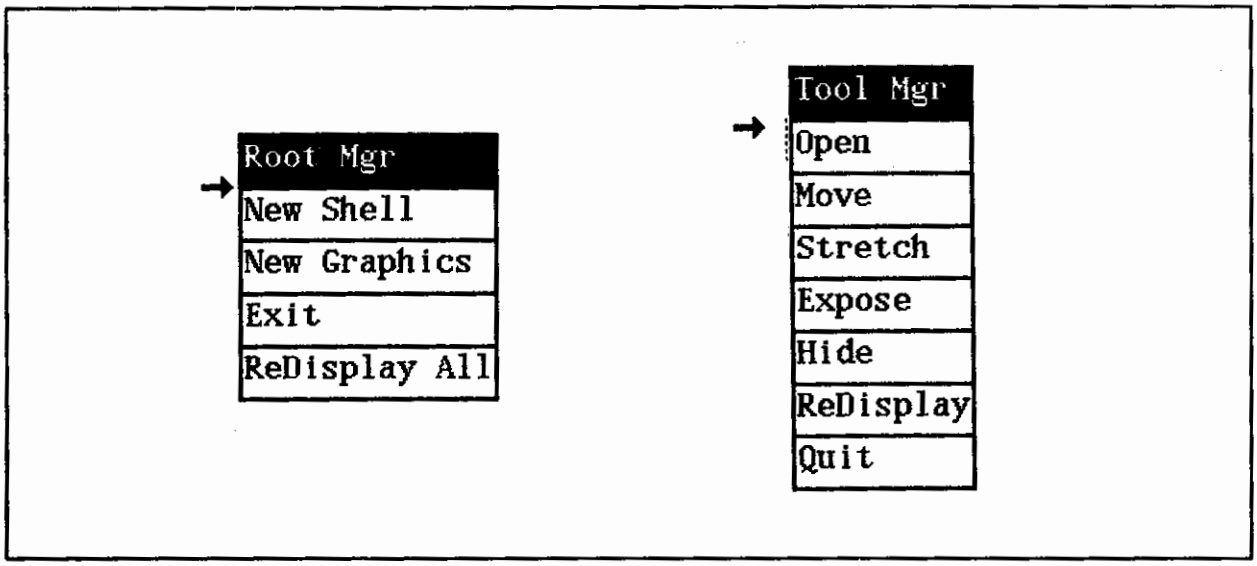


Figure 1-5: Pop-up Menus

Choosing the "New Shell" item in the "Root Mgr" (short for Root Manager) menu creates another shell tool.

Now examine the following figure, which shows the state of the display following the creation of a new shell tool. Note that the new shell tool window, the third shell tool window on the display, overlaps the other two open tool windows.

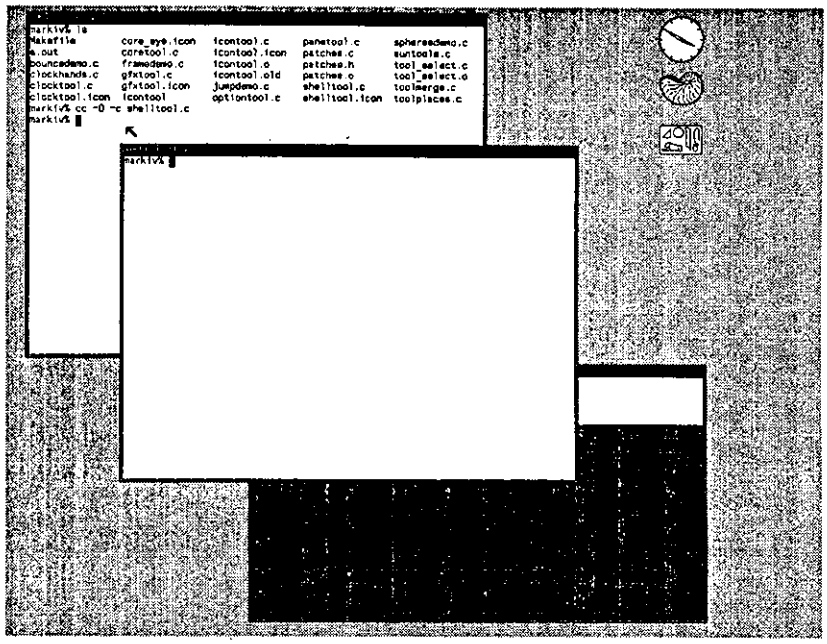


Figure 1-6: Adding Another Shell Tool

One shell tool can be used to compile a program, a second to edit its source code, a third to read and answer mail, and so on. Every tool keeps all of the context associated with its task. Although such multi-tasking can also be achieved through C-Shell job control, having the context of each task visibly represented concurrently is only available in the window system. If the number of concurrent tasks becomes so large that the display becomes cluttered, the user can simply shrink or make into icons those tool windows that are least interesting. Thus, the clock tool unobtrusively provides the current time, just one of many possible background tasks providing feedback on the state of the workstation.

More comprehensive explanations of the facilities provided with SunWindows are available in *suntools*(1). *Sample Tools* in this guide provides the programmer's perspective of these facilities.

In brief, SunWindows is:

an open architecture
: the user can extend the collection of available tools.

a database of windows
: that are simultaneously active.

simultaneous multiple screens
: the user can run SunWindows on more than one monitor.

color and monochrome screens
: for running both color and black-and-white programs.

overlapping windows
: that increase the user's effective working space.

icons for providing a visual reminder of a tool or process.

pop-up menus
for quick entry of commands.

terminal emulation
for UNIX compatibility.

text transfer between windows
for quick invocation of commands and text entry.

mouse and display entry
of commands and parameters.

Chapter 2

SunWindows Implementation Overview

In the previous chapter we looked at SunWindows from the user's perspective. Here we investigate SunWindows from the programmer's point of view. In particular, we describe the major components of the SunWindows implementation and their interactions.

Note the distinction between a *user* — a person who *uses* SunWindows, and a *programmer* who writes programs to run in the SunWindows environment. In addition, we introduce the term *client* to designate software that uses some other software package. A software package presents a programmatic interface to its clients.

2.1. Architectural Principles

SunWindows is built according to four architectural principles that are important to you as a programmer:

Open-ended

SunWindows is a *tool box* and a *kit of parts* — you can create tools aimed at specific application areas by tailoring and gluing together existing SunWindows packages. These tools can access all the facilities of the workstation that are available to SunWindows itself. Thus SunWindows is not a closed system aimed only at a single idealized user.

Integrated

SunWindows provides standard packages with which you can expand the facilities available to the user. The standard packages impose a framework on components. By using these standard packages and working within the framework, expanded facilities that you or other people write will exhibit the same level of user-interface integration that the standard Sun tools provide. In addition, the package-level integration will be consistent across different implementations.

Layered Implementation

SunWindows is a selectively layered system. The highest (most abstract) layer is called *suntool* — a collection of utilities that provide a framework and parts kit for constructing user interfaces. The middle layer, called *sunwindow*, provides facilities to share and arbitrate display and input devices between concurrent programs. The lowest (closest to the workstation's hardware) layer, called the *pixrect* layer, provides primitive access to the Sun Workstation's display. In general, the highest and most abstract layer has more functionality and generality at some expense in efficiency. Later sections of this chapter describes the layers in detail.

Information Hiding

Many of the major packages of SunWindows are implemented with the concepts of *data*

abstractions in mind. A data abstraction is a collection of subroutines and private data structures — only the subroutines access the data structures, and the interface to the data abstraction is defined entirely in terms of the interface that the subroutines provide. This practice is encouraged to provide flexibility of implementation for both SunWindows itself and for programmers writing new tools using SunWindows facilities. For example, SunWindows' lowest level provides device-independent access to bit-mapped devices in a framework where new devices can be added with no impact on any existing code using this level. Additionally, by hiding the implementation information, SunWindows minimizes the impact on existing code due to reimplementation of the support for a particular bit-mapped device.

2.2. Layers of Implementation

As we mentioned above, there are three layers in SunWindows. Each layer has an associated library of C routines which you can use to create your programs. The three layers, from the most abstract to the most system-dependent are:

suntool
: implements a multi-window executive and application environment, supporting many user interface facilities. These facilities include pop-up menus, selections, icons and several subwindow packages supporting terminal emulation and mouse and display entry of commands and parameters. The associated library is *libsuntool.a*.

sunwindow[1]
: implements a manager for overlapping windows. This management includes creating and manipulating windows, maintaining the windows' images, a stream input format for keyboards and mice, and distribution of those user inputs. The associated library is *libsunwindow.a*.

pixrect
: provides a device-independent interface to pixel operations. The associated library is *libpixrect.a*.

[1] Note that the term 'sunwindow' refers to the layer or level of implementation while the word 'SunWindows' is the name of the Sun window system.

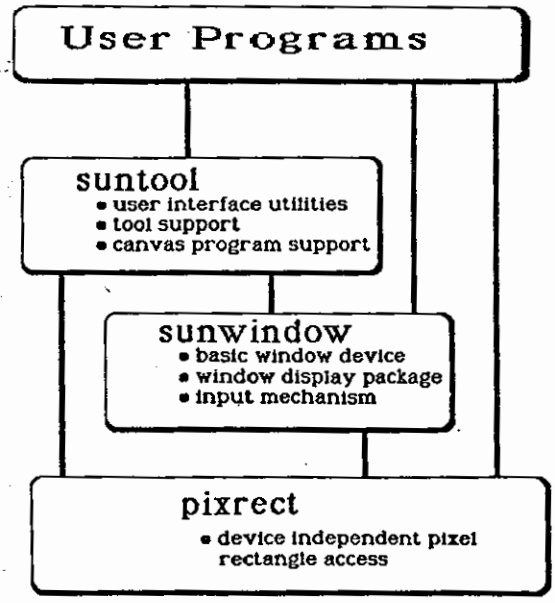


Figure 2-1: SunWindows Layers

As you can see from the figure, there is programmatic access to any or all of these levels.

2.2.1. Suntool Layer

The *suntool* layer provides user interface utilities. The name *suntool* indicates that a client (an application program) at this level is a tool. Such a client presents a complete user interface oriented to a particular application.

Several tools are provided with SunWindows. These include:

the shell tool
: a terminal emulator

the graphics tool
: which provides display space to simple graphics programs,

the icon tool
: used to create and modify cursor and icon images, and

the clock tool
: a simple tool that continuously updates a display of the time of day.

We describe these ready-built tools in more detail in *Sample Tools* and provide sample code in appendix B in the *Programmer's Reference Manual for SunWindows* and in */usr/suntool/src*. We describe how to write your own tools in *Applications — Tools and Canvas Programs.*

The user interface utilities that the *suntool* layer provides are:

a standard tool manager
: for the region of the display belonging to the tool. This region is enclosed by the "window which is the tool's frame," a phrase we will shorten to "tool window." This *tool window* identifies the tool by a name stripe at the top of the window and places

borders around the enclosed subwindows. It also generates a default icon for the tool if the tool writer does not provide one.

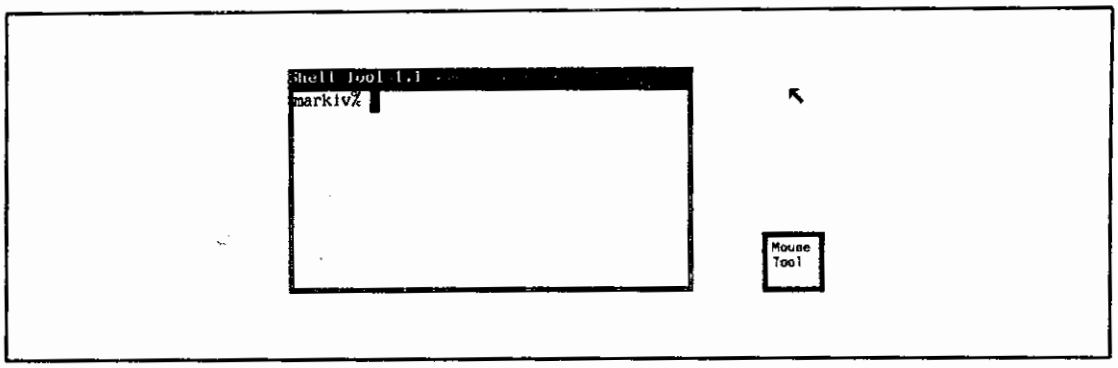


Figure 2-2: Standard Tool Window and a Default Icon

window management
A collection of routines for manipulating the position, size and overlapping structure of windows. These routines constitute the heart of a window manager for tool windows and subwindows.

an executive framework
which supplies the main loop of a client program and coordinates the activities of its various subwindows.

a menu package
that implements pop-up menus.

a simple prompting facility
that displays client messages to the user.

full screen access
for temporarily overriding the input and output hierarchies. Applications of this full screen access are menu display and item choice, making screen dumps, and displaying prompts, often in conjunction with awaiting user confirmation of some action.

global text selection
for specifying a span of text that may be of interest across window boundaries. This selection is typically used to make copies within or between windows. Graphic selections are supported with this mechanism.

Also provided are several subwindow types that can be incorporated in the tool, and an implementation of a simple tiling mechanism for subwindows. The provided subwindow types are:

terminal emulator subwindow (ttysw)
that provides emulation of a "smart" Sun terminal;

graphics subwindow (gfxsw)
for programs that want to display graphics in the SunWindows environment without undertaking all the responsibilities of a standard tool. Such programs are called *canvas programs*. The graphics subwindow also provides the ability for a program to run on top of an existing window.

option subwindow (optsw)
which provides sophisticated mouse and display entry of commands and parameters. It is the window system analog to entering command-line arguments and typing

mnemonic commands to an application. An option subwindow contains a number of items of various types, each item corresponding to one parameter. Existing option item types include labels, booleans, enumerated choices, text parameters, and command buttons.

message subwindow (msgsw)
for displaying textual messages such as error messages and prompts to the user.

empty subwindow (esw)
for tending a window that will be covered by another window. The empty subwindow is thus a place holder.

The following figure shows each of the subwindow types:

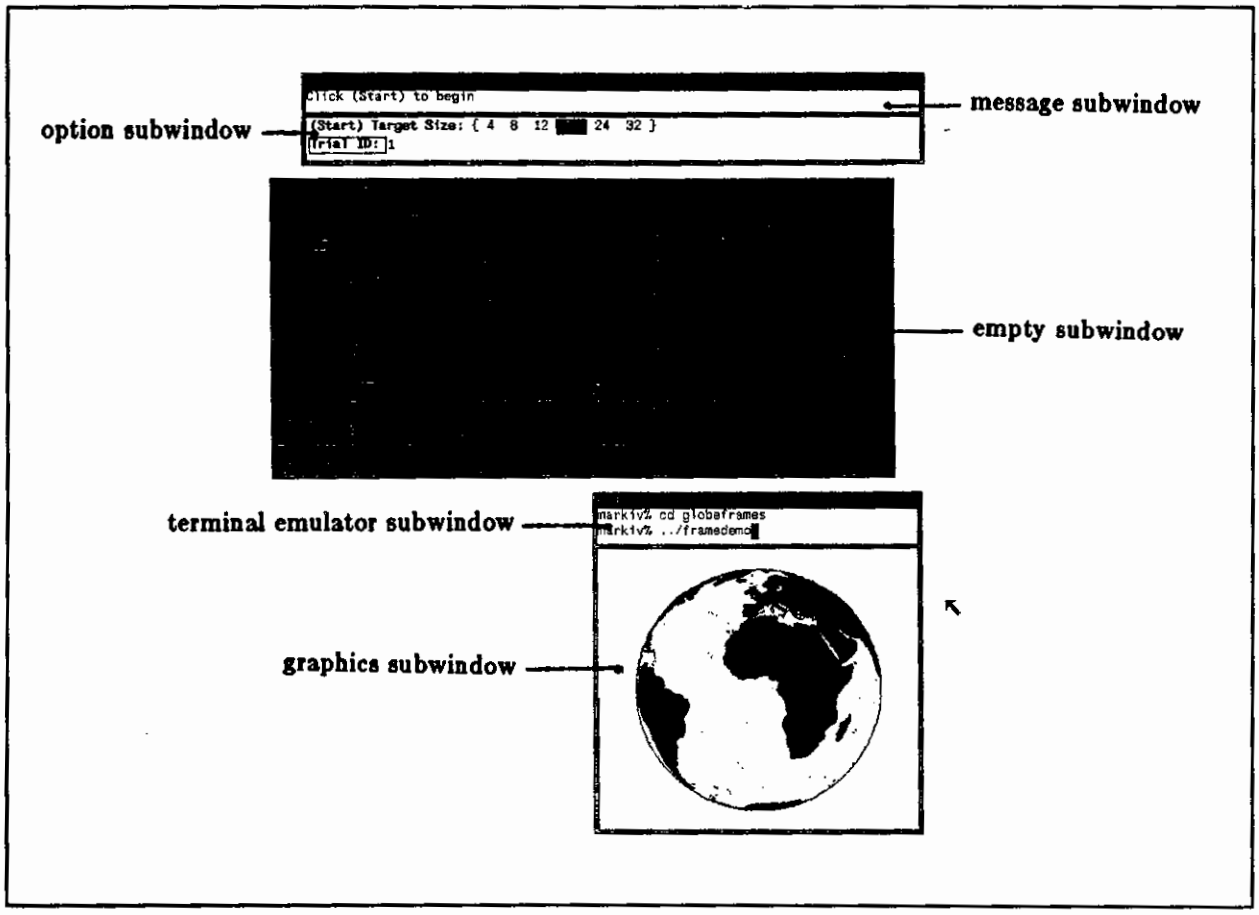


Figure 2-3: System Subwindow Types

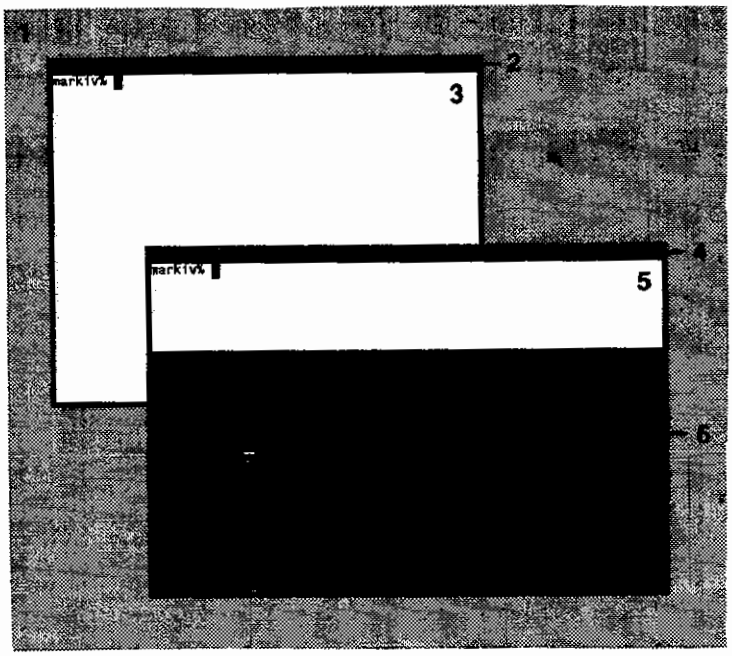
You can also make a custom subwindow from a skeletal version, as we describe in *Writing a More Sophisticated Tool* and in *Additional Topics*.

