Part Number 800-1116-01
Revision: D of 7 January 1984
For: Sun System Release 1.1

# Programmer's Reference Manual

*for*

# SunWindows

## the Sun Window System

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

# Credits and Acknowledgements

A preliminary implementation of the Sun Window System was written at Sun Microsystems, Inc. in December 1982 and January 1983. It incorporated a number of low-level operations and data, including raster operations and fonts, provided by Tom Duff of Lucasfilm, Ltd.

# Revision History

| Rev | Date | Comments |
|---|---|---|
| A | 15 July 1983 | Preliminary draft release of this Programmer's Reference Manual. |
| B | 15 September 1983 | 0.9 release of this Programmer's Reference Manual. |
| C | 1 November 1983 | Additions to pixrect creation, input handling, and tool facilities. |
| D | 7 January 1984 | Many corrections; additions, changes, and deletions to user interface, option subwindow, graphic subwindow, and window manager; changes to sunwindow library to accommodate color and multiple screens and to the pixrect library to support color pixrects. |

# Changes in the 1.1 Release of SunWindows

This notice describes changes, deletions, and additions to SunWindows from release 1.0 to release 1.1. The main differences in 1.1 center around adding support for multiple screens and color displays.

SunWindows includes sources for window application programs in */usr/suntool/**, the suntool library */usr/lib/libsuntool.a*, the sunwindow library */usr/lib/libsunwindow.a*, and the pixrect library */usr/lib/libpixrect.a*. For *suntools*(1) user information, refer to the *User's Manual for the Sun Workstation*. For more detailed SunWindows programming information, refer to the reference section of the *Programmer's Reference Manual for SunWindows*.

## Upgrading from 1.0 to 1.1 SunWindows

1.0 programs must be recompiled to run in 1.1.

## User Interface Differences — Changes to /usr/suntool/

### Additions to the User Interface

Additions to the user interface are:

*suntools*
: Now takes an extensive argument list to control the environment of the window system. This includes indicating color, which screen, inversion, and so on. See OPTIONS in *suntools*(1) in the *User's Manual for the Sun Workstation* for details.

*spheresdemo*
: Now produces multiple colored spheres. A –g command line argument produces varying shades of gray spheres. These grays may not appear gray until the cursor is positioned in the window in which the spheres are being drawn.

*jumpdemo*
: Now produces colored vectors. A –c command line argument causes the vectors to sparkle via colormap rotation.

System Fonts
: More fonts are available for use as the DEFAULT_FONT. See */usr/suntool/fixedwidthfonts/**

and *suntools*(1) in the *User's Manual for the Sun Workstation*.

Exiting suntools
Typing ^D followed by a ^Q to the Root Window exits *suntools*.

### New Programs

*adjacentscreens*(1)
Tells the window system the physical relation of screens.

*lockscreen*(1)
Puts a "lock" on and hides the current window context so logging out is no longer necessary.

*perfmon*(1)
A graphic performance monitor.

## Suntool Library

The changes to the suntool library involve several changes to the option subwindow interface, making the graphics subwindow more robust, and simplifying window management utilities.

### Option Subwindow Changes

Three routines now take different arguments, return different values, and/or behave differently than they used to. These are:

*optsw_text*
Takes a fifth argument, the address of a notify procedure, exactly as for the other item-creation routines. The notify procedure is called whenever the value of the text item is changed, except by a call to *optsw_setvalue*. It will be called with handles for the option subwindow and the item which changed. *Optsw_getvalue* should be used to actually retrieve the new value. This parameter to *optsw_text* may be NULL to indicate "no notification."

*optsw_getvalue*
Behaves differently for text items; its second (destination) argument should now be a pointer to a struct *string_buf*, as defined in *optionsw.h*. This protects against the case where the value of the item is longer than the client's buffer. In such a case, the buffer is filled, and the max count is returned; no terminating NULL is written in the client's buffer. A subsequent call to *optsw_getvalue* for that item will return the next fragment, until the whole value has been reported. A terminating NULL is written in the buffer when there is room for it, and a subsequent call to *optsw_getvalue* will start anew at the beginning of value.