2.2.2. Sunwindow Layer

The *sunwindow* layer of the system maintains a database of windows. This database is structured as a collection of trees, with one tree per display device. Each tree has a root at the top and descendants toward the bottom. We will use the metaphor of a family tree to describe this

database, since it provides a convenient terminology, and the concept of *age* is useful for describing the locations of windows in the tree. Note, however, that the metaphor is not exact: it is possible to change a window's position in the tree, hence a window's age is subject to change.

When one window is located directly above another in the tree, the first is called the *parent*; the second is the *child*. A window may have any number of children, but only one parent. All the child windows of a parent are called *siblings*. Parents are older than their children, and siblings have distinct ages, which establishes a full order on them (no twins). The window at the top of the tree is called the *root* window; it is the ancestor of all other windows in the tree. The following figure shows these relationships.



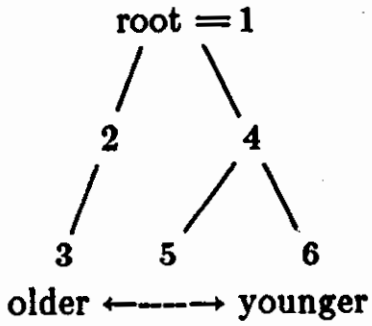


Figure 2-4: Window Tree

Each window in the database occupies a region of the display. Child windows usually occupy a region completely contained within their parent's region, but should a child's region extend beyond the parent's region, the excess portions of the child's region are not visible to the user. Thus, the child's visible region is clipped so it does not extend beyond its parent's region.

When two or more windows have regions with a common area, they are said to overlap in that area. When windows of direct descent overlap in a common area, the youngest of the overlapping windows is visible to the user in that area of the display. Thus, the youngest window obscures its ancestors where they overlap. When two siblings overlap, the younger sibling obscures the older sibling. In addition, the younger sibling obscures all of the older sibling's descendants. By recursively applying these rules, the visible portions of each window are computed.

The sunwindow layer provides facilities to create, destroy, move, stretch and shrink windows. It provides for repositioning a window at different places in the window tree.

The sunwindow layer allows definitions of a different cursor image to track the mouse in each window. It also provides inquiry and control over the mouse position.

The sunwindow layer provides locking primitives to enable clients to arbitrate access to the display. Such arbitration is necessary because two or more clients, each running in a separate user process, may be painting into windows that share a common region of the display. To guarantee that the user sees the correct display image where the windows overlap or clip, these separate processes need to have a consistent idea about the positions, sizes and relationships of the windows in the window tree. The locking primitives ensure that one client cannot change the window tree while another client is either modifying the window tree or painting to the display.

Various events can make a window's image incorrect. For example, the following figure shows two overlapping windows. When the bottom one is brought to the top, the area that was covered by the top window must be repaired. This area is indicated by the dotted line in the following figure.

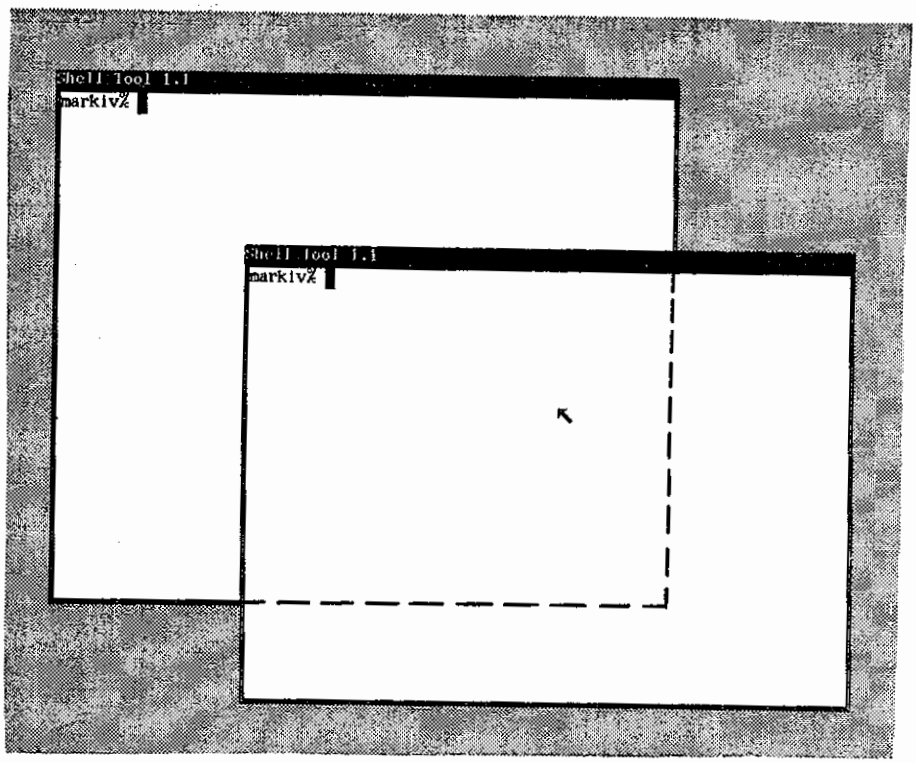


Figure 2-5: Damage

Other examples of damage are the window growing bigger, the window changing from iconic to standard presentation, or some other window being destroyed that previously had obscured the window. Most of these events occur in sync with the window's standard processing, but events such as another window being destroyed are asynchronous. The incorrect portions of a window's image are known as *damage*. This damage is always composed of previously hidden areas of a window that have become exposed. When a window's image becomes incorrect, the window owner process is notified of this problem via UNIX's asynchronous signaling mechanism. In particular, a SIGWINCH signal is delivered to the window owner process. Later, when the window process decides to correct its window image, the sunwindow layer provides calls to determine the current damage for the window. Windows may either recompute their contents for redisplay, or they may elect to be *retained*. A retained window has a full backup of its image in

main memory, and need merely copy this backup to the display when required. SunWindows also provides facilities for colormap sharing on color displays.

The user interacts with multiple windows via a single keyboard and mouse. User inputs are unified into a single stream at this level, so that actions with the mouse and keyboard can be coordinated. In particular, input events are time-stamped and entered into the queue in the order they occur, independent of the device which generated them. This unified stream is then distributed to different windows, according to user or programmatic indications. Windows may be selective about which input events they will process, and rejected events will be offered to other windows for processing. This enables terminal-based programs to run within windows which will handle mouse interactions for them. Separate collections of windows may reside on separate screens, and the user input facilities treat them as if they were all on one huge screen.

2.2.3. Pixrect Layer

The *pixrect* layer of the system provides a uniform interface to devices which can hold raster images, such as bit-mapped displays and memory. The pixrect layer defines a standard set of operations on pixels; regardless of the actual device containing the pixels, each operation is invoked in the same fashion and has the same results. This is similar to UNIX's system interface to files, with a single set of operations being used to manipulate file descriptors independent of the underlying file implementations. The pixrect layer provides a way of defining a rectangular array of pixels on a device and then binding to the array the specific procedures which provide the general pixrect operations for that device. A particular pixrect (for *pix*el *rect*angle) is a combination of one of these arrays of pixels and the operations used to manipulate the array. This layer of SunWindows is named after the pixrect structure because that structure is central to the entire layer.

The concept of a pixrect is very general. A particular pixrect may refer to an entire display, or to an image as small as a single character in a font, or to a particular cursor image. The array containing the pixels may be visible on a display, or be stored in memory or (conceivably) on some mass storage device. Peculiarities of specific devices are hidden below the pixrect interface: some displays use only a single bit per pixel, and display black on white, or green on black; others user three, eight, or twenty-four bits to describe the color of each pixel. Some displays address pixels in two dimension, with an origin in the upper left, or bottom left, or center of the screen; other displays, and most forms of memory, address pixels in a linear fashion, with the first pixel of one row immediately following the last pixel of the preceding row. Some devices provide hardware support for common operations, while others require all operations to be performed in software. To the programmer using pixrects, all pixrects are described in the same way and manipulated by the same operations.

The operations supported by pixrects include: single pixel reads and writes, writing vectors, and a variety of *RasterOps* (each of which determines its resulting image by a logical function of corresponding pixels from source, destination, and (sometimes) mask pixrects). Color pixrects provide operations for manipulating a *colormap*, which translates a pixel value to a specific color, and for isolating the bits of a color pixel's value, so that an image may be treated in *planes* which can be operated on independently. Monochrome pixrects provide the same operations, without doing very much for them; thus images designed for color displays generally produce reasonable results on a monochrome display, and vice versa. Where hardware support exists for a pixrect operation, the implementation takes advantage of it to provide increased efficiency; but the generality of the interface is not sacrificed. See *Pixel Data and Operations* in the *Programmer's Reference Manual for SunWindows* for details on the pixrect layer.

A new device may be incorporated in the pixrect layer, by providing a new implementation of the basic operations. The process is akin to adding a new device to the kernel, with the advantage that most of the pixrect implementation is in user code.

2.3. Choosing the Appropriate Layer

When you start writing applications to make full use of the bit-mapped display's capabilities, you have a number of choices. One of these choices is which level of the SunWindows facilities to use:

- If you only want to modify the basic user interface presented by the *suntools* program, you can simply re-write *suntools* while continuing to use all of the other SunWindows facilities.
- However, if you want to modify the basic appearance of the individual tools, you must replace portions of the *suntool* level as well. In general, you can replace individual packages in the *suntool* level rather than discarding the entire level.
- If you want to replace the entire user interface that comes with SunWindows, you must rewrite both the *suntools* program and all of the *suntool* level. However, you can still use the basic input and output sharing and arbitration facilities of the *sunwindow* and *pixrect* levels.
- If you don't care about the sharing and arbitration between concurrent processes, you can discard the *sunwindow* level and simply use the *pixrect* level, thereby keeping the device-independent pixel access provided by the the *pixrect* level.

You are encouraged to use the highest possible layer of the SunWindows interfaces. Only after careful consideration should you delve into the lower layers, as it lessens the overall integration of your code with the rest of the system.

Several other interesting possibilities exist: For programs that want to run standalone, you can use facilities described in the next chapter that allow a program using only a single window to be developed and executed within *suntools* and then also run outside *suntools*. Many of the Sun demonstration programs fit this category and use this facility. For instance, you can run *bouncedemo*, the bouncing ball demonstration, outside of *suntools*.

Finally, SunCore is an alternative if SunWindows doesn't provide what you want. SunCore is the Sun implementation of the ACM Core graphics standard. The SunCore library provides routines for:

- Three-dimensional floating-point coordinate systems with hidden-surface elimination provided by the system.
- Flexible viewing transformations and scaling of input coordinates.
- A rich set of primitives such as polygons and shading.
- Display-list segmentation.

Programs using SunCore are more portable than those using SunWindows, but the extra generality and sophistication of the Core is computationally expensive. If you decide to use SunCore, you need not lose all of the advantages of a window system because SunCore programs can run both inside and outside of SunWindows. See the *Programmer's Reference Manual for SunCore* for further details on SunCore.

Chapter 3

Applications — Tools and Canvas Programs

This chapter introduces the basic concepts that high-level SunWindows applications programmers need to know. Applications are divided into categories and the programmatic implications of writing programs of a particular class are discussed. The definitions of terms introduced in previous chapters are refined here to the level required for writing SunWindows client programs.

Note: Window code can only be written in the C programming language.

3.1. Applications Concepts

A SunWindows program falls into one of three categories, depending on the relationship that the program has with windows:

Tool A tool usually creates and owns more than one window. It has a mechanism for dealing with multiple windows within a single user process. Most application programs in SunWindows are tools. *Gfxtool*, which we describe in detail later, is a simple example of a tool, containing a terminal subwindow and a graphics subwindow.

Canvas program
A *canvas* program is characterized by its creating and managing only one window. Since a canvas program owns only one window, the mechanism required for dealing with it is simpler than for a tool. For example, *bouncedemo* owns a graphics subwindow in which it paints a bouncing ball.

Non-interactive utility
A non-interactive utility is typified by not creating any windows. Instead it queries and manipulates the state of one or more existing windows. Since this type of program does not own any windows, it never has to deal with window input or SIGWINCH signals. *Toolplaces*, described later in this chapter, is a simple example of a program that just finds out where windows are on the screen. *Screendump* is another non-interactive utility that takes the screen image and dumps it to a printer for hardcopy output.

The process that is responsible for displaying a window's image and reading its input is said to *own* that window. The relationship between programs (user processes) and windows varies; there is no one-to-one correspondence between them. If a window's size, position or relative relationship with other windows is changed, some sort of window management operation has been performed. Sometimes it is a window's owner and sometimes some external agent that invokes window management activity.

It is important to make the distinction between a window and a SunWindows program. Windows are pseudo-devices, referenced as */dev/winxx*, where *xx* is the number of the window. A

window is the operating system's handle on a specific area of the screen. UNIX uses a window to multiplex user input and screen output among competing user processes. Like other UNIX devices, a window can be opened, closed, read, waited on for input, and given input/output control commands (through a procedural interface).

Most of the software interfaces in SunWindows follow a common paradigm. First, a *client* will call some creation/initialization routine, which returns a pointer to some data, typically a C structure or a nested group of such structures. Then it calls other procedures to perform specific tasks, generally passing that pointer as an argument which provides state information.

We refer to the implementation of such an interface as a *manager*, the data which it provides and manipulates as an *object*, and the pointer to that data as a *handle*. Strictly speaking, the *object* is the general class that the manager deals with, a specific copy of which is called an *instance*.

Thus, a client will call on a *tool manager* to create a new instance of a *tool object*; the manager returns a *tool handle* which points to a particular instance of the tool object. The client then may call various *subwindow managers*, passing each the tool handle, to have a number of subwindow objects added to the tool object.

Note that the set of things with which a program concerns itself depends on a program's relationship with windows.

3.2. What is a Tool?

A tool consists of two or more windows owned by a single user process. SunWindows provides a mechanism, called the *tool manager*, for multiplexing multiple windows. A tool manager owns one window, the tool window, which defines the maximum extent of the display region that the tool controls. The tool manager usually displays the name of the tool in a stripe at the top and a border all around the region of its tool window. Subwindow(s) occupy areas within this tool window and move with it when it is moved.

Each subwindow object is a building block for constructing a tool. A tool's subwindow managers provide most of the user interface to the tool's facilities. SunWindows defines a minimum interface that a software implementation must provide in order to properly be called a *subwindow package*. Any subwindow package can be plugged into a tool object if it meets the programmatic interface requirements of such a package.

3.2.1. Designing a Tool

One of the major thrusts of SunWindows is to offload user interface design from each and every application. The intent is that most programmers will use the existing packages for most of their user interface activities. This means that the programmer is able to concentrate on what is unique about his application. It also means that because applications are using common packages, the user interface will be highly consistent across applications.

Tool design is a matter of deciding:

- How should you lay out the available tool window real estate into subwindows? As examples, look at the tools supplied with SunWindows in *Sample Tools* below and try running those tools in */usr/suntool/*tool*. Many tools place a message subwindow across the top, an option subwindow beneath for specifying parameters and invoking commands, and below that, an application-specific subwindow.

- Which SunWindows-supplied subwindow packages will you be able to use? The available subwindow packages are listed in *SunWindows Implementation Overview.*
- What subwindow implementation(s) will you have to provide? The SunWindows supplied *mousetool* and *icontool* provide their own subwindow implementations.
- What SunWindows supplied facilities can you use in your own subwindow implementation? Packages which provide pop-up menus, prompts, a selection manager, and icons are available from the suntool library. Mouse cursor shape and input mask control are available from the sunwindow library. The pixwin display package is available for accessing the screen in this overlapping window environment.
- How do you unify all these user interface pieces with your program's purpose to produce a harmonious application? Again, the SunWindows-supplied *mousetool* and *icontool* are good examples of tying together multiple subwindows into a single application.

3.2.2. Flow of Control in a Tool

When you get to the point of writing code, note that all tools follow the same basic steps:

Create a tool object
: Some of the properties of the tool object are specified at create time.

Create a subwindow object
: for each subwindow and associate it with the tool object. The initial state of each subwindow object is specified at this time.