*optsw_setplace*
Has had its arguments changed to be parallel with *optsw_getplace*. It third argument is now a pointer to a struct *item_place*, instead of the struct *rect* pointer it used to take; the struct contains a rect, and four boolean bit flags indicating that a value is to be fixed for that item.

### Option Subwindow Deletions

The *struct opt_item* is no longer defined in a public header file. Routines which used to return a pointer to such a struct (all of the item-creation routines, for instance) now return an opaque pointer (*caddr_t*). Routines which took a pointer to such a struct as an argument now accept the opaque pointer. Inquiry and manipulation functions are provided to support access to the items without commitment to their internal representation.

### Option Subwindow Additions

Two new structs are defined, one for *optsw_getplace* and *optsw_setplace*, one for *optsw_getvalue* on text items:

*struct item_place*
: Encodes the information about an item's size and location which the client may see and modify. A pointer to such a struct is passed to *optsw_getplace* (which fills it in) and *optsw_setplace*, which uses it to establish an item's location, size, and willingness to change.

*struct string_buf*
: Provides a counted buffer for text items' values to be stored into. Limit should be the size of the buffer on a call to *optsw_getvalue*.

The following new routines are also provided:

*optsw_getcaret(osw)*
: Returns an item handle for the item which currently has the caret in *osw*, or NULL if there is no text item in *osw*.

*optsw_setcaret(osw, ip)*
: Makes the optionsw text item referred to by *ip* be the one which has the caret in the indicated optionsw.

*optsw_getfont(osw)*
: Returns a pointer to the struct *pixfont* which is currently being used by the optionsw.

*optsw_getplace(osw, ip, place)*
: Stores into the *item_place* struct pointed to by *place* a description of the size, position, and fixedness of the item indicated.

*optsw_nextitem(osw, ip)*
: Given an item in an optionsw, returns a handle for the next item in sequence.

*optsw_removeitems(osw, ip, count, reformat)*
: Removes at most *count* items from *osw*, making them inaccessible to the user, but not destroying them.

*optsw_restoreitems(osw, ip, count, reformat)*
: Restores at most *count* items in *osw*, starting at the item indicated by *ip*; returns the number restored.

### Graphic Sunwindow Changes

Graphic sunwindow changes to the interface are:

*gfxsw_setinputmask*
Should be called instead of *win_setinputmask*. This call takes additional arguments as well.

### Graphic Sunwindow Deletions

The graphics sunwindow procedure *gfxsw_cleanup* was removed from the interface because it is now obsolete due to the new implementation of the graphic subwindows. Graphics subwindows now use blanket windows instead of the old window takeover mechanism. You can instead call *gfxsw_done*, or do nothing at all, from SIGINT and SIGHUP handling routines.

### Graphic Sunwindow Additions

Graphic sunwindow additions to the interface are:

*gfxsw_catchsigwinch*
Catches and handles SIGWINCH.

*gfxsw_catchsigtstp*
Catches and handles SIGTSTP.

*gfxsw_catchsigcont*
Catches and handles SIGCONT.

*gfxsw_notusingmouse*
May be called if your program doesn't use the mouse; this is optional.

*gfxsw_inputinterrupts*
A substitute utility for tty process control while using the window input mechanism.

### Window Management Deletions

Window management interface deletions include:

*Wmgr_changelevelonly*
Removed in favor of the similar procedure *wmgr_changelevel*.

*Wmgr_changestate*
Was removed in favor of the similar procedures *wmgr_open* and *wmgr_close*.

*Wmgr_setupmenus*
Removed in favor of *wmgr_setupmenu*. The interface no longer supports *wmgr_rootmenu* (moved into client code, see the *suntools.c* source), thus, the change of plurality.