Install the tool window
: into the database of windows, so that the tool window and its subwindows can receive input and display themselves on the screen.

Invoke the main loop
: of the program. The tool manager notifies each of the subwindow managers and the tool manager itself of pending input and window changes.

Perform a clean-up step
: to release resources associated with the various objects when the tool is terminated normally.

Writing a Simple Tool provides a concrete example of tool writing that details these steps.

3.2.3. Sample Tools

Here is a list of existing tools along with the basic facilities that they use. You may want to refer to them as examples when writing your own applications.

shelltool Uses a single terminal emulator subwindow.

gfxtool Uses a terminal emulator subwindow and an empty subwindow in which canvas programs can be run without overwriting the terminal text.

icontool Uses an option subwindow for parameter specification and command invocation. A message subwindow is available for error messages. Two client-defined subwindow implementations are used. This tool illustrates mouse tracking and cursor shape alteration. You may use this tool to generate cursor and icon images for your own applications.

panetool Uses four message subwindows. This tool serves as an example of stacked menus and input redirection.

optiontool Uses one option subwindow. This tool serves as an extensive example of option subwindow possibilities.

clocktool Uses one message subwindow to display the time and date. This tool also shows how to dynamically change the tool window's icon.

coretool Uses a single empty subwindow. This program is usually invoked programmatically by SunCore and serves as a view surface for CORE programs with independent window management.

mousetool Uses a message subwindow for status messages, and an option subwindow for parameter specification and command invocation. One client-defined subwindow implementation is used. This is discussed in detail in *Writing a More Sophisticated Tool*.

The source code for these programs is available in */usr/suntool/src/*tool.c*.

3.3. Canvas Programs

As defined above, canvas programs only create one window. This has the advantage of not requiring multiplexing of notifications to multiple windows, a considerable simplification over a tool. Canvas programs can be categorized into those that are interactive and those that are not. Not surprisingly, non-interactive canvas programs are easier to write because they do less.

3.3.1. The Graphics Subwindow

One mechanism often used to write canvas programs is the graphics subwindow package. It accommodates both interactive and non-interactive canvas programs. The flow of control for interactive programs is essentially the same as for tools. The main difference is that instead of there being a tool manager and multiple subwindow managers, there is just the graphics subwindow manager. For non-interactive canvas programs, the flow of control is usually an infinite loop in which some flags are checked in order to know when to respond to window management operations.

The graphics subwindow manager provides the following features:

- A graphics subwindow can be used to take over the area of an existing window by laying itself on top of the existing window. When the existing window is manipulated the graphics subwindow is changed as well. This frees the canvas program of any concern for providing window management operations to the user.
- A graphics subwindow-based program can be run both inside and outside of the *suntools* window environment.
- A graphics subwindow manager will manage a retained window for you.

Writing a Simple Canvas Program provides a concrete example of using the graphics subwindow.

3.3.2. Sample Canvas Programs

Here is a list of existing canvas programs along with the basic facilities that they are using. You may want to refer to them as examples when writing your own applications.

bouncedemo, jumpdemo, spheresdemo
Non-interactive graphics subwindow based programs.

framedemo Interactive graphics subwindow based program.

suntools Interactive canvas program not using the graphics subwindow package. The program shows simple menu use. This program does many other things that are related to initializing the window environment.

The source code for these programs is available in */usr/suntool/src/*demo.c.*

3.4. Non-interactive Utility Programs

A SunWindows non-interactive utility program doesn't create any windows. Instead, it queries or changes the state of existing windows. Since this type of program does not own any windows, the problems associated with user input and screen output are not present.

3.4.1. Sample Non-interactive Utility Program

Toolplaces is a program that traverses the second level of a window tree and displays the positions of the windows on the screen. Sample output is:

```
markiv% toolplaces
toolname  160    80   730   532   956     0   64   64
            1
toolname  265   246   730   532    64   736   64   64   0
toolname   18   226   730   567   959   736   64   64   0
toolname  263    14   730   532     0   736   64   64   0
markiv%
```

These coordinates are read with the origin (0,0) in the upper lefthand corner, *x* increasing to the right, and *y* increasing down.

The source code for ***toolplaces*** is available in */usr/suntool/src/toolplaces.c.* Refer to ***suntools***(1) for more details.

3.5. Review

In this chapter, you have been introduced to the following points:

- What the software terms ***object, handle, instance,*** and ***manager*** mean.
- A window is different from a window-related program. The basic phrases directly related to windows are ***window owner*** and ***window management.***
- When considering writing a SunWindows application, you should have an idea as to whether it will best be done as a ***tool, canvas program,*** or ***non-interactive utility.***
- When considering writing a SunWindows application, you should have an idea of programs to go to for sample code that matches your application. Note that detailed descriptions of tool

and canvas examples are presented in later chapters.

Chapter 4

Writing a Simple Tool

Here we describe how to write a tool, and illustrate what a *window*, a *tool object*, and a *subwindow object* are.

We walk through the C language source code for a program called the *gfxtool* (*gfx* stands for *graphics*). We illuminate and give substance to broader concepts sketched out earlier. Each line in this program is explained; however at times you are asked to accept descriptions that don't go into much detail. Detail is covered in the reference manual.

Gfxtool runs in the SunWindows environment and consists of two subwindows:

a terminal emulator subwindow
: from which canvas programs can be run

an empty subwindow
: in which canvas programs can draw. Thus, terminal text is kept separate from the graphic image, and the graphics do not disturb the characters in the terminal emulator. Note that *gfxtool* does not actually generate any interesting graphics, rather it provides a virtual terminal and a blank canvas where a graphics application can draw pictures.

Gfxtool is a very simple tool, being composed entirely of glue that brings together existing subwindow packages into a single entity. Understanding the concepts in this program is critical to being able to do more interesting applications.

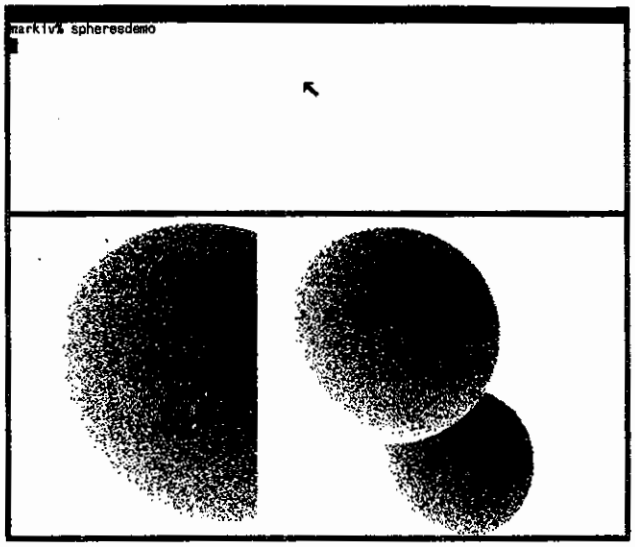


Figure 4-1: *Gfxtool* Running *spheresdemo*

4.1. The *gfxtool* Code

As described in *Flow of Control in a Tool*, *gfxtool* has some distinct parts, namely:

- *Gfxtool* creates a tool object.
- It then creates a terminal emulator and an empty subwindow.
- It installs the tool.
- It calls the main loop.
- It cleans up.

Here is a listing of *gfxtool.c*. You may want to glance at it now, however, it is primarily for reference as you read the subsequent explanation. Extensive C comments are removed in favor of the accompanying text.

```
#ifndef lint
static  char sccsid[] = "@(#)gfxtool.c 1.1 84/04/04 SMI";
#endif

#include <suntool/tool_hs.h>
#include <suntool/emptysw.h>
#include <suntool/ttysw.h>
#include <stdio.h>    /* Source of NULL */


/* define global data */
#include "gfxtool.icon"
mpr_static(gfxic_mpr, 64, 64, 1, icon_data);
static  struct icon icon = {64, 64, (struct pixrect *)NULL, 0, 0, 64, 64,
          &gfxic_mpr, 0, 0, 0, 0, (char *)NULL, (struct pixfont *)NULL,
          ICON_BKGRDGRY};
```

```
static  sigwinchcatcher(), sigchldcatcher();
static  struct tool *tool;

main(argc, argv)
        int argc;
        char **argv;
{

/* declare local variables */
        char    *toolname = "Graphics Tool 1.1";
        struct  toolsw *ttysw, *emptysw;
        char    name[WIN_NAMESIZE];

        /* create tool */
        tool = tool_create(toolname, TOOL_NAMESTRIPE|TOOL_BOUNDARYMGR,
           (struct rect *)NULL, &icon);
        if (tool == (struct tool *)NULL)
                exit(1);

        /* create subwindows */
        ttysw = ttysw_createtoolsubwindow(tool, "ttysw",
           TOOL_SWEXTENDTOEDGE, 200);

        emptysw = esw_createtoolsubwindow(tool, "emptysw",
           TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE);
        if (ttysw == (struct toolsw *)NULL ||
           emptysw == (struct toolsw *)NULL)
                exit(1);

        /* install the tool */
        signal(SIGWINCH, sigwinchcatcher);
        tool_install(tool);

        /* start child process */
        win_fdtoname(emptysw->ts_windowfd, name);
        we_setgfxwindow(name);
        signal(SIGCHLD, sigchldcatcher);
        if (ttysw_fork(ttysw->ts_data, ++argv,
           &ttysw->ts_io.tio_inputmask,
           &ttysw->ts_io.tio_outputmask,
           &ttysw->ts_io.tio_exceptmask) == -1) {
                perror("gfxtool");
                exit(1);
        }

        /* notification loop */
        tool_select(tool, 1 /* means wait for child process to die*/);

        /* clean up */
        tool_destroy(tool);
        exit(0);
}

static sigchldcatcher() { tool_sigchld(tool); }

static sigwinchcatcher() { tool_sigwinch(tool); }
```

4.2. Include Files

The header files included for this program are:

```
#include <suntool/tool_hs.h>
#include <suntool/emptysw.h>
#include <suntool/ttysw.h>
#include <stdio.h>  /* Source of NULL */
```

A brief description of each of these follows:

<suntool/tool_hs.h>
includes the header files needed to deal with the tool manager and subwindow managers. The *_hs* suffix to "tool" stands for HeaderS and is a convention used by SunWindows to indicate that this header file includes other header files. A header file that includes other header files simplifies the tedious job of determining exactly which header files are required and then ordering them based on their interdependencies. (This job can be especially difficult for software which is built in multiple layers, as is SunWindows.) The "suntool/" means that "tool_hs.h" is found in the */usr/include/suntool* directory.

<suntool/emptysw.h>
is a single header file that defines structures and declarations pertinent to the empty subwindow manager. In particular, an external reference to *esw_createtoolsubwindow*() is used.

<suntool/ttysw.h>
is a single header file that defines structures and declarations pertinent to the Sun terminal emulator subwindow manager. In particular, an external reference to *ttysw_createtoolsubwindow*() is used.

4.3. Defining Global Data

The global definitions provided are:

```
#include "gfxtool.icon"
mpr_static(gfxic_mpr, 64, 64, 1, icon_data);
static  struct icon icon = {64, 64, (struct pixrect *)NULL, 0, 0, 64, 64,
            &gfxic_mpr, 0, 0, 0, 0, (char *)NULL, (struct pixfont *)NULL,
            ICON_BKGRDGRY};

static  sigwinchcatcher(), sigchldcatcher();
static  struct tool *tool;
```

The definitions are made global to the module to:

- take advantage of global structure initialization,
- declare the types of procedures referenced later (*sigwinchcatcher* and *sigchldcatcher*), and
- share data between procedures (*tool* structure).

The data structures define the icon displayed when the tool window is tiny:

- *"Gfxtool.icon"* contains the data structure *icon_data* that defines the bit pattern of the graphic image displayed in the icon. The include file "gfxtool.icon" is output from a specialized bitmap editor tool named *icontool* that is distributed with SunWindows.
- A *memory pixrect* is a data structure that describes raster images in main memory. The *mpr_static* macro constructs a memory pixrect containing the image data of *icon_data*. The

constructed memory pixrect is called *gfxic_mpr* and is 64 bits wide by 64 bits high by 1 bit deep. The depth of a pixel is the number of bits in a pixel. This number controls how many values, black and white or color, the pixel can display.

- *Icon* is a structure which describes the layout of the icon. The icon includes a background (ICON_BKGRDGRY), graphic image (*gfxic_mpr*) and text tag (NULL implies there is no text associated with this icon). The other fields in the structure are position and size data that you can find more about in the reference manual.

4.4. Declaring the Local Variables

The variable definitions are:

```
main(argc, argv)
        int argc;
        char **argv;
{
        char    *toolname = "Graphics Tool 1.1";
        struct  toolsw *ttysw, *emptysw;
        char    name[WIN_NAMESIZE];
```

Main is the first routine called when a C program is started. *Argc* is the count of arguments to the program that are described as strings in *argv*. These arguments are ignored in *gfxtool.c* proper but are passed through to *ttysw_fork*.

Now the other local variables:

Toolname
is the name and version number of the tool passed to *tool_create*. This string is displayed in the tool window's name stripe.

Ttysw and *emptysw*
are the subwindow handles for the two subwindows we create.

Name
is a temporary variable that holds a window device name. A window is conventionally named "/dev/win#" where # is a decimal number between 0 and 255.

4.5. Creating the Tool Object

The first thing that the program does is create a tool object. *Tool_create* dynamically allocates a tool object and returns a tool handle (called *tool* here) that points to this tool object.

```
tool = tool_create(toolname, TOOL_NAMESTRIPE|TOOL_BOUNDARYMGR,
    (struct rect *)NULL, &icon);
if (tool == (struct tool *)NULL)
        exit(1);
```

- The tool manager takes care of displaying the tool window. When the tool window is its normal size, the tool window displays a name stripe and borders around its subwindows. *Toolname* is the text in the name stripe. TOOL_NAMESTRIPE indicates that the name stripe is to be displayed. When the tool window is tiny, it displays *icon*.
- Normally, the initial size of the tool window and its position on the screen is determined inside *tool_create* by examining the environment parameter WINDOW_INITIALDATA (see *we_getinitrect* in the reference manual). A program may override this default by specifying an

explicit position and size. In this case, the default is used because the position and size structure pointer is NULL.

- The tool object is a framework in which subwindow objects are collected. The tool manager is responsible for laying out the positions and dimensions of its subwindows so that they tile the surface of the tool window. The TOOL_BOUNDARYMGR flag tells the tool manager to let the user move the boundaries between subwindows by using the mouse to point at and drag the boundaries.

The *tool_create* routine can fail in a variety of unusual situations:

- Enough virtual address space couldn't be allocated.
- The program was started outside of the SunWindows environment.
- A free window device couldn't be found.

If *tool_create* fails, the returned pointer is NULL. An error message will have been written to *stderr* (the usual UNIX error stream) in this case. Thus, the appropriate action is to terminate the program. A non-zero argument to *exit* indicates abnormal termination.

4.6. Creating Subwindow Objects

Next, subwindow objects are created to populate the tool. A terminal emulation subwindow object and an empty subwindow object are created in the calls to *ttysw_createtoolsubwindow* and *esw_createtoolsubwindow*, respectively. The calling sequence to all subwindow manager creation routines is essentially the same:

```
ttysw = ttysw_createtoolsubwindow(tool, "ttysw",
    TOOL_SWEXTENDTOEDGE, 200);
emptysw = esw_createtoolsubwindow(tool, "emptysw",
    TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE);
if (ttysw == (struct toolsw *)NULL ||
    emptysw == (struct toolsw *)NULL)
        exit(1);
```

- The tool handle *tool* identifies the object in which the subwindow object is installed.
- The second argument is a string that names the subwindow instance and differentiates it from other subwindow instances in the same tool. Although not used now, this naming of subwindow instances may eventually allow manipulation of subwindow managers by scripts.
- The third (width) and fourth (height) arguments are hints to the tool manager's tiling mechanism that indicate the preferred dimensions of the subwindow. The constant TOOL_SWEXTENDTOEDGE indicates that the tiling mechanism is to extend the dimension of the subwindow to the right or bottom edge of the tool window. The tiling algorithm uses the order that subwindow objects are created to place the subwindows in English reading order. The first subwindow goes in the upper left-hand corner. Subsequent subwindows go to the right and then down to the next "row" depending on the constraints of the width and height hints.

Subwindow object creation calls that fail return a NULL subwindow handle. The types of problems that might be encountered are similar to those within *tool_create*. An error message will have been written to standard error in this case. Again, the appropriate action is to terminate the program.

Although not done here, at this point in the code one often calls routines to further specify subwindow instance behavior. For example, one would define the layout of items in an option subwindow manager here.

4.7. Installing the Tool

Up to this point, we have been building up structure under the tool object. The next thing to do is to install the tool window into the database of windows that are actually displayed on the screen:

```
signal(SIGWINCH, sigwinchcatcher);
tool_install(tool);
```

SunWindows uses asynchronous software interrupts to notify a program that one or more of the windows it controls needs to refresh part of its image. A program controlling windows needs to set up a procedure to receive this SIGWINCH interrupt. The call to *signal*(2) sets *sigwinchcatcher* as the procedure called when a SIGWINCH interrupt is sent to this program.