### Window Management Additions

Window management additions to the interface are:

*wmgr_open, wmgr_close, wmgr_move, wmgr_stretch, wmgr_top, wmgr_bottom, wmgr_refreshwindow* Are the highest level window management routines and correspond exactly to tool menu operations.

*wmgr_changerect*
Provides finer control of moving and stretching user interaction.

*wmgr_confirm*
A standard confirmation utility.

*wmgr_handletoolmenuitem*
Switch to call top level window management routines based on *wmgr_toolmenu* menu item chosen.

*wmgr_setrectalloc* and *wmgr_getrectalloc*
Global storage of next default window position.

## Sunwindow Library

The changes to the suntool library center around keeping up with the pixrect library changes by providing a pixwin operation to match each pixrect operation and cleaning up the interface to screen structures.

### Changes to the Interface

The screen struct was completely overhauled to accommodate color and multiple screens. However, the *scr_rect* was left untouched and is the field that high level clients most often use. Therefore, no source changes should probably be required by most programs.

*Win_screennew*
Now has a different calling sequence. It now takes a struct screen pointer and returns a window file descriptor. It used to take a window file descriptor and and struct screen pointer.

*Win_screenpositions*
Was renamed *win_setscreenpositions*.

### Deletions from the Interface

With the advent of blanket windows, using *win_setowner* to temporarily change ownership of windows is no longer recommended.

### Additions to the Interface

The pixwin additions that correspond to the equivalent pixrect additions are:

*pw_region*
Pixwin region operation.

*pw_ttext*
Pixwin transparent text operation.

*pw_batchrop*
Pixwin batchrop operation.

*pw_stencil*
Pixwin stencil operation.

*pw_putattributes* and *pw_getattributes*
Pixwin attributes control.

*pw_putcolormap* and *pw_getcolormap*
Pixwin colormap control.

The pixwin additions that extend pixrect functionality are:

*pw_setcmsname* and *pw_getcmsname*
Pixwin colormap segment name access.

*pw_preparesurface*
Pixwin surface preparation (colormap segment related).

*pw_cyclecolormap*
Pixwin colormap utility.

The pixwin font utilities that share the system font are:

*pw_pfsysopen* and *pw_pfsysclose*
Sharing of the default system font is provided.

The screen-related additions are:

*win_getscreenpositions*
Retrieve neighbors of the window's screen.

*win_setkbd* and *win_setms*
Change keyboard and mouse devices used by the screen.

*win_initscreenfromargv*
Standard command line to screen specification parser.

These are the additions related to the new blanket window mechanism:

*win_insertblanket*
Insert window into display tree and treat as a "blanket" window (one that always covers its parent).

*win_removeblanket*
Remove blanket window from display tree.

*win_isblanket*
Check 'is a window a blanket window?'

## Pixrect Library

Pixrects features a slightly modified interface to support color pixrects. Also, the font format and memory pixrect format has changed.

### Deletions from the Interface

The create operation is removed from the pixrect operations vector. *Pr_open*, *mem_create* and *pr_region* define the available pixrect creation alternatives.

## Changes to the Interface

A new frame buffer naming convention now exists:

*/dev/fb*
The default frame buffer for a machine. This replaces */dev/console* which has other tty-related functions, and */dev/bw0* which no longer exists.

Frame Buffer Naming Convention
The general naming convention for a frame buffer follows the form */dev/CTU* in which:

*C* is either "bw" (for monochrome displays) or "cg" (for color displays).

*T* is the type of the display, such as "one" or "two" for Sun–1 or Sun–2 respectively.

*U* is the unit number starting from 0, indicating which specific frame buffer.

Some examples of frame buffer names are: */dev/bwone0*, */dev/bwtwo0* and */dev/cgone0*.