Now that the program is ready to handle the SIGWINCH interrupt, the call to *tool_install* causes the tool window and its subwindows to be included in the database of windows that are actually displayed on the screen. During the call to *tool_install*, a SIGWINCH signal occurs, and *sigwinchcatcher* is called (asynchronously). What *sigwinchcatcher* does when it is called is described below in *Notification Loop.*

4.8. Client Code

Most of the code in this program is boilerplate, meaning that almost every other tool has the same sequence of calls. Of course, the number and kind of subwindows vary from tool to tool. The code described in this section is different from the boilerplate code; it is specific to the *gfxtool* and its terminal emulator subwindow object set up. The code in this section starts the child process that interacts with the user through the terminal emulator subwindow:

```
win_fdtoname(emptysw->ts_windowfd, name);
we_setgfxwindow(name);
signal(SIGCHLD, sigchldcatcher);
if (ttysw_fork(ttysw->ts_data, ++argv,
    &ttysw->ts_io.tio_inputmask,
    &ttysw->ts_io.tio_outputmask,
    &ttysw->ts_io.tio_exceptmask) == -1) {
        perror("gfxtool");
        exit(1);
}
```

Typically, a canvas program runs under a shell process which runs under the tool. There is a problem of communicating the name of the window that the canvas program is to use from the tool to the canvas program. This communication is done through the environment parameter WINDOW_GFX. A subwindow object contains its (process specific) window's device identifier, called a window *file descriptor. Win_fdtoname* translates the file descriptor of the empty subwindow into its name, which is meaningful across process boundaries. *We_setgfxwindow* sets the environment parameter WINDOW_GFX to be *name.* This window name is not be confused with the name passed into *tool_create, ttysw_createtoolsubwindow* or *esw_createtoolsubwindow.*

One of the things that the program wants to do is notice when its child process terminates. An asynchronous software interrupt called SIGCHLD is generated when a child process terminates. *Signal* causes *sigchldcatcher* to be called when a SIGCHLD is received. What *sigchldcatcher* does when it is called is described in *Notification Loop.*

Ttysw_fork forks the program named by the second string in *argv. Ttysw->ts_data* is data specific to the terminal emulator subwindow package. The reason for the other parameters is

outside the context of this discussion. It is sufficient to say that the variables pointed to by these parameters are modified in *ttysw_fork*.

The fork can fail; this is indicated by a return value of –1. An error message is written to standard error by calling *perror*. The appropriate action is to terminate the program.

4.9. Notification Loop

Unlike other programs that you may have written, in a tool program you don't need to poll all the various devices on which you expect input or output activity. Instead, a centralized *notification manager* calls out to interested parties when it detects certain events:

```
tool_select(tool, 1 /* means wait for child process to die*/);

static sigchldcatcher() { tool_sigchld(tool); }

static sigwinchcatcher() { tool_sigwinch(tool); }
```

Tool_select is called to serve as a notification manager for the tool object and its subwindow objects.

- Input, output, or a timeout detected for a object causes a procedure associated with the object to be called. The procedure is expected to respond to this notification appropriately, for example by reading available input, processing it and returning.
- A SIGWINCH interrupt generates a call to *sigwinchcatcher*. This, in turn, calls *tool_sigwinch* with the tool handle as an argument. A procedure associated with each subwindow object responds to this notification appropriately. For example, it rescales its image (if the size of its window has changed), repaints and returns.

The procedures that are notified of interesting events were established for the object instances returned from the tool object creation and subwindow object creation calls described above; *tool_create, ttysw_createtoolsubwindow* and *esw_createtoolsubwindow*.

As mentioned above, this program also needs to wait for the termination of its child process. *Sigchldcatcher* is called due to a SIGCHLD interrupt. *Sigchldcatcher* then calls *tool_sigchld*. This causes *tool_select* to return after cleaning up the terminated child process (see the *wait*(2) system call for more information about cleaning up after a child process). The second argument to *tool_select*, in this case 1, specifies how many child processes it should clean up before returning.

4.10. Cleanup

The call to *tool_destroy* is made so that the tool manager and its subwindow managers may gracefully terminate any ties to the outside world, such as open files, subprocesses, etc.

```
tool_destroy(tool);
exit(0);
```

Just as there are procedures associated with a subwindow manager that are called in the notification loop, another procedure is called when a subwindow instance is being destroyed. The call to *exit* terminates the program and the argument 0 indicates that the program terminated normally.

4.11. Review

After reading this section you should have a grasp of the following concepts:

- A *window* is a UNIX device upon which the *tool package* and *subwindow packages* are built.
- A *tool object* is a framework that is designed to help you manage multiple subwindow objects within a single process.
- A *subwindow manager* is designed to provide a specific set of user interface functions.
- Much of tool writing is setting up subwindow structure.
- The flow of control is based on notification and exceptions from the tool manager, not on polling.

From here you may choose to go to another example provided in *Writing a More Sophisticated Tool* to expand what you have learned. That example includes the following:

- Using other predefined subwindow packages that the top level program interacts with more directly than the ones used in this simple example.
- Creating a subwindow not using any predefined subwindow package.

Chapter 5

Writing a Simple Canvas Program

This chapter describes a simple *canvas program*, called *canvasflash*, that runs in SunWindows. A canvas program is one that owns a single window. In SunWindows, such a program is often written using the *graphics subwindow* package. This example shows how to use the graphics subwindow package to get a single window on which the application draws. The graphics subwindow package shields the canvas application from some of the complexities of window ownership. The graphics subwindow package is by no means the only vehicle for writing canvas programs. It is simply a mechanism to expedite canvas applications.

Canvasflash creates a window and draws a vertical line, a white square and a string of text within the window. Every second it inverts the images within the window. Inverting the line causes it to appear white and essentially "disappear"; the text appears in reverse video; and the white square appears black.

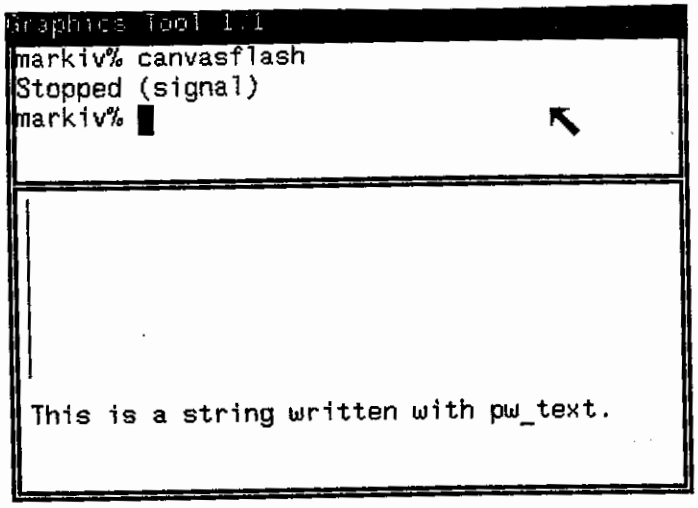


Figure 5-1: *canvasflash* Output

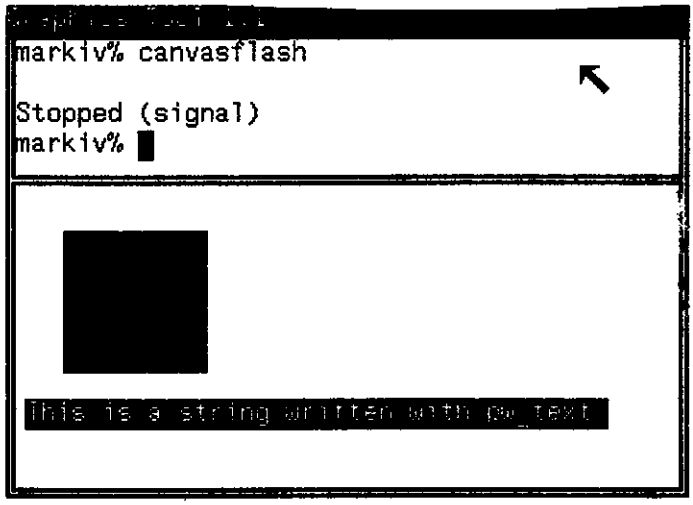


Figure 5-2: Inverted *canvasflash* Output

With no arguments the program runs indefinitely. To run the program for some number of iterations, invoke it with the -n *number* argument where *number* is the number of iterations:

markiv% **canvasflash** –n *number*

The source to this example is provided in the */usr/suntool/src/canvasflash.c* file. To compile the source, use:

markiv% **cc –o canvasflash canvasflash.c –lsuntool –lsunwindow –lpixrect**

5.1. The *canvasflash* Code

The flow of *canvasflash* is as follows:

- *Canvasflash* creates a graphics subwindow object and clears the window associated with it.
- It loops for a specified number of iterations. Within the loop it checks some flags used for window housekeeping and reacts appropriately (details later).
- Once everything is in a known state, *canvasflash* displays the line, square and string of text. The operator that displays the graphics objects is inverted every iteration.

Here is a listing of */usr/suntool/src/canvasflash.c*. You may want to glance at it now, however, it is primarily meant to be referred to as you read the subsequent explanation. Extensive C comments are removed in favor of the accompanying text.

```
#ifndef lint
static  char sccsid[] = "@(#)canvasflash.c 1.1 84/04/04 SMI";
#endif

/* External Declarations */
#include <stdio.h>
#include <suntool/gfx_hs.h>

extern    struct pixfont *pw_pfsysopen();

static    struct pixfont *font;
static    struct gfxsubwindow *gfx;
```

```
main(argc, argv)
    int  argc;
    char **argv;
{
    int  op;

    /* initialization */
    if ((gfx = gfxsw_init(0, argv)) == NULL) {
        fprintf(stderr, "Unable to open graphics subwindow.\n");
        exit(1);
    }

    if ((font = pw_pfsysopen()) == NULL) {
            fprintf(stderr, "Unable to open default font.\n");
            exit(1);
    }
    pw_writebackground(gfx->gfx_pixwin, 0, 0,
       gfx->gfx_rect.r_width, gfx->gfx_rect.r_height, PIX_CLR);

    /* display loop */
    while (gfx->gfx_reps--) {

        /* check to see if window has changed size or been exposed */
        if (gfx->gfx_flags & GFX_DAMAGED)
            gfxsw_handlesigwinch(gfx);

        /* screen has been corrupted and must be redrawn */
        if (gfx->gfx_flags & GFX_RESTART) {
            gfx->gfx_flags &= ~GFX_RESTART;
            pw_writebackground(gfx->gfx_pixwin, 0, 0,
               gfx->gfx_rect.r_width, gfx->gfx_rect.r_height, PIX_CLR);
        }

        /* change raster operation between each iteration */
        op = (gfx->gfx_reps % 2) ? PIX_SRC : PIX_NOT(PIX_SRC);

        /* sample pw_* calls */
        pw_vector(gfx->gfx_pixwin, 5, 5, 5, 100, op, 1);
        pw_writebackground(gfx->gfx_pixwin, 25, 25, 75, 75, op);
        pw_text(gfx->gfx_pixwin, 5, 125, op,
           font, "This is a string written with pw_text.");
        sleep(1);
    }

    /* clean up */
    pw_pfsysclose();
    gfxsw_done(gfx);
}
```

5.2. External Declarations

This section describes the explicit external declarations that must be included to compile this program. The first include statement allows the program to access standard error (*stderr*) for diagnostic output:

```
#include <stdio.h>
```

It also allows the program to use the buffered output *printf* family.

The graphics subwindow package that the program uses is part of the *suntools* library with include files in */usr/include/suntool*. In the include statement:

```
#include <suntool/gfx_hs.h>
```

the *_hs* construct refers to *all* header files needed to run a canvas application that is based on the "gfx" (graphics) subwindow.

5.3. Initialization

This section describes the set-up undertaken before entering the looping part of the program. The following code creates a graphics subwindow object:

```
if ((gfx = gfxsw_init(0, argv)) == NULL) {
        fprintf(stderr, "Unable to open graphics subwindow.\n");
        exit(1);
}
```

The call to *gfxsw_init*() parses *argv* according to the description of arguments to the demo programs in *suntools*(1), and returns a handle to a graphics subwindow object. The handle is a pointer to a **struct** *gfxsubwindow* which contains fields that the canvas application uses:

gfx_pixwin
: is a **struct** *pixwin* pointer. A *pixwin* provides access to a window's visible surface. A program displays in the window by operating through the pixwin.

gfx_rect
: is a **struct** *rect* that describes the current size of the window. This value is updated by the graphics subwindow manager.

gfx_flags
: is an **int** of window housekeeping flags. These flags, as well as *gfx_rect*, are updated asynchronously by the graphics subwindow manager when something happens to affect the window's size or visibility.

gfx_reps
: is a count of the number of repetitions that a cyclic canvas program may use to count down with. It is initialized in *gfxsw_init* to a very large number. If a "-n #" argument sequence is found in *argv* then # is used as the number of repetitions.

If the returned pointer is NULL, an error condition exists. The message "Unable to open graphics subwindow." is displayed, and the program exits.

In a similar fashion, we open a font file that is used while writing text to the screen, and display an error message if there is a problem:

```
if ((font = pw_pfsysopen()) == NULL) {
        fprintf(stderr, "Unable to open default font.\n");
        exit(2);
}
```

We then call *pw_writebackground*() to clear the window:

```
pw_writebackground(gfx->gfx_pixwin, 0, 0,
  gfx->gfx_rect.r_width, gfx->gfx_rect.r_height, PIX_CLR);
```

It writes zeros, defined as the background color, on its destination, *gfx->gfx_pixwin.*

5.4. Display Loop

This part of the program loops until the user interrupts the program or until the repetition counter (*gfx->gfx_reps*) goes to zero:

```
while (gfx->gfx_reps--) {
```

The program is responsible for decrementing this counter to keep track of the number of iterations left to be done.

A graphics subwindow object contains a set of housekeeping flags that the program interrogates.

```
if (gfx->gfx_flags & GFX_DAMAGED)
    gfxsw_handlesigwinch(gfx);
```

The status flag GFX_DAMAGED indicates when part of the window has become exposed or when the window has changed size. Removing an overlapping window or changing the window's size may expose a portion of the window, whose image must then be redrawn. The description of a previously hidden area of the window is known as *damage* because the application may need to redraw part of its image.

The standard thing to do when the GFX_DAMAGED flag is set is to call *gfxsw_handlesigwinch.* Unless the window's size has changed, this routine can repair the damage if the graphics subwindow's pixwin has been made retained.

The graphics subwindow manager sets the GFX_RESTART flag on window size changes or when there is some part of the window for the client to refresh.

```
if (gfx->gfx_flags & GFX_RESTART) {
    gfx->gfx_flags &= ~GFX_RESTART;
    pw_writebackground(gfx->gfx_pixwin, 0, 0,
        gfx->gfx_rect.r_width, gfx->gfx_rect.r_height, PIX_CLR);
}
```

Many canvas applications will scale their contents to current dimensions. The minimum that needs to be done is to clear the flag and repaint the window. Here we just clear the window.

To flash the demo, we alternate PIX_SRC and PIX_NOT(PIX_SRC) as display operations:

```
op = (gfx->gfx_reps % 2) ? PIX_SRC : PIX_NOT(PIX_SRC);
```

Each pixwin operation is given a source image, a destination image, and an operator. The operator determines the relationship between the source image and the destination image. For now, it is important to note only that PIX_SRC maps its source directly onto its destination. PIX_NOT(PIX_SRC) inverts its source before mapping it onto the destination.

The following function calls are the meat of the program, as they draw the graphics in the window:

```
pw_vector(gfx->gfx_pixwin, 5, 5, 5, 100, op, 1);
pw_writebackground(gfx->gfx_pixwin, 25, 25, 75, 75, op);
pw_text(gfx->gfx_pixwin, 5, 125, op,
    font, "This is a string written with pw_text.");
```

Pw_vector()
draws a vector onto a destination pixwin. It accepts as arguments the destination, endpoints for the vector, a raster operation to apply to the source before writing, and a source value to write (color).

Pw_writebackground()
is used to draw a solid square on the display.

Pw_text()
writes text in a specified font. It accepts as arguments a destination pixwin, a location within the destination at which to begin writing (the y component is the baseline, not the upper edge of the text), a raster operation that specifies how to combine the text with the destination, a handle to the desired font — typically a pointer to a *pixfont* structure acquired by opening the font, and a string of text to be printed.

The *sleep* procedure waits for 1 second before returning. This slows the action so that you can clearly see the flashing.

```
sleep(1);
```

5.5. Cleanup

To cleanup before exiting the program, we have:

```
pw_pfsysclose();
gfxsw_done(gfx);
```

- *Pw_pfsysclose*() frees the resources used for the program's text font.
- *gfxsw_done*() frees the resources allocated for the graphics subwindow.

5.6. Review

After reading this section, you should know:

- that canvas programs can be written using the graphics subwindow package,
- that canvas programs need not be written using the graphics subwindow package,
- how to write a non-interactive canvas program, and
- that pixwins are used to write on the display.

Chapter 6

Writing a More Sophisticated Tool

Here we examine a tool which uses SunWindows facilities to perform a more sophisticated operation than the *gfxtool* covered in *Writing a Simple Tool*. This tool, called the *mouse tool*, evaluates different mice for speed and accuracy; it can also be used as a drill to improve a user's skill with the mouse.

The mouse tool presents a kind of shooting gallery to the user: it displays a sequence of randomly positioned targets to be selected with the mouse. After any click of the left mouse button in the window, the current target is removed from the window and the next one presented. If the cursor was within the target, the time that target was kept up is computed; otherwise, 1 is added to a count of errors. A continuously-updated report of the user's speed and accuracy is displayed in a message subwindow.

The following figure presents a view of the mouse tool in operation.

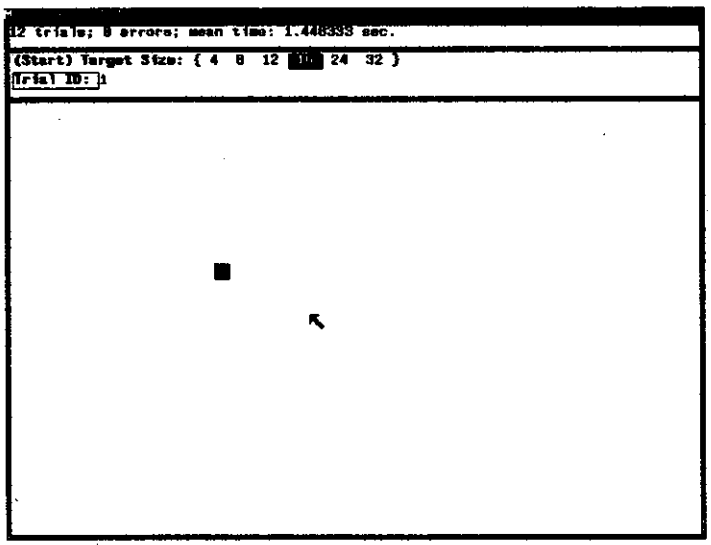


Figure 6-1: *mousetool* Output

The mouse tool incorporates three subwindows, two standard and one special:

option subwindow
: is the vehicle through which the user controls the parameters of the tool's operation, such as the size of the target and the particular sequence of random positions.

message subwindow
displays the current report.

range subwindow
is where targets are set up as on a firing range — this is a custom subwindow.

6.1. Overview of the Mouse Tool Processing

Processing in the mouse tool occurs in four phases:

Tool initialization
When the tool starts up, it creates its various windows, adds the desired items to the option subwindow, and provides handlers for the events of interest (size change signals and input events). Then the tool's window is installed, and the standard tool control structure is invoked.

Sequence initialization
It is possible to run multiple sequences of trials with one invocation of the tool. At the start of each sequence, the tool sets the size of the target to be used, resets the counters that record performance, determines the particular sequence of target locations, and presents an initial target for the user to select.

Trial Processing
When a sequence of trials is begun, the tool notes the time when a target is put up, and waits for a button-down on the mouse's left button inside the subwindow which holds the target. When the button-down is received, the time is noted, and the mouse coordinates are checked to determine whether the user hit the target. If so, the mean time to hit is updated, else the number of errors is. In either case, the new result is reported in the message subwindow, and another trial is started.

The user may start a new sequence of trials at any time by selecting the **(Start)** command, or by changing the target size, in the option subwindow.

Termination
The tool will repeat trials and sequences of trials endlessly, until stopped. The most convenient way of terminating the tool is to bring up the Tool Manager menu (available in any of the tool's windows, since none of them provides a menu of its own), and select the **Quit** item. This causes the tool to exit its main loop. Then a call to *tool_destroy* releases resources allocated by the tool and its subwindows, and the tool's process terminates normally.

6.2. The *mousetool* Code

The following pages present the code of *mousetool.c*. The in-depth discussion that follows covers this code pretty much in order. Extensive C comments are removed in favor of the accompanying text.

```
#ifndef lint
static  char sccsid[] = "@(#)mousetool.c 1.1 84/04/04 SMI";
#endif
```

```
#include <stdio.h>
#include <suntool/tool_hs.h>
#include <suntool/optionsw.h>
#include <suntool/msgsw.h>

extern long              random();
extern char             *initstate();
extern struct pixfont    *pw_pfsysopen();


/* tool and sunwindows-specific data */
static struct tool              *tool;
static struct toolsw            *msg_tsw, *opt_tsw, *range_tsw;
static struct pixwin            *range_pixwin;
static struct msgsubwindow      *msw;
static caddr_t                   osw, start_item, size_item, srand_item;

static struct typed_pair     start_label = { IM_TEXT, "Start" };
static struct typed_pair     size_label  = { IM_TEXT, "Target Size" };
static char                 *size_tags[] = { "4", "8", "12", "16", "24", "32", 0};
static struct typed_pair     size_choices = { IM_TEXTVEC, (caddr_t)size_tags };
static int                   size_values[]= { 4, 8, 12, 16, 24, 32 };
static struct typed_pair     srand_label = { IM_TEXT, "Trial ID" };

static u_int  running = FALSE,
              cur_x, cur_y, size = 16,
              cum_time, errors, trials_done,
              win_width, win_height, width_limit, height_limit;

static char   random_state[256];

static struct timeval   start_time, stop_time;
static struct timezone  trash_zone;

static struct pixfont  *font;

static    start_proc(), sigwinched(), range_selected(), range_sighandler();
```

```
main()
{

/* create the tool */
    tool = tool_create("Mouse Tool 1.1", TOOL_NAMESTRIPE | TOOL_BOUNDARYMGR,
            (struct rect *)NULL, (struct icon *)NULL);
    if (tool == (struct tool *)NULL)  lose("Couldn't create tool");
    font = pw_pfsysopen();
    if (font == (struct pixfont *)NULL)  lose("Couldn't get default font");

    /* create and initialize the message subwindow */
    msg_tsw = msgsw_createtoolsubwindow(tool, "", TOOL_SWEXTENDTOEDGE,
            (font->pf_defaultsize.y * 3) / 2, "Click (Start) to begin", font);
    if (msg_tsw == (struct toolsw *)NULL)  lose("Couldn't create message subwindow");
    msw = (struct msgsubwindow *)msg_tsw->ts_data;

    /* create and initialize the option subwindow */
    opt_tsw = optsw_createtoolsubwindow(tool, "", TOOL_SWEXTENDTOEDGE,
            font->pf_defaultsize.y * 3);
    if (opt_tsw == (struct toolsw *)NULL)  lose("Couldn't create option subwindow");
    init_options();

    /* create and initialize the range subwindow */
    range_tsw = tool_createsubwindow(tool, "",
            TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE);
    if (range_tsw == (struct toolsw *)NULL)  lose("Couldn't create range subwindow");
    init_range();
    initstate(1, random_state, 256);
    signal(SIGWINCH, sigwinched);

    /* install tool */
    tool_install(tool);
    tool_select(tool, 0);

    /* terminate tool */
    tool_destroy(tool);
    pw_pfsysclose();
    exit(0);
}

lose(str)
char *str;
{   fprintf(stderr, str); exit(1);
}

static
sigwinched()
{   tool_sigwinch(tool);
}
```

```
static
init_options()
{
    struct item_place place;

    osw = opt_tsw->ts_data;
    start_item = optsw_command(osw, &start_label, start_proc);
    if (start_item == NULL) lose("Couldn't create start item");
    size_item = optsw_enum(osw, &size_label, &size_choices, 0, 3, start_proc);
    if (size_item == NULL) lose("Couldn't create size item");
    srand_item = optsw_text(osw, &srand_label, "1", 0, (int (*)()) NULL);
    if (srand_item == NULL) lose("Couldn't create random initializer item");
    optsw_getplace(osw, srand_item, &place);
    place.rect.r_left = optsw_coltox(osw, 0); place.fixed.x = TRUE;
    place.rect.r_width = optsw_coltox(osw, 60); place.fixed.w = TRUE;
    optsw_setplace(osw, srand_item, &place, FALSE);
}

static
init_range()
{
    struct inputmask  mask;

    range_pixwin = pw_open(range_tsw->ts_windowfd);
    if (range_pixwin == (struct pixwin *)NULL) lose("Couldn't create range pixwin");
    range_tsw->ts_io.tio_handlesigwinch = range_sighandler;
    range_tsw->ts_io.tio_selected = range_selected;
    input_imnull(&mask);
    win_setinputcodebit(&mask, MS_LEFT );
    win_setinputmask(range_tsw->ts_windowfd, &mask, (struct inputmask *)NULL, WIN_NULLLINK);
}


/* respond to damage to range subwindow */
static
range_sighandler(sw)
caddr_t sw;
{
    struct rect    r;
    int            val;

    win_getsize(range_tsw->ts_windowfd, &r);
    win_width = r.r_width; win_height = r.r_height;
    width_limit = r.r_width - size; height_limit = r.r_height - size;
    pw_damaged(range_pixwin);
    pw_writebackground(range_pixwin, 0, 0, r.r_width, r.r_height, PIX_CLR);
    pw_donedamaged(range_pixwin);
    if (running) start_proc(osw, start_item);
}
```

```
/* get notification from option subwindow */
#define SEED_BUF_SIZE 256
static
start_proc(sw, ip, new_value)
caddr_t  sw;
caddr_t  ip;
int      new_value;
{
    int                 seed, val;
    char                buf_data[SEED_BUF_SIZE];
    struct string_buf   buf;

    if (running) {
        running = FALSE;
        msgsw_setstring(msw, "Restarting");
        sleep(2);
    }
    pw_writebackground(range_pixwin, 0, 0, win_width, win_height, PIX_CLR);
    size = size_values[optsw_getvalue(size_item, (caddr_t)&val)];
    width_limit = win_width - size; height_limit = win_height - size;
    buf.limit = SEED_BUF_SIZE; buf.data = buf_data;
    optsw_getvalue(srand_item, (caddr_t)&buf);
    seed = atoi(buf_data);
    initstate(seed, random_state, 256);
    set_target();
}

/* respond to user inputs in target range */
static
range_selected(sw, ibits, obits, ebits, timer)
caddr_t          sw;
int              *ibits, *obits, *ebits;
struct timeval   **timer;
{
    int                 x, y;
    struct inputevent ie;

    if (input_readevent(range_tsw->ts_windowfd, &ie) == -1) {
        perror("input_readevent failed");
        abort();
    }
```

```
    *ibits = *obits = *ebits = 0;
    x = ie.ie_locx; y = ie.ie_locy; stop_time = ie.ie_time;
    if (!running) {
        cum_time = trials_done = errors = 0;
        running= TRUE;
    } else {
        trials_done += 1;
        if (x < cur_x || y < cur_y ||
           x >= cur_x + size || y >= cur_y + size) {
            errors += 1;
        } else {
            if (stop_time.tv_usec < start_time.tv_usec) {
                stop_time.tv_usec += 1000000;
                stop_time.tv_sec -= 1;
            }
            cum_time += stop_time.tv_usec - start_time.tv_usec +
                        (stop_time.tv_sec - start_time.tv_sec) * 1000000;
        }
    }
    pw_writebackground(range_pixwin, cur_x, cur_y, size, size, PIX_CLR);
    report();
    set_target();
}

static
set_target()
{
    cur_x = random() % width_limit;
    cur_y  = random() % height_limit;
    pw_writebackground(range_pixwin, cur_x, cur_y, size, size, PIX_SET);
    gettimeofday(&start_time, &trash_zone);
}

static
report()
{
    int       good_trials = trials_done - errors;
    double    mean_time;
    char      buf[256];

    if (good_trials == 0) {
        mean_time = 0;
    } else {
        mean_time =  ((float)cum_time / (float)good_trials ) /  1000000;
    }
    sprintf(buf,
        "%d trials; %d errors; mean time: %f sec.",
        trials_done, errors, mean_time);
    msgsw_setstring(msw, buf);
}
```

6.3. External Declarations

The first lines of the program are concerned with procedures and data declared elsewhere. Four header files are included:

```
#include <stdio.h>
#include <suntool/tool_hs.h>
#include <suntool/optionsw.h>
#include <suntool/msgsw.h>
```

<stdio.h>
provides the declarations for things in the standard I/O library, *fprintf*, for instance, and declares NULL.

<suntool/tool_hs.h>
includes the collection of header files needed for dealing with tools and subwindows. Its function is the same as in the *gfxtool* described in *Writing a Simple Tool.*

<suntool/optionsw.h>
has the declarations of the procedures, structures, and defined constants needed for an option subwindow.

<suntool/msgsw.h>
has the declarations of the procedures, structures, and defined constants needed for a message subwindow.

There are also external declarations for two C library procedures, which implement a random number generator; these declarations do not appear in any header file.

```
extern long         random();
extern char         *initstate();
extern struct pixfont          *pw_pfsysopen();
```

6.4. Global Data Definitions

The next lines declare global data for the program. These items have two possible functions: they are referenced from several procedures or they maintain a state; a few items which could be local to a procedure appear here simply to keep them associated with related items which *are* global. There are also declarations of four procedures which will be defined later; C requires such declarations to allow the addresses of these procedures to be used before the procedures are defined.

6.4.1. Tool- and Sunwindows-Specific Data

The first lines of global data are directly concerned with the mouse tool's interface to the window system:

```
static struct tool          *tool;
static struct toolsw        *msg_tsw, *opt_tsw, *range_tsw;
static struct pixwin        *range_pixwin;
static struct msgsubwindow  *msw;
static caddr_t               osw, start_item, size_item, srand_item;
```

The tool handle (pointer to a *struct tool*) returned by *tool_create* will be stored in *tool.* Handles for the three subwindows (pointers to *struct toolsw*'s) will be stored in *msg_tsw, opt_tsw,* and *range_tsw* when they are returned from their respective creation routines. Because the mouse tool uses the pixwin display interface directly to display in the range subwindow, there is a pointer here for the range subwindow pixwin structure. *Msw* will hold a pointer to the private

data for the message subwindow. *Osw* will hold a pointer to the private data for the option subwindow, and the three option items in that subwindow will be identified by handles stored in *start_item, size_item,* and *srand_item.*

The next six lines hold data used to define the three items in the option subwindow:

```
static struct typed_pair    start_label    = { IM_TEXT, "Start" };
static struct typed_pair    size_label     = { IM_TEXT, "Target Size" };
static char                 *size_tags[]   = { "4", "8", "12", "16", "24", "32", 0};
static struct typed_pair    size_choices   = { IM_TEXTVEC, (caddr_t)size_tags };
static int                  size_values[]  = { 4, 8, 12, 16, 24, 32 };
static struct typed_pair    srand_label    = { IM_TEXT, "Trial ID" };
```

All three items have text for their textual labels, as indicated by the type IM_TEXT in the *struct typed_pair.* The enumerated item also has an array of labels for its various choices (type IM_TEXTVEC), and another array for translating the value of the item. (This is needed because the option subwindow implementation deals with the value as an index in the range 0 - 5, where we want the corresponding values 4, 8, 12, etc.)

6.4.2. Other Global Data

The other lines of global data declarations are:

```
static u_int        running = FALSE,
                    cur_x, cur_y, size = 16,
                    cum_time, errors, trials_done,
                    win_width, win_height, width_limit, height_limit;

static char         random_state[256];

static struct timeval   start_time, stop_time;
static struct timezone  trash_zone;

static struct pixfont   *font;
```

Running is a state variable used to indicate whether a sequence of trials has been started.

Cur_x, cur_y, and *size*
define the current target.

Cum_time, errors, and *trials_done*
accumulate statistics for a sequence of trials.

Win_width and *win_height*
are the cached dimension of the range subwindow, and *width_limit* and *height_limit* are those same dimensions less the size of the target. (These limits are used to calculate target positions so that the target is always fully visible.)

Random_state
is a state vector required by the random number library routines.

Start_time and *stop_time*
are used to note the time a target is put up and the time of the subsequent button-push that ends that trial. The system call that provides a time also requires a place to store a time-zone, which we ignore; *trash_zone* provides this area. The message subwindow creation routine requires a font handle with which to display the current

message; this will be stored in *font*.

Finally, the four procedures whose addresses will be used before they are defined are declared:

```
static      start_proc(), sigwinched(), range_selected(), range_sighandler();
```

6.5. Main Procedure

When the mouse tool is started, control is passed to the procedure *main*. Thus, this procedure provides a top-level view of the tool's processing. Nineteen of the 22 lines are concerned with initialization, and the last two provide clean termination: all of the normal processing for the tool is encapsulated within the call to *tool_select*.

6.6. Initializing the Mouse Tool

The general strategy of tool initialization is to create a tool object, and then to create its subwindow objects in the order they appear in the tool window, initializing each in turn. When the whole structure is built up, SunWindows is told that the tool is ready to handle signals from the window system, and the tool's window structure is inserted into the tree of windows for its display. This is the point at which the tool's windows become visible on the screen.

Most of the procedures that will be called in the initialization phase may indicate they were unable to perform their task. The mouse tool's response to all of these is the same: it explains what went wrong as best it can, and gives up. This is handled by the procedure *lose*() described in *Remaining Initialization*.

6.6.1. Creating the Mouse Tool

The first two statements of *main*() create the tool object with standard options, or complain if that's not possible:

```
tool = tool_create("Mouse Tool 1.1", TOOL_NAMESTRIPE | TOOL_BOUNDARYMGR,
            (struct rect *)NULL, (struct icon *)NULL);
if (tool == (struct tool *)NULL)  lose("Couldn't create tool");
```

- "Mouse Tool 1.1" is the string that is displayed in the mouse tool's name stripe, and in its icon when it is closed.
- Two flags indicate respectively that the mouse tool should be displayed with the usual black stripe containing the mouse tool's name at the top (TOOL_NAMESTRIPE), and that the user should be able to adjust the boundaries between subwindows (TOOL_BOUNDARYMGR). Both these facilities are implemented by library procedures which will be invoked as needed; no further client actions are required to provide them.
- The tool's size and position are left to the window manager's discretion by supplying NULL instead of a pointer to a rectangle in the third argument.
- Similarly, the NULL in the fourth argument indicates that the default icon should be used for this tool. This is a square with a thick black border, and the tool's name displayed inside as shown in the following figure.

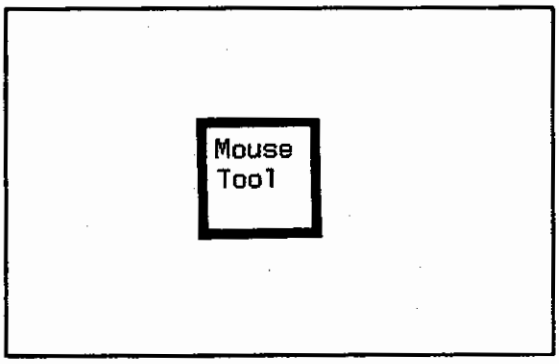


Figure 6-2: Default Tool Icon

6.6.2. Creating and Initializing the Message Subwindow

The next lines of *main* create the first subwindow object for the tool. This is the message subwindow where messages will be displayed to the user.

```
font = pw_pfsysopen();
if (font == (struct pixfont *)NULL)  lose("Couldn't get default font");
msg_tsw = msgsw_createtoolsubwindow(tool, "", TOOL_SWEXTENDTOEDGE,
        (font->pf_defaultsize.y * 3) / 2, "Click (Start) to begin", font);
if (msg_tsw == (struct toolsw *)NULL)  lose("Couldn't create message subwindow");
msw = (struct msgsubwindow *)msg_tsw->ts_data;
```

Since the message subwindow requires its clients to specify which font to use for displaying the messages, the first step is to determine the system default font, and to store its handle in the global variable *font*.