The font format used in *pf_open* has been changed. The old format was in VAX byte order. The new format is in Motorola 68000 byte order (reversed from the VAX). You can tell if a font is in the new format by using the *file*(1) program on the font file in question. The font file should be listed as "vfont definition". The program *vswap*(1) converts a font file from the old format to the new.

The structure format of *mpr_data* (the memory pixrect internal data format) has changed slightly. The *Bint md_primary* field has been split into **short** *md_primary* and **short** *md_flags*. The overall length of *mpr_data* remains the same.

## Additions to the Interface

*pr_stencil*
Provides spatial masking of the destination pixrect for control of the areas of the destination pixrect to be painted by the source pixrect.

*pr_putcolormap* and *pr_getcolormap*
Provides a unified colormap and reversevideo interface for both color and monochrome pixrects.

*pr_putattributes* and *pr_getattributes*
Provides access to a bitplane mask which specifies the modifiable bits in destination pixrect pixels.

*pf_ttext*
Uses character bitmap as a stencil through which the specified color is squirted, hence background shows through around the characters.

# Table of Contents

# Table of Contents

# List of Tables

# Preface

The *Programmer's Reference Manual for SunWindows* provides primarily reference material on SunWindows, the Sun window system. It is intended for programmers of applications using window system facilities.

## Manual Contents

The contents of the manual are:

Chapter 1 — *Overview* — Describes basic hardware and software support and the layers of implementation of SunWindows, the *pixrect* layer, the *sunwindow* layer, and the *suntool* layer.

Chapter 2 — *Pixel Data and Operations* — Describes pixel data and operations in the lowest level output facilities of SunWindows, pixrects, pixrectops, memory pixrects, and text facilities for pixrects.

Chapter 3 — *Overlapped Windows: Imaging Facilities* — Explains image generation on windows which may overlap other windows.

Chapter 4 — *Window Manipulation* — Describes the *sunwindow* layer facilites for creating, positioning, and controlling windows.

Chapter 5 — *Input to Application Programs* — Discusses how user input is made available to application programs.

Chapter 6 — *Suntool: Tools and Subwindows* — Discusses how to write a tool, and covers creation and destruction of a tool and its subwindows, the strategy for dividing work among subwindows, and the use of routines provided to accomplish that work.

Chapter 7 — *Suntool: Subwindow Packages* — Discusses *subwindows* as building blocks in the construction of a tool, covers the currently existing subwindows, and suggests the approach for creating new kinds of subwindows.

Chapter 8 — *Suntool: User Interface Utilites* Covers user interface utilities, the independent packages for use with the *suntools* environment, includes the actual window manipulation routines used by *tool windows*, the *icon* facility, the *selection* manager, the *fullscreen* access mechanism, and *menus* and *prompts*.

Appendix A — *Rects and Rectlists* — Describes the geometric structures used with the *sunwindow* layer and provides a full description of the operations on these structures.

Appendix B — *Sample Tools* — Provides an annotated collection of some simple tools to be used both as illustrations and as templates for client programmers; includes a graphics tool, a window pane tool, an option tool, and an icon tool.

Appendix C — *Sample Graphics Programs* — Provides an annotated selection of several graphics programs for writing your own graphics programs; includes code for a bouncing ball

demonstration and for a "movie camera" program that displays files as frames from a movie.

*Note:* The reference section of the *Programmer's Reference Manual for SunWindows* is neither a user guide nor an explanation of the internals of the window system. It presents the material in a bottom-up fashion with primitive concepts and facilities described first. It is not intended to be read linearly front-to-back; glance at the table of contents and the chapters on tools to get a general idea of how to use the rest of the material.

The *User's Manual for the Sun Workstation* provides user information under *suntools*(1) for SunWindows and under the appropriate entry for the particular application programs. The *Beginner's Guide to the Sun Workstation* provides a brief tutorial on general use of the mouse and the SunWindows pop-up menus.