The subwindow object is actually created in a call to *msgsw_createtoolsubwindow*; its arguments indicate details specific to this particular message subwindow:

- *Tool* identifies the tool object this subwindow is to be part of by passing in the handle which *tool_create* provided. The tool object will be modified to include the new subwindow object created in this call.
- Since subwindow names are not used for anything at this point, an empty string is given for the name of this argument.
- TOOL_SWEXTENDTOEDGE indicates that the width of the subwindow is to be elastic, stretching to fill whatever the width of the tool window happens to be.
- The height of the message subwindow is defined to be 1.5 times the height of characters in the current font.
- The string "Click (Start) to begin" specifies the text that will be displayed when the message subwindow first appears.
- *Font* specifies the font that will be used to display messages; its value is the font handle which was provided by the call to *pw_pfsysopen*() in the first line of this section.

In calls to message subwindow procedures, the information needed about the subwindow is encapsulated in the subwindow's private data, the object pointed to by *ts_data* in the *toolsw* struct. Therefore, we save this handle in *msw* for convenience in passing it to those message

subwindow procedures.

6.6.3. Creating and Initializing the Option Subwindow

The next subwindow created is an option subwindow, which provides a control panel for the tool. Because the message subwindow was specified to spread across the tool, this second subwindow will also begin at the left edge, starting immediately below the message subwindow:

```
opt_tsw = optsw_createtoolsubwindow(tool, "", TOOL_SWEXTENDTOEDGE,
        font->pf_defaultsize.y * 3);
if (opt_tsw == (struct toolsw *)NULL)  lose("Couldn't create option subwindow");
init_options();
```

The arguments to *optsw_createtoolsubwindow* exactly parallel those to *msgsw_createtoolsubwindow,* leaving off the text and font.

Because initializing this subwindow object is somewhat more complicated, creating several option items has been isolated in a subroutine:

```
static
init_options()
{
    struct item_place place;
```

Among the structs declared in *optionsw.h* is a struct *item_place,* which describes the layout aspects of an option item (the x and y origins, the width and height, and four boolean flags indicating whether each of those may be adjusted when the window must be rearranged). This struct will be used to ensure that the third item which holds text begins a new line and is wide enough to display its contents.

As with the message subwindow, the usual object of interest to option subwindow procedures is the private data specific to the subwindow's nature as an option subwindow. The handle on this data is stored in the global variable *osw:*

```
osw = opt_tsw->ts_data;
```

Then we proceed to create the option items. The option subwindow package provides a distinct creation routine for each kind of item (boolean, command, enumerated, label, text); each routine returns either a handle on the item created, or NULL if it cannot create an item.

The first item in the window is a command; that is, it invokes some procedure when the user selects it with the mouse.

```
start_item = optsw_command(osw, &start_label, start_proc);
if (start_item == NULL)  lose("Couldn't create start item");
```

A tool may have several option subwindows; *osw* indicates which option subwindow this item is to be added to. It will be positioned in the upper left corner of the window because it is the first item created for this subwindow. The text "Start" labels it, as defined in the static data item *start_label.* When the user selects the button on the screen, we want the procedure *start_proc* to be invoked.

The next item is to be an *enum,* an item whose value is one of a small set of defined values:

```
size_item = optsw_enum(osw, &size_label, &size_choices, 0, 3, start_proc);
if (size_item == NULL)  lose("Couldn't create size item");
```

Its label has likewise been predefined as the text "Target Size." The choices available to the user are listed in the the array *size_choices*, namely 4, 8, 12, 16, or 32. No flags are currently defined for enumerated items, so we explicitly pass 0 here. We indicate which choice we want to be the initial value of the item by giving its index in the choice array as the fifth argument: *size_choices*[3] is "16." Note that the value of an enumerated item is reported and manipulated as an index into an array, *not* the text of the choice the user sees. Finally, whenever the user changes the value of this item, we want to restart the sequence of trials with the new target size, so we specify that *start_proc* is to be invoked in this case as well as when the "Start" command is selected.

The last item in the option subwindow is a seed for the random number generator; this allows the user to replicate a sequence of trials exactly, lending some validity to comparisons. The seed is actually a number, but the user will enter it as text, so we want a text item:

```
srand_item = optsw_text(osw, &srand_label, "1", 0, (int(*)())NULL);
if (srand_item == NULL) lose("Couldn't create random initializer item");
```

The specification of the containing option subwindow and the label of the item are just as in the previous items. The initial value of the item will be the string "1"; we don't want any of the options specified by flags (the only one currently defined masks the value, as for passwords); and we don't need any procedure to be invoked when the value changes.

We do, however, care about the placement of the item. Since text is generally elastic, the option subwindow's layout routines will have a much easier time, and generally produce a more pleasant result, if we constrain the variability of the item:

```
optsw_getplace(osw, srand_item, &place);
place.rect.r_left = optsw_coltox(osw, 0); place.fixed.x = TRUE;
place.rect.r_width = optsw_coltox(osw, 60); place.fixed.w = TRUE;
optsw_setplace(osw, srand_item, &place, FALSE);
```

Optsw_getplace stores a description of the item's current size, position, and flexibility into the struct *place*. We then modify that description to ensure that the item starts at the left edge of the window, and the combined width of the label and value is sufficient to hold 60 characters. We are using the *optsw_coltox* utility routine provided by the option subwindow package to translate character coordinates to pixels. That description is then handed back to the option subwindow package, which will now refrain from modifying the fixed aspects of this item. The fourth argument to *optsw_setplace* indicates whether the subwindow should be laid out again, taking the new information for this item into account. Since the window hasn't appeared yet, there's no point to rearranging it, hence our "FALSE."

This completes initialization of the option subwindow.

6.6.4. Creating and Initializing the Range Subwindow

The last subwindow created is the range in which targets will be displayed. There is no established subwindow type which provides this capability for us, so we will build it ourselves on top of lower-level facilities.

The first step looks very familiar:

```
range_tsw = tool_createsubwindow(tool, "",
        TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE);
if (range_tsw == (struct toolsw *)NULL) lose("Couldn't create range subwindow");
```

Tool_createsubwindow is a procedure which creates a tool subwindow of *no* defined type. Its

only interesting properties are that it will expand across and down to fill the remainder of the tool window's area.

The steps required to give the range subwindow some interesting behavior are again isolated in a separate procedure to keep from cluttering *main*:

```
init_range();
```

We look at that procedure here:

```
static init_range()
{
        struct inputmask  mask;
```

Since there is no established subwindow type implementing the range subwindow for us, we will be doing our own input and output. The input mask controls which user input events are actually read from this window.

In order to draw on the window safely, we want to use the pixwin facilities detailed in *Overlapped Windows: Imaging Facilities* in the reference manual. Our first step is to get a pixwin for the window:

```
range_pixwin = pw_open(range_tsw->ts_windowfd);
if (range_pixwin == (struct pixwin *)NULL)  lose("Couldn't create range pixwin");
```

This pixwin handle will be used in the routines which draw on this window.

The next step is to set up the tool subwindow object so it will be notified of interesting events. In our case, there are two such events:

- a signal (SIGWINCH) indicating that the window has changed and may need repair, and
- arrival of a user input event to be read and processed.

For each such event, the address of a procedure which can handle it must be stored in an appropriate procedure pointer:

```
range_tsw->ts_io.tio_handlesigwinch = range_sighandler;
range_tsw->ts_io.tio_selected = range_selected;
```

This declares that *range_sighandler* is the procedure to call when a SIGWINCH indicates damage to the range subwindow, and *range_selected* is the procedure to call when the user generates input of interest.

The last step in initializing the range subwindow is to specify which input events are of interest. As it turns out, all we care about is a button-down on the left mouse button. So we start by making our mask accept no events at all, and then turn on the bit that indicates interest in the left mouse button. That is the mask we tell the system to use:

```
input_imnull(&mask);
win_setinputcodebit(&mask, MS_LEFT );
win_setinputmask(range_tsw->ts_windowfd, &mask, NULL, WIN_NULLLINK);
```

The NULL third argument to *win_setinputmask* indicates that we have no desire to dispose of any events already queued for this window. (There should be none, since the window hasn't appeared on the screen yet.) The WIN_NULLLINK in the fourth argument indicates that the next window to be offered an input event that the range subwindow doesn't want should be the default, namely its parent (the tool window).

This concludes the initialization of the range subwindow, the last of the tool's windows.

6.6.5. Remaining Initialization and Utilities

The remaining initialization is very simple:

```
initstate(1, random_state, 256);
signal(SIGWINCH, sigwinched);
tool_install(tool);
```

The random number generator is initialized to start with its default sequence of positions. The operating system is informed via the *signal*(3) system call that this process is prepared to receive SIGWINCH signals, and that the procedure to invoke when they occur is *sigwinched*. Until now, the various windows have existed in limbo as they were being set up; *tool_install* is responsible for putting them on the screen.

This completes the tool initialization discussion. Before we turn to normal tool processing, however, let us examine two brief procedures which are defined immediately after *main*. The first is the procedure which has been used throughout initialization to note an unrecoverable failure:

```
lose(str)
char    *str;
{       fprintf(stderr, str); exit(1);
}
```

Lose takes a string as its argument, prints it on the standard error stream, and exits the program.

The second procedure is even shorter, though somewhat more central to our program. It is invoked when the process receives a SIGWINCH signal:

```
static
sigwinched()
{       tool_sigwinch(tool);
}
```

This procedure runs asynchronously; that is, almost any other statement in the program may be in the midst of executing when this is called. Therefore, we want to be very brief about the processing done here, and to be careful neither to rely on the state of data in other parts of the program, nor to invalidate assumptions those other parts may make about their data.

In light of these considerations, the mouse tool's response to a SIGWINCH is simply to set a flag noting that the interrupt has occurred, and to return. This is done by the call to *tool_sigwinch*; the *tool* argument allows that procedure to know where to set the flag. All other processing in response to the signal is delayed until the program's normal processing gets around to noticing this flag and invoking appropriate handlers synchronously. This is described under *tool_select*() in the next section.

This concludes the discussion of tool initialization for the mouse tool.

6.7. Normal Tool Processing

Once the tool has been set up and installed, it begins its normal processing. This is done entirely in response to outside events: user inputs and window manipulations. Therefore, this section will consist of descriptions of subroutines and the circumstances under which they are called.

To get into this state, however, it is necessary to start listening for these events. This is done in one line of *main*, which encapsulates the main loop of the program:

```
tool_select(tool, 0);
```

The *tool* argument gives *tool_select* a handle on all of the status data it needs; the 0 means that this tool doesn't expect to fork any subordinate process that would need to be collected when it finishes.

Tool_select repeatedly does a *select*(2) system call, which blocks until input is available on some device, a timer runs down, or a signal is received. When it is awakened by one of these, *tool_select* finds the corresponding procedure to call in the various tool and tool subwindow objects, and invokes it. After that procedure returns, it goes back to the top of its loop to await the next thing to do.

Potentially, that's a lot of things to take care of. But the mouse tool keeps things very simple:

- It never sets a timer.
- Its tool window, message subwindow, and option subwindow have signal handlers of their own that respond to SIGWINCHes without bothering the client.
- The message subwindow does not accept any input, and the tool window and option subwindow have input handlers provided by their implementors.

Inputs and signals for the range subwindow must be handled. There are two cases when the option subwindow will call out to mouse tool code: when the user selects the "Start" button, and when the user changes the "Target size" item.

Thus, there are three circumstances which must be provided for in mouse tool code:

- the range subwindow receives a SIGWINCH, indicating it must repair damage;
- the option subwindow notifies the tool that one of its two events of interest has occurred; or
- the range subwindow has a user input to process, generated by the user pressing the left mouse button while the cursor is in the range window.

Each of these cases has a procedure to respond to it; we will consider them in order.

6.7.1. Responding to Damage to the Range Subwindow

When the range window is damaged a SIGWINCH is sent to the process which owns the window, that is, the mouse tool. This SIGWINCH is caught by the procedure *sigwinched*, as described above, which simply sets a flag in the tool object and returns. The occurrence of the signal is enough to awaken *tool_select* from its *select* if it is in one; otherwise, the flag will be noted the next time *tool_select* comes around to the top of its loop, and *tool_select* will avoid going to sleep.

In either case, *tool_select* will determine that the range window has been damaged, find the address of *range_sighandler* in its tool subwindow object, and invoke it:

```
static range_sighandler(sw)
caddr_t sw;
{
        struct rect r;
        int         val;

        win_getsize(range_tsw->ts_windowfd, &r);
        win_width = r.r_width; win_height = r.r_height;
        width_limit = r.r_width - size; height_limit = r.r_height - size;
```

The first thing *range_sighandler* does is update its idea of the size of its window. *Win_getsize* stores the window's rectangle into the struct addressed by its second argument. The interesting parts of this rectangle, namely the width and height, are copied into global data (a window's coordinate system always has a 0,0 origin). The limits on the origin for displaying the target must also be updated, since a formerly valid position may now lie outside the window.

Assuming the window has been damaged, we now do a very simplistic repair job:

```
        pw_damaged(range_pixwin);
        pw_writebackground(range_pixwin, 0, 0, r.r_width, r.r_height, PIX_CLR);
        pw_donedamaged(range_pixwin);
```

There may not have been any actual damage — for instance, the window may have shrunk — or it may be that only a little of the window needs to be repaired. By bracketing the code with *pw_damaged* and *pw_donedamaged*, we ensure that we write only to the areas of the screen that need it.

Since changing the size or visible portion of the window seriously affects the sequence of target positions, the mouse tool takes a SIGWINCH as an indication that any current sequence of trials should be terminated and a new one started. To do this, it simply duplicates the call that the option subwindow makes when the user selects the "Start" button:

```
        if (running)  start_proc(osw, start_item);
}
```

This completes the processing of a SIGWINCH on the range subwindow.

6.7.2. Notification from the Option Subwindow

The option subwindow implementation takes care of a lot of details for its clients, including signal handling and feedback as the mouse is moved in the window. But eventually, the user will perform some action which requires the subwindow manager to notify its client. In the mouse tool, there are two such events: the user may select the "Start" button, or change the "Target size." In both cases, because of the way those items were initialized, the subwindow implementation will call *start_proc*:

```
#define SEED_BUF_SIZE   256
static
start_proc(sw, ip, new_value)
caddr_t     sw;
caddr_t     ip;
int         new_value;
{
        int                     seed, val;
        char                    buf_data[SEED_BUF_SIZE];
        struct string_buf       buf;
```

Several small bugs have been left in this code, in the interests of realism. Here, the declarations of the arguments to *start_proc* are a little awkward, due partly to our short-cut of using the same procedure for both events. The option subwindow package will call its client's *notify* procedures with arguments that identify the subwindow and the item an event is associated with; if the item has a numeric value (a boolean or enumerated item), a third argument gives the new value. In the current case, the declarations are written for the fuller case, when the target-size item has changed; *new_value* will be undefined when this procedure is called to respond to a button-click on the "Start" command.

As it happens, none of these arguments serves any purpose in this code anyway, except perhaps to reduce the number of complaints from *lint*(1).

A sequence of trials may already have started when *start_proc* is invoked, so the first step is to terminate any old sequence:

```
if (running) {
        running = FALSE;
        msgsw_setstring(msw, "Restarting");
        sleep(2);
}
```

(The *sleep* allows the user time to read the message before anything else starts happening.)

Next we guarantee that the range is clear; no old targets are cluttering up the image, for instance:

```
pw_writebackground(range_pixwin, 0, 0, win_width, win_height, PIX_CLR);
```

Then we make sure we're up to date on the parameters:

```
size = size_values[optsw_getvalue(size_item, (caddr_t)&val)];
width_limit = win_width - size; height_limit = win_height - size;
```

Optsw_getvalue takes a handle on the item of interest and a pointer to a place to store the result; for items with numeric values, it returns the same value it stores. Because the value is defined to be an index, rather than the value the user sees, we use it to index into the array of values to translate to the actual sizes. The limits on target positions are always calculated, in case the target size has changed or this is the first call to *start_proc*.

(Another bug: if the window's dimensions are smaller than the target size, the limits can go negative here. This will result in no target being visible.)

The other parameter to be updated is the seed for the random number generator. This can be changed at any time by the user, without affecting the rest of the program — we don't want to be restarting a run of trials on every keystroke. So the value must be retrieved explicitly before each sequence is begun:

```
        buf.limit = SEED_BUF_SIZE; buf.data = buf_data;
        optsw_getvalue(srand_item, (caddr_t)&buf);
        seed = atoi(buf_data);
        initstate(seed, random_state, 256);
```

For text items like the random seed, *optsw_getvalue* behaves slightly differently. It stores characters into a buffer addressed by its second argument, and returns the count of characters stored. To allow for strings which are larger than the buffer, the destination is actually a struct which includes a limit on how many characters may be stored; there is provision for multiple calls to *optsw_getvalue* to collect all of a long value through a short buffer (this is detailed in *Option Subwindow* in the reference manual).

Note: Probably the worst bug, the mouse tool ignores all this, assuming that no user will enter more than 256 characters into the seed item.

Having retrieved the string, we convert it to an integer and store it in *seed*. (This is a rather less dangerous reliance on cooperative users, since *atoi* will return 0 if it can't make sense of the string; that's as good a seed as any other.) Finally, that seed is used to initialize the random number generator to the desired sequence.

This completes the preparations needed for a sequence of trials. What remains is to put up the first target to start the sequence:

```
        set_target();
    }
```

This procedure is discussed below.

With that call, we have completed the processing required in response to activity in the option subwindow.

6.7.3. Responding to User Inputs in the Target Range

The last circumstance to which the mouse tool must respond is input in the range subwindow. Because of the way the input mask for this window was initialized, this occurs only when the user clicks the left mouse button in the window. That is, each invocation of this procedure corresponds to a trial on a single target.

Most routines that only handle window input tend to have similar structures, at least at the beginning:

```
static
range_selected(sw, ibits, obits, ebits, timer)
caddr_t          sw;
int              *ibits, *obits, *ebits;
struct timeval   **timer;
{
    int                     x, y;
    struct inputevent       ie;

    if (input_readevent(range_tsw->ts_windowfd, &ie) == -1) {
        perror("input_readevent failed");
        abort();
    }
    *ibits = *obits = *ebits = 0;
```

The arguments to a *selected*-handler are useful in the general case where one routine may be attending to multiple I/O devices and a timer as well; they allow the routine to determine what event actually needs to be responded to on each call. But for the mouse tool, there is no timer to be reset; the "ibits" masks will have the one bit turned on that indicates that input is available on the range subwindow's *fd*; and *sw* will hold the contents of the range subwindow's *ts_data*, which we never stored anything useful into. So this part of the procedure is just a boilerplate.

The call to *input_readevent* will fill *ie* with the next available input event, or draw attention to a system error. Making the "*x*bits" masks zeros resets the specification of what I/O devices the range subwindow manager is expecting activity on. All zeros indicates the default case, which is waiting for user input in the window.

Note: Failing to reset these bits before returning from a *selected* routine can cause a problem.

The next step is to extract the information we want from the event struct. By the way we set the input mask, we know this is a button-down on the left mouse button. We want to know when and where it happened:

```
x = ie.ie_locx; y = ie.ie_locy; stop_time = ie.ie_time;
```

To avoid biasing its results by the time needed to move the cursor from the "Start" button down into the range window, the tool does not count the first target. Rather, it uses the first click as the signal to reset counters and state to begin a new sequence:

```
if (!running) {
    cum_time = trials_done = errors = 0;
    running= TRUE;
```

If the sequence of trials is already under way, this trial must be accounted:

```
} else {
    trials_done += 1;
    if (x < cur_x || y < cur_y ||
       x >= cur_x + size || y >= cur_y + size) {
        errors += 1;
```

A trial is averaged into the running time only if the user actually hit the target; otherwise, the count of misses is incremented.

If the user hit the target, the number of microseconds since the target was put up is calculated and accumulated:

```
        } else {
            if (stop_time.tv_usec < start_time.tv_usec) {
                stop_time.tv_usec += 1000000;
                stop_time.tv_sec -= 1;
            }
            cum_time += stop_time.tv_usec - start_time.tv_usec +
                (stop_time.tv_sec - start_time.tv_sec) * 1000000;
        }
```

Then the old target is taken down:

```
        pw_writebackground(range_pixwin, cur_x, cur_y, size, size, PIX_CLR);
```

the results of that trial are posted:

```
        report();
```

and the next trial is begun:

```
        set_target();
    }
```

Two small subroutines remain to be described. *Set_target* puts up a target and notes the time:

```
    static
    set_target()
    {
        cur_x = random() % width_limit;
        cur_y = random() % height_limit;
        pw_writebackground(range_pixwin, cur_x, cur_y, size, size, PIX_SET);
        gettimeofday(&start_time, &trash_zone);
    }
```

This is straightforward: random coordinates are chosen for the new target, clamped to keep the target inside the window boundaries; a black square of side *size* is written at those coordinates; and the time it was put up is noted.

The *report* procedure, which posts the most recent results, is not much more complicated:

```
static
report()
{
        int         good_trials = trials_done - errors;
        double      mean_time;
        char        buf[256];

        if (good_trials == 0) {
                mean_time = 0;
        } else {
                mean_time = ((float)cum_time / (float)good_trials ) /  1000000;
        }
        sprintf(buf, "%d trials; %d errors; mean time: %f sec.",
                        trials_done, errors, mean_time);
        msgsw_setstring(msw, buf);
}
```

Having calculated the most recent results and used *sprintf* to format a line with them, we request the message subwindow to change its display to this new line.

This completes discussion of the normal tool processing for the mouse tool. Note that all of the processing is handled as small procedures which respond quickly to specific events. There is very little global intelligence required. Note also that most of the work is done by calling on established utilities. Even in the range subwindow, the amount of work that needs to be done is easily controlled by initializations which allow us to ignore the vast body of uninteresting events.

6.8. Terminating the Tool

When *tool_select* returns, the two lines of code at the end of *main* are reached:

```
tool_destroy(tool);
pw_pfsysclose();
exit(0);
```

Tool_select returns when some procedure calls *tool_done*, which sets another flag in the tool's data. It would be possible to have such a call somewhere in the mouse tool code — a "Quit" command in an option subwindow is a fairly standard way of getting to one — but in this case, we leave it to the standard tool manager menu.

Library routines which implement the *tool* object include procedures for displaying and processing a standard menu, one of whose items is "Quit." When this menu item is selected (and after the user confirms his intentions), the tool menu handler calls *tool_done*. This causes *tool_select* to break out of its loop and return, which leads to the call to *tool_destroy*.

Library implementations like the option subwindow tend to be more careful about cleaning up than we were for the range window, since they have no guarantee that their client is about to exit — the resources they've used may be needed by some other procedures in a new phase of processing. *Tool_destroy* checks each subwindow of the tool in turn for a subwindow *destroy* procedure and invokes it if it exists. After all subwindows have cleaned up their own areas, the tool does the same with its private data.

Finally, everything has been cleaned up, and the tool exits normally.

Chapter 7

Writing a More Sophisticated Canvas Program

This chapter describes a canvas program called *canvasinput*, an interactive extension of *canvasflash*, which was described in *Writing a Simple Canvas Program*. Be sure you understand *canvasflash* as we discuss only what is different about *canvasinput* here.

Canvasinput creates a retained graphics subwindow and then waits for the user to enter a command. The user can use the keyboard or the mouse and a menu to specify that either a vertical line, a black square or a string of text be drawn within the window. He can also clear the window or terminate the program. The user can enter all commands from the menu. In addition, he can press the first letter of the name of a menu item to invoke the associated menu item.

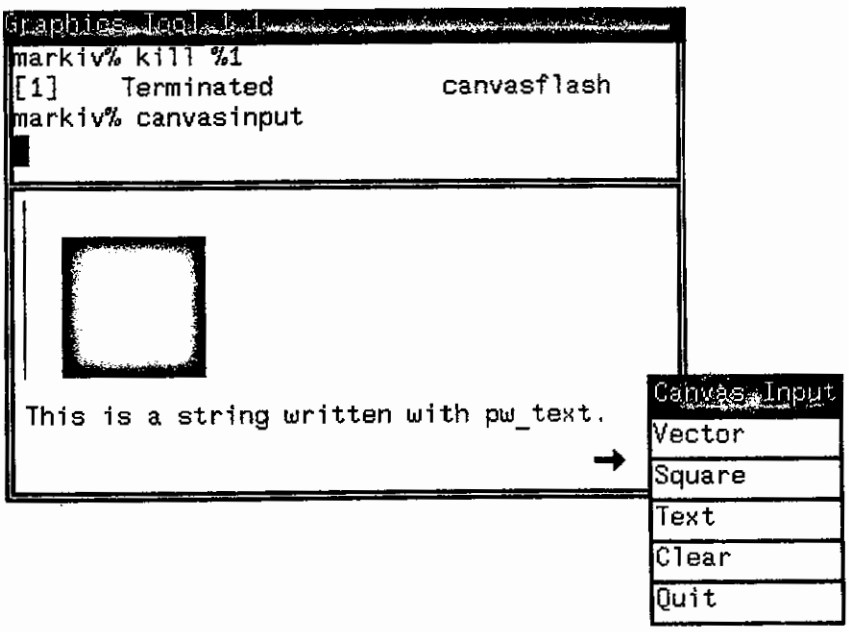


Figure 7-1: *canvasinput* Output

The source to this example is provided in the */usr/suntool/src/canvasinput.c* file. To compile the source, use:

markiv% cc -o canvasinput canvasinput.c -lsuntool -lsunwindow -lpixrect

7.1. The *canvasinput* Code

The flow of *canvasinput* is as follows:

- *Canvasinput* gets a graphics subwindow handle.
- The subwindow is enabled to receive user input.
- The program calls a notification manager. This manager is given the address of a routine, *canvas_selected*, to invoke when input arrives.
- *Canvas_selected* is the guts of this program.

Here is a listing of */usr/suntool/src/canvasinput.c*. You may want to glance at it now, however, it is primarily meant to be referred to as you read the subsequent explanation. Extensive C comments are removed in favor of the accompanying text.

```
#ifndef lint
static  char sccsid[] = "@(#)canvasinput.c 1.1 84/04/04 SMI";
#endif

 /* External Declarations */
#include <stdio.h>
#include <suntool/gfx_hs.h>
#include <suntool/menu.h>

extern    struct menuitem *menu_display();

static    struct gfxsubwindow *gfx;

 /* Menu Definition */
static    struct menuitem menu_items[] = {
    {MENU_IMAGESTRING, "vector",     (caddr_t)'v'},
    {MENU_IMAGESTRING, "square",     (caddr_t)'s'},
    {MENU_IMAGESTRING, "text",       (caddr_t)'t'},
    {MENU_IMAGESTRING, "clear",      (caddr_t)'c'},
    {MENU_IMAGESTRING, "quit",       (caddr_t)'q'},
};
static    struct menu menu_body = {
    MENU_IMAGESTRING, "Commands",
    sizeof(menu_items) / sizeof(struct menuitem), menu_items,
    (struct menu *)NULL, (caddr_t)NULL
};
static    struct menu *menu_ptr = &menu_body;

main(argc, argv)
    int argc;
    char **argv;
{
    int  canvas_selected();
    struct    inputmask im;

    /* Initialization */
    if ((gfx = gfxsw_init(0, argv)) == NULL) {
        fprintf(stderr, "Unable to open graphics subwindow.\n");
        exit(1);
    }
    input_imnull(&im);
    im.im_flags |= IM_ASCII | IM_NEGEVENT;
    win_setinputcodebit(&im, MENU_BUT);
    gfxsw_setinputmask(gfx,
```

```
        &im, (struct inputmask *)NULL, WIN_NULLLINK, 1, 1);
    gfxsw_getretained(gfx);

    /* Notification Manager */
    gfxsw_select(gfx, canvas_selected, 0, 0, 0, (struct timeval *)NULL);

    /* Cleanup */
    gfxsw_done(gfx);
}

/* Notification Handling */
canvas_selected(gfx, ibits, obits, ebits, timer)
    struct  gfxsubwindow *gfx;
    int     *ibits, *obits, *ebits;
    struct  timeval **timer;
{
    struct  menuitem *mi;
    struct  inputevent ie;

    if (gfx->gfx_flags & GFX_RESTART) {
        gfx->gfx_flags &= ~GFX_RESTART;
        pw_writebackground(gfx->gfx_pixwin, 0, 0,
          gfx->gfx_rect.r_width, gfx->gfx_rect.r_height, PIX_CLR);
    }
    if (*ibits & (1 << gfx->gfx_windowfd)) {
        if (input_readevent(gfx->gfx_windowfd, &ie)) {
            perror("canvasinput");
            exit(1);
        }
        if (ie.ie_code == MENU_BUT && win_inputposevent(&ie) &&
          (mi = menu_display(&menu_ptr, &ie, gfx->gfx_windowfd)))
            ie.ie_code = (short) mi->mi_data;
        switch (ie.ie_code) {
        case 'v':
            pw_vector(gfx->gfx_pixwin, 5, 5, 5, 100, PIX_SET, 0);
            break;
        case 's':
            pw_writebackground(gfx->gfx_pixwin,
              25, 25, 75, 75, PIX_SET);
            break;
        case 't':
            pw_text(gfx->gfx_pixwin, 5, 125, PIX_SRC,
              (struct pixfont *)NULL,
              "This is a string written with pw_text.");
            break;
        case 'c':
            pw_writebackground(gfx->gfx_pixwin, 0, 0,
              gfx->gfx_rect.r_width, gfx->gfx_rect.r_height,
              PIX_CLR);
            break;
        case 'q':
            gfxsw_selectdone(gfx);
            break;
        default:
            gfxsw_inputinterrupts(gfx, &ie);
        }
    }
    *ibits = *obits = *ebits = 0;
}
```

7.2. External Declarations

This section describes the explicit external declarations, other than the ones described in the chapter on *canvasflash*, that must be included to compile this program.

The menu package that the program uses is part of the *suntools* library with include files in */usr/include/suntool*. The include statement:

```
#include <suntool/menu.h>
```

contains the data structure definitions required for using the menu package. The following external reference to the menu manager procedure is required as well:

```
extern  struct menuitem *menu_display();
```

7.3. Defining the Menu

This section describes the static structures that make up the menu that is passed to the pop-up menu manager.

Defining a single menu is a two-step process: first define the menu item array, and second, install those items in a menu object. A menu item is composed of a type, a display data pointer, and 32 bits of data private to the client of the menu manager:

```
static struct menuitem menu_items[] = {
        {MENU_IMAGESTRING,  "vector",   (caddr_t)'v'},
        {MENU_IMAGESTRING,  "square",   (caddr_t)'s'},
        {MENU_IMAGESTRING,  "text",     (caddr_t)'t'},
        {MENU_IMAGESTRING,  "clear",    (caddr_t)'c'},
        {MENU_IMAGESTRING,  "quit",     (caddr_t)'q'},
};
```

Our menu items are of type MENU_IMAGESTRING which means that the display data is a string. We are using the first character of the display data string as our private data. The character will be used to identify the menu item returned by the pop-up menu manager.

A menu object contains a title and a description of its menu items:

```
static struct menu menu_body = {
        MENU_IMAGESTRING, "Commands",
        sizeof(menu_items) / sizeof(struct menuitem), menu_items,
        (struct menu *)NULL, (caddr_t)NULL
};
static struct menu *menu_ptr = &menu_body;
```

The title of our menu is "Commands" and it is of type MENU_IMAGESTRING. The next argument translates to the number of elements in the menu item array which is followed by the address of the array. The second to last field is used when displaying multiple menus, and the last field is reserved for the use of the menu manager.

7.4. Initialization

This section describes the set up (other than that described in the chapter on *canvasflash*) undertaken before entering the notification manager.

In SunWindows, each window has an input mask indicating which actions to receive. This screening reduces the amount of data that an application must process. For instance, if an application is not tracking the mouse, it doesn't need to receive mouse motion events. Also, user actions not sent to one window may be redirected to another window. The following code sets up the input mask that we need:

```
input_imnull(&im);
im.im_flags |= IM_ASCII | IM_NEGEVENT;
win_setinputcodebit(&im, MENU_BUT);
gfxsw_setinputmask(gfx,
    &im, (struct inputmask *)NULL, WIN_NULLLINK, 1, 1);
```

The call to *input_imnull* initializes the input mask *im* to be null. A flag in the mask is set so that ASCII keyboard input is allowed through the mask. *Win_setinputcodebit* is called to enable the menu button.

When the menu button goes down, we will call the menu manager to handle interactions with the user. We know that the menu manager will return when the menu button goes up. A button going down is called a *positive* input event and a button going up is called a *negative* input event. We get positive events for a button by default when we call *win_setinputcodebit*. We enable all negative input events for which we have enabled a corresponding positive input event in the mask by setting IM_NEGEVENT in the mask flags.

Gfxsw_setinputmask sets the mask *gfx->gfx_windowfd*. This is the mask that we have defined that the graphics subwindow uses. The NULL third argument to *gfxsw_setinputmask* indicates that we have no desire to dispose of any events already queued for this window. (There should be none, since the window hasn't appeared on the screen yet.) The WIN_NULLLINK in the fourth argument indicates that the next window to be offered an input event that the graphics subwindow doesn't want should be the default, namely its parent. The last two non-zero arguments indicate that we expect both mouse and keyboard input, respectively.

Note: Don't confuse the calling sequence of *gfxsw_setinputmask* with the lower level *win_setinputmask*. *Win_setinputmask* is called for windows in general. *Gfxsw_setinputmask* is called for graphics subwindows in particular.

Next we tell the graphics subwindow manager to manage a retained pixwin:

```
gfxsw_getretained(gfx);
```

With a retained pixwin, the graphics subwindow manager maintains a backup copy of the window image. If part of the graphics subwindow becomes exposed, the graphics subwindow manager repaints the damaged area from the retained pixwin. Without a retained pixwin, it is the programs responsibility to reair the damaged areas. We pay for a retained pixwin with the extra memory that is allocated for the backup copy of the window image. Also, for every pixwin write in our program a write is made to the backup image as well as the window.

7.5. Notification Manager

Now that initialization is done, we are ready to wait for user actions to drive the command interpreter of the program. The *gfxsw_select* routine waits for input.

```
gfxsw_select(gfx, canvas_selected, 0, 0, 0, (struct timeval *)NULL);
```

Gfxsw_select is the notification manager for this graphics subwindow. For user actions that pass through the input mask, *gfxsw_select* calls *canvas_selected.* that is used in a canvas program. *Canvas_selected* is called when there is input available on the graphics subwindow. It is possible to have *canvas_selected* called in other situations, such as output pending, input pending on devices other than the graphics subwindow, or a timer expiring by defining non-NULL values to the last four parameters. However, since the last four parameters are NULL, the notification manager, by default, only waits for input available on the subwindow.