## A Note About Special Terms

Several terms in this manual have meanings distinct from their common definitions or introduce concepts that are specific to programming in the SunWindows environment. We discuss the most important here.

The word *client* indicates a program that uses window system facilities. This is in contrast to *user*, which refers to a human.

Terms referring to display hardware, such as *framebuffer*, *pixel*, and *rasterop*, are used in well-established senses; novices who are confused should consult one of the standard texts, such as *Fundamentals of Interactive Computer Graphics* by J.D. Foley and A. Van Dam, Addison-Wesley, 1983.

The position of the mouse is indicated by a *cursor* on the screen; this is any small image that moves about the screen in response to mouse motions. The term "cursor" is used elsewhere to indicate the location at which type-in will be inserted, or other editor functions performed. The two concepts are not often distinguished. To keep them distinct, we use the term *caret* to refer to the type-in location.

A *menu* is a list of related choice items displayed on the screen in response to a user mouse-action. The user chooses one menu item by pointing at it with the cursor. Such menus are called *transient* or *pop-up*; they are displayed only while a mouse button is depressed, and are typically used for invoking parameterless operations.

A *rect* is a structure that defines a rectangle.

A *rectlist* is a structure that defines a list of rects.

*Up-down encoded keyboards* are devices from which it is possible to receive two distinct signals when a key is pressed and then released.

An *icon* is a small form of a window that typically displays an identifying image rather than a portion of the window contents; it is frequently used for dormant application programs. For example, the default icon for a closed Shell Tool is a conch shell, representing the UNIX "C-Shell".

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

# Chapter 1

# Overview

## 1.1. What is SunWindows?

SunWindows is the Sun window system. It is a *tool box and parts kit*, not a closed, finished, end product. Its design emphasizes extensibility, accessibility at multiple layers, and provision of appropriate parts and development tools. Specific applications are provided here both as examples and because they are valuable for further development. The system is designed to be expanded by clients.

The system is explicitly *layered* with interfaces at several levels for client programs. There is open access to lower levels, and also convenient and powerful facilities for common requirements at higher levels. For instance, it is always possible for a client to write directly to the screen, although in most circumstances it is preferable to employ higher-level routines.

## 1.2. Hardware and Software Support

The Sun Microsystems Workstation provides hardware and software support for the construction of high-quality user interfaces. Hardware features include:

- provision of a processor for each user, a prerequisite for powerful, responsive, cost-effective systems;
- a bit-mapped display which allows arbitrary fonts and graphics to be used freely to make applications programs easier to learn and use;
- hardware support of fast and convenient manipulation of image data;
- a mouse pointing device for selecting operations from menus or for pointing at text, graphics and icons; and
- an up-down encoded keyboard that supports sophisticated function-key interfaces at once simpler and more efficient than most command languages.

Sun software is similarly structured to support high-quality interactions. The software features are:

- a uniform interface to varied pixel-oriented devices that allows convenient incorporation of new devices into the system, and clean access to all these devices by application programs;
- an extended device independence for input such as function keys and locators, to user-interface features;
- a window management facility that keeps track of multiple overlapping windows, allowing their creation and rearrangement at will. The facility arbitrates screen access, detects destructive interactions such as overlapping, and initiates repairs. It also serializes and distributes user inputs to the multiple windows, allowing full type-ahead and mouse-ahead; and

- built on all these facilities, an executive and application environment that provides a system for running existing UNIX programs and new applications, taking advantage of icons, menus, prompts, mouse-driven selections, interprocess data exchange, a forms-oriented interface and useful cursor manipulations.

## 1.3. Layers of Implementation

There are three broad *layers* of SunWindows. These layers may be identified by the libraries that contain their implementations. The organization of the reference part of this manual reflects the three layers as described below.

1. The *pixrect* level provides a device-independent interface to pixel operations.
2. The *sunwindow*[1] level implements a manager for overlapping windows, including imaging control, creation and manipulation of windows, and distribution of user inputs.
3. The *suntool* level implements a multi-window executive and application environment. In its user interface, it includes a number of relatively independent packages, supporting, for instance, *menus* and *selections*.

### 1.3.1. Pixrect Layer

Chapter 2 describes the *pixrect* layer of the system. This level generalizes RasterOp display functions to arbitrary rectangles of pixels. Peculiarities of specific pixel-oriented devices, such as dimensions, addressing schemes, and pixel size and interpretation, are encapsulated in device-specific interfaces, which all present the same uniform interface to clients.

The concept of a pixrect is quite general; it is convenient for referring to a whole display, as well as to the image of a single character in a font. It may also be used to describe the image which tracks the mouse.

There is a balance between functionality and efficiency. All pixrects clip operations that extend beyond their boundaries. Since this may require substantial overhead, clients which can guarantee to stay within bounds may disable this feature. Where hardware support exists, it is taken advantage of without sacrificing generality: All pixrects support the same set of operations on their contents.

These operations include general raster operations on rectangular areas, vectors, batch operations to handle common applications like text, and compact manipulation of constant or regularly-patterned data. A stencil operation provides spatial, two-dimension masking of the source pixrect with a mask pixrect to control the areas of the destination pixrect to be written.

Color pixrects, as well as monochrome pixrects, are well supported. There are uniform operations for accessing a pixrect's *colormap*. A colormap maps a pixel value to a screen color. The pixel planes affected by other operations can be controlled as well. Monochrome pixrects support the same interface as color pixrects. Programs intended primarily for color pixrects usually produce reasonable images on monochrome pixrects, and vice versa.

[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.

## 1.3.2. Sunwindow Layer

Chapters 3 through 5 introduce *windows* and operations on them. A window is a rectangular display area, along with the process or processes responsible for its contents. This layer of the system maintains a database of windows which may *overlap* in both time and space. These windows may be nested, providing for distinct *subwindows* within an application's screen space.

Windows existing concurrently may all access a display; the window system provides locking primitives to guarantee that these accesses do not conflict.

Arbitration between windows is also provided in the allocation of display space. Where one window limits the space available to another, it is necessary to provide *clipping*, so neither interferes with the other's image. One such conflict handled by the *sunwindow* layer arises when windows share the same coordinates on the display: one *overlaps* the other.

When one window impacts another window's image without any action on the second window's part, SunWindows informs the affected window of the damage it has suffered, and the areas that ought to be repaired. Windows may either recompute their contents for redisplay, or they may elect to have a full backup of their image in main memory, and merely copy the backup to the display when required.

On color displays, colormap entries are a scarce resource. When shared among multiple applications, they become even more scarce. Arbitration between windows is provided in the allocation of colormap entries. Provisions are made to share portions of the colormap.

Windows may be created, destroyed, moved, stretched or shrunk, set at different levels in the overlapping structure, and otherwise manipulated. The *sunwindow* level of the system provides facilities for performing all these operations. It also allows definition of the image which tracks the mouse while it is in the window, and inquiry and control over the mouse position.

Separate collections of windows may reside on separate screens. The user interacts with these multiple screens with his 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. 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.

## 1.3.3. Suntool Layer

Chapters 6 through 8 of the reference part of this manual describe the *suntool* level of the system. While the first two layers provide client interfaces, the *suntool* level provides the user interface.

We refer to an application program that is a client of this level of the window system as a *tool*. This term covers the one or more programs and processes which do the actual application processing. It also refers to the collection of windows through which the tool interacts with the user. This collection often includes a special *icon*, which is a small form the tool may take to be unobtrusive on the screen but still identifiable. Simple examples of tools may include a calculator, a bitmap editor, and a terminal emulator. Sun provides a few ready-built tools, several of which are illustrated in Appendix B. Customers are expected to develop their own tools to suit their specific needs.