Gfxsw_select loop indefinitely and can be terminated by a call to *gfxsw_selectdone,* which is described below.

7.6. Handling Notifications

This section describes what is going on in the *canvas_selected* routine. *Canvas_selected* is called when something interesting has happened that the canvas program should react to.

The graphics subwindow notification manager calls *canvas_selected* when the size of the window changes.

```
if (gfx->gfx_flags & GFX_RESTART) {
    gfx->gfx_flags &= ~GFX_RESTART;
    pw_writebackground(gfx->gfx_pixwin, 0, 0,
        gfx->gfx_rect.r_width, gfx->gfx_rect.r_height, PIX_CLR);
}
```

The GFX_RESTART flag is set when the size of the window changes. *Canvasinput* clears the flag and simply clears the window image.

Now we see if input is pending on the graphics subwindow:

```
if (*ibits & (1 << gfx->gfx_windowfd)) {
```

**Ibits* is a mask of the file descriptors that have input pending. If the bit position that corresponds to the graphics subwindow is set, there is input pending on the graphics subwindow.

Next, we read a single input event. An input event is a packet of information that describes the state of the input devices when the event occurred. The input event describes the event identifier, the position of the mouse, the time of day, and the state of shift buttons.

```
if (input_readevent(gfx->gfx_windowfd, &ie)) {
    perror("canvasinput");
    exit(1);
}
```

The call to *input_readevent* will fill *ie* with the next available input event, or draw attention to a system error. The next step is to instigate menu processing if the menu button went down.

```
if (ie.ie_code == MENU_BUT && win_inputposevent(&ie) &&
    (mi = menu_display(&menu_ptr, &ie, gfx->gfx_windowfd)))
```

We want to start menu processing when the right mouse button goes down. *Ie.ie_code* is equal to MENU_BUT when the input event concerns the right mouse button. *Win_inputposevent(&ie)* returns true when the input event is positive; that is, the button went down. When these tests are true the menu manager *menu_display* is called.

Menu_display is responsible for displaying menu(s), tracking the mouse over the menu items, and returning a menu item handle. A NULL menu item handle is returned if no item was chosen. *Menu_display* takes a pointer to a menu pointer so that, in the case of stacked menus, the top menu can be returned via modifying *menu_ptr*. The input event handle that prompted the menu action and the graphics subwindow are passed in as well.

If the user made a menu choice, the long word of private data associated with the menu item is placed in the input event:

```
ie.ie_code = (short) mi->mi_data;
```

This is done because we know that the private data contains values that are equal to the characters that are tested for in the following **switch** statement. Thus, we simulate the case where the user typed a single character at the keyboard.

For the most part, the arms of the **switch** statement on the input event are similar to the code described in the chapter about the *canvasflash* program. One notable exception is that you can use a NULL pixfont handle when calling *pw_text* to mean the default system font:

```
pw_text(gfx->gfx_pixwin, 5, 125, PIX_SRC,
    (struct pixfont *)NULL,
    "This is a string written with pw_text.");
```

The default arm of the **switch** statement is used to look for and act on some common interrupt character sequences (such as, ^C, DEL, ^D, ^Z, and control-shift-backslash) used with terminal-based programs:

```
gfxsw_inputinterrupts(gfx, &ie);
```

Some programs dynamically change the collection of input and output devices on which they wish to wait. To accommodate such programs, before returning from *canvas_selected* back to the notification manager, the conditions under which *canvas_selected* will be called again must be respecified.

```
*ibits = *obits = *ebits = 0;
```

Here we make the input(**ibits*), output(**obits*) and exception(**ebits*) masks zero. Since all the masks are 0, the notification manager, by default, only waits for input available on the subwindow. Details on mask usage are available in the reference manual under *toolio*.

Note: Don't forget to reset these bits before returning; this is a common programming error.

7.7. Termination and Cleanup

The call to *gfxsw_selectdone* is made to tell the graphics subwindow notification manager, *gfxsw_select*, that it should return to its caller. In this case, *gfxsw_select* will return to *main* after *canvas_selected* returns. Clean up after returning from *gfxsw_select* is as described in the chapter on *canvasflash*.

7.8. Review

After reading this chapter, you know:

- that interactive canvas programs can be written using the graphics subwindow package,
- the programming basics of using the pop-up menu package,
- that using a retained pixwin can simplify your program but has time and space penalties, and
- what input masks are used for and what input events are.

Chapter 8

Additional Topics

Here we discuss how to implement a subwindow package and describe the program, called *suntools*, that initializes and terminates a window environment.

8.1. Implementing a Subwindow Package

As we have previously discussed, most SunWindows clients use standard subwindow packages to supply most of their user interface. However, when addressing an application area with additional user interface requirements, the programmer must either modify an existing subwindow package or write a new one. There are certain conventions to observe to have the new package work properly with the existing SunWindows facilities. These conventions are described below and in *Minimum Standard Subwindow Interface* in the *Programmer's Reference Manual for SunWindows*.

There is a procedure in all of the existing subwindow packages that creates an instance of the subwindow type within the *tool* structure. This instance is part of the *suntool* layer. However, the subwindow packages do not assume that they necessarily exist within the framework of the *tool* structure. This allows the ambitious user interface programmer to replace portions of the *suntool* layer without having to re-write all of the subwindow packages.

8.1.1. Facilities Provided By All Subwindows

Every subwindow package must provide the same minimum set of facilities. Given a package called "package," we have five routines, whose name are constructed according to this convention:

package_*init*
: Initializes new instance of package's subwindow type

package_*selected*
: Handles notifications of timeout and input available events

package_*handlesigwinch*
: Processes repaint requests, detects and handles changes in subwindow size

package_*done*
: Releases window without closing window's file descriptor; deallocates resources

package_*createtoolsubwindow*
: Creates *toolsw* structure, adds instance of the subwindow to the tool

These routines are described in more detail below. The descriptions are in terms of the declarations for the empty subwindow. First there's the structure definition for empty subwindow

```
struct    emptysubwindow {
    int     em_windowfd;
    struct pixwin *em_pixwin;
};
```

8.1.1.1. Initialization

"Package-*init*" must be able to take a new window device descriptor, and possibly some additional parameters, and initialize a new instance of the package's subwindow type to use the provided window.

```
struct    emptysubwindow *esw_init(windowfd)
    int     windowfd;
```

This initialization includes all of the set up necessary for both output and input.

8.1.1.2. Notification of Events

The *emptysubwindow* does not accept input or need to use timeouts, so there is no *esw_selected*. It it were to accept input in the future, the procedure declarations for *esw_selected* would look like:

```
esw_selected(esw, ibits, obits, ebits, timer)
    struct      emptysubwindow *esw;
    int         *ibits, *obits, *ebits;
    struct      timevalue **timer;
```

"Package_*selected*" is so name because it is similar to the *select*(2) system call. SunWindows dictates the types and order of the parameters to this routine. The parameters include pointers to file descriptor selection masks and a timer that the routine must explicitly modify. If the package does not support input or need to use timeouts, it can elect not to supply this routine. See *Toolio Structure* and *File Descriptor and Timeout Notifications* in the *Programmer's Reference Manual for SunWindows* for further details.

8.1.1.3. Handling Changes in Windows

```
esw_handlesigwinch(esw)
    struct    emptysubwindow *esw;
```

Each package must provide a routine to process repaint requests and to detect and handle changes in the size of a subwindow managed by the package. By convention, this procedure is named "package_*handlesigwinch*" because it is closely related to the handling of SIGWINCH signals. To detect that a subwindow has changed size, the package must have previously stored away the old size, since the old size is unavailable by the time this procedure is called. Again, SunWindows dictates the argument and return values of this procedure.

8.1.1.4. Releasing Window and Deallocate Resources

```
esw_done(esw)
    struct    emptysubwindow *esw;
```

"Package_*done*" cleanly releases a window being managed by the package. This routine should not close the window's file descriptor, but it must deallocate all resources that the package had allocated. Again, SunWindows dictates the argument of this procedure.

8.1.1.5, Creating Tool Subwindow Structure

"Package_*createtoolsubwindow*" allows the package to cooperate more closely with the *suntool* layer's notion of a *tool*. This routine creates a *toolsw* structure and thereby adds an instance of the subwindow to the tool object identified by the routine's first argument. The first four argument values as well as the return value are dictated by SunWindows.

```
struct toolsw *esw_createtoolsubwindow(tool, name, width, height)
    struct    tool *tool;
    char      *name;
    short     width, height;
```

8.1.2. Subwindow Packages and *createtoolsubwindow*

We now examine the expected behavior of a subwindow package's "package_*createtoolsubwindow*" routine. For reference, we reproduce the definition of the *toolsw* structure below:

```
struct    toolsw {
    struct toolsw *ts_next;
    int       ts_windowfd;
    char      *ts_name;
    short     ts_width;
    short     ts_height;
    struct toolio ts_io;
    int       (*ts_destroy)();
    caddr_t ts_data;
};
```

"Package_*createtoolsubwindow*" is responsible for the following:

- calling *tool_createsubwindow* to allocate and initialize a new *toolsw* structure. The window file descriptor for the subwindow is then available in *toolsw->ts_windowfd* upon return from *tool_createsubwindow.*
- calling the package_*init* function with the subwindow's window file descriptor, and placing the returned pointer to the new subwindow data in *toolsw->ts_data.*
- providing the address of the SIGWINCH handler ("package_*handlesigwinch*") by placing it in *toolsw->ts_io.tio_handlesigwinch.*
- providing the address of the input handler routine ("package_*selected*") by placing it in *toolsw->ts_io.tio_selected.* If the package does not support input, it leaves this field set to the null value that *tool_createsubwindow* initialized it to.

- providing the address of the termination routine ("package_*done*") by placing it in *toolsw->ts_destroy*.

Further details about *tool_createsubwindow* are available in *Subwindow Creation* in the *Programmer's Reference Manual for SunWindows*. See *Writing a More Sophisticated Tool* for an example of typical code in a package's tool subwindow creation routine, namely the code that sets the *range_tsw* variable in the body of the *init_range* routine.

8.1.3. Instance Data

The implementor of a general purpose subwindow package cannot know what all of the ultimate uses of the package will be. In particular, the package should allow a single user process to contain multiple instances of its subwindows. Thus, there must be separate copies of the implementation-specific data for each instance, rather than a single set of variables that are global to the implementation modules. As an example, the string that a message subwindow displays cannot be kept in a static variable in the implementation module, as it would preclude having two message subwindows in a single tool.

SunWindows encourages a framework that involves each subwindow package having two associated data structures. One structure is public to the clients of the package and contains all of the information that the implementor believes it is safe for the clients to have access to. The other structure is private to the package's implementation and contains the information needed to support the subwindow type that clients should not know about. A unique copy of this pair of data structures represents each instance of the subwindow type. For the subwindow types that SunWindows supplies, each subwindow's public structure is named "package*subwindow*" by convention, where *package* is the name of the subwindow package. For example, the public structure associated with the empty subwindow package is named *emptysubwindow*.

Note: the option subwindow has no public data outside the *toolsw* structure. Rather, it supplies procedures for manipulating its private data, much like the pixrect implementation.

8.1.4. Handing Over Control of Input

A programmer often has to decide which piece of the code will handle various input events, such as menu events. Leaving the decision to the programmer, it is important that a subwindow package not swallow input events that it is not interested in. For instance, suppose that the programmer wants the standard *Tool Mgr* menu to be available when the right mouse button is depressed anywhere in his tool (if the right button does not have some other meaning in a subwindow). If one of the subwindow packages used by the tool accepts all of the mouse button events and then simply discards them, it will be impossible to get the *Tool Mgr* menu in a subwindow of that type when a mouse button is depressed. This argument extends to all of the other input events. Thus, a subwindow package should only enable itself for those events for which the package performs non-trivial processing.

8.2. Initialize and Terminate a Window Environment — *suntools*

Most SunWindows programs assume that a user-accessible program does basic set up for the window environment, and that it has been run before they are started. *Suntools* is a standard initialization and termination program which is distributed with SunWindows. *Suntools* does the following:

Initializes the window environment
: establishing a window database, and opening the I/O devices that will be used.

Provides a root window
: for the display and maintains its image.

Provides the Root Manager menu
: which allows creation of shell and graphics tools.

Automatically starts up
: a set of tools specified in a file.

Provides for exiting
: the window system.

Appendix A

Glossary

This glossary defines the terms used here that have meanings that are different from their common definitions, those that introduce concepts that are specific to programming in the SunWindows environment, as well as standard terms.

client Software which uses the facilities provided by other software. The mouse tool is a client of the option subwindow package. By a common anthropomorphism, also the programmer of other software.

clock tool A simple tool that continuously updates a display of the time of day.

damage That portion of a window that needs to be repainted to restore the window's image integrity; exposure of a previously invisible area of a window.

canvas program
A SunWindows program that owns only one window.

caret The location at which type-in is inserted or other text editing functions performed. Note that in other systems this location is sometimes called the "cursor".

cursor A small image that moves about the screen in response to mouse motions to indicate the position of the mouse. See *caret*.

graphics tool
A tool that provides display space to canvas programs.

handle A pointer to an object.

icon A small graphic identifying image that represents rather than displays the contents of a window.

icon tool A tool for creating and modifying icon and cursor images

manager The software which creates and manipulates an object.

menu A displayed list of related items for user choice.

object A piece of data, usually a C structure, used by a piece of software to implement a certain abstraction.

overlapping windows
Windows that may obscure one another on the display.

painting A general term for setting pixel values to form an image; includes painting text as well as pictures.

pane tool A sample tool that demonstrates some user interface utilities.

pixel A single displayable point on a screen or in memory; a picture element.

pixrect A structure which binds together the definition of a rectangle of pixels and the set of operations which are used to manipulate them; an access method for rectangular pixel data.

pixrect layer
The layer of SunWindows that provides a uniform interface to devices which can hold raster images.

pixwin A pixel window; an object which encapsulates the locking and clipping information needed to support a multi-window system.

private data and procedures
Elements of the implementation of a package which are not made available to the package's clients. Clients should never have to access these elements, and should not be able to detect any changes to their implementations.

public data and procedures
Elements of the implementation of a package which are defined as its interface to its clients.

rect A structure that defines a rectangle.

rectlist A structure which uses a list of *rect*s to define a complex sub-region of a rectangle.

repair Regenerating the image for a part of a window which has just become visible ("is damaged").

retained window/pixwin
A pixwin on a display that maintains a backup copy in memory of the window's image. This allows fast repair of arbitrarily complex images, at the cost of a fixed overhead in painting.

shell tool A terminal emulator tool.

stacked menus
A set of menus which are all presented at the same time in a display which resembles an offset stack of papers: the header of each menu is visible below and slightly to the left of the header of the menu behind it, and the items of the top menu are also available for selection.

subwindow A window which is subordinate to another. This is established by a structural relationship in the window database, and implies that the subwindow is contained within and displayed on top of the other.

subwindow abstraction
An implementation which provides subwindows of a particular type with particular capabilities to clients; an instance of such a subwindow.

subwindow object
A C structure used by a subwindow package to implement a subwindow abstraction.

subwindow handle
A pointer to a subwindow object.

subwindow package
The software that performs a useful service which can be plugged into a tool object because it meets the programmatic interface requirements of a subwindow object.

suntool layer
The layer of SunWindows that provides the user interface utilities.

SunWindows
The Sun window system.

sunwindow layer
The layer of SunWindows that maintains a database of windows, provides imaging, locking and clipping support for multiple windows, and distributes user inputs among multiple windows.

tiling Arranging elements in a planar figure (such as subwindows within a parent window) in such a fashion that they cover that figure completely and do not overlap among themselves.

tool A program written using the suntool library which includes a *tool window*. More generally, a program that owns more than one window.

tool window
The underlying window UNIX device for presenting the visible image of a tool.

up-down encoded keyboard
A device which generates two distinct signals when a key is pressed and then released.

user A person using the system.

window Generally a rectangular display area, along with the process or processes responsible for its contents; specifically a UNIX device for multiplexing access to a screen surface.

window management
The activity of changing a window's size, position or overlapping relationship with other windows.

Appendix B

Bibliography

Foley, J.D. and Van Dam, A., *Fundamentals of Interactive Computer Graphics*, Addison-Wesley, 1983. Presents graphics concepts and a complete graphics application program written in Pascal.

Kernighan, Brian W. and Ritchie, Dennis C., *The C Programming Language*, Prentice-Hall, Inc., 1978. Tutorial instruction on the C programming language; descriptions of the major features; and reference material.

Newman, William M. and Sproull, Robert F., *Principles of Interactive Computer Graphics*, McGraw-Hill, Inc., 1979. The classic exposition of interactive computer graphics, especially for raster graphics.

Index

READER COMMENT SHEET

Dear Customer,
We who work here at Sun Microsystems wish to provide the best possible documentation for our products. To this end, we solicit your comments on this manual. We would appreciate your telling us about errors in the content of the manual, and about any material which you feel should be there but isn't.

Typographical Errors:
Please list typographical errors by page number and actual text of the error.

Technical Errors:
Please list errors of fact by page number and actual text of the error.

Content:
Did this guide meet your needs? If not, please indicate what you think should be added or deleted in order to do so. Please comment on any material which you feel should be present but is not. Is there material which is in other manuals, but would be more convenient if it were in this manual?

Layout and Style:
Did you find the organization of this guide useful? If not, how would you rearrange things? Do you find the style of this manual pleasing or irritating? What would you like to see different?