SunWindows provides some common components of tools:

- an executive framework that supplies the usual "main loop" of a program and coordinates the activities of the various subwindows;
- a standard *tool window* that frames the active windows of the tool, identifying it with a name stripe at the top and borders around the subwindows. Each tool window has a facility for manipulating itself in the overlapped window environment. This includes adjusting its size and position, including layering, and moving the boundaries between subwindows;
- several commonly used *subwindow* types that can be instantiated in the tool;
- a standard scheme for laying out those subwindows; and
- a facility that provides a default *icon* for the tool.

The *suntools* program initializes the window environment. It provides for:

- automatic startup of a specified collection of tools;
- dynamic invocation of standard tools;
- management of the default window called the *root* window, which underlies all the tools; and
- the user interface for leaving the window system.

Users who wish some other form of environment management can replace the *suntools* program, while retaining the tools and supporting utilities.

The facilities provided in the *suntool* library are relatively independent; they can be used with window contexts other than *suntools*. The *icon* facility mentioned above is in this category, as are the window manipulation facilities of *suntools*. There is also a package for presenting *menus* to the user and interpreting the response.

# Chapter 2

# Pixel Data and Operations

This section discusses pixel data and operations in the lowest-level output facilities of SunWindows. These facilities will frequently be accessed indirectly, through higher-level abstractions described in chapters 3 through 8. However, some client implementors will deal at this level, for instance to include new display devices in the window system. The header file */usr/include/pixrect/pixrect_hs.h* includes the header files that you need to work at this level of the window system. It will also suffice to include */usr/include/suntool/suntool_hs.h* or */usr/include/sunwindow/sunwindow_hs.h.*

## 2.1. Pixrects

The fundamental object of pixel manipulation in the window system is the *pixrect.* A pixrect encapsulates a rectangular array of pixels along with the operations which are defined on that data. Pixrects are designed along the model of *objects* in an object-oriented programming system. They combine both data and operations, presenting their clients with a simple interface: a well-defined set of operations produces desired results, and details of representation and implementation are hidden inside the object.

The pixrect presents only its dimensions, a pointer to its operations, and a pointer to private data which those operations may use in performing their tasks. Further, the set of operations is the same across all pixrects, though of course their implementations must differ. This object-oriented style allows similar things which differ in small details to be gathered into a unified framework; it allows clients to use the same approach to all of them, and allows implementors to add new members or improve old ones without disturbing clients.

The pixrect facility satisfies two broad objectives:

- To provide a *uniform interface to a variety of devices* for independence from device characteristics where they are irrelevant. Such characteristics include the actual device (pixrects may exist in memory and on printers as well as on displays), the dimensions and addressing schemes of the device, and the definition of the pixels, that is, how many bits in each, how they are aligned, and how interpreted. Color and monochrome devices use the same interface. Programs intended primarily for color pixrects usually produce reasonable images on monochrome pixrects, and vice versa.
- To provide a proper *balance of functionality and efficiency* for a full range of pixel operations with performance close to that achieved by direct access to the hardware. Pixrect operations include generalized rasterops, vectors, text and other batch operations, compact manipulation of uniform and regularly-patterned data, as well as single-pixel reads and writes. All provide for clipping to the bounds of the rectangle if desired; this facility may be bypassed by clients which can perform it more efficiently themselves. A stencil function provides spatial masking of the source pixrect with a stencil pixrect to control the areas of the destination pixrect to be written. Where specialized hardware exists and can be used for a particular operation, it is, but not at the expense of violating the device-independent interface.

### 2.1.1. Pixels: Coordinates and Interpretation

Pixels in a pixrect are addressed in two dimensions with the origin in the upper left corner, and $x$ and $y$ increasing to the right and down. The coordinates of a pixel in a pixrect are integers from 0 to the pixrect's width or height minus 1.

A pixrect is characterized by a *depth*, the number of bits required to hold one pixel. A large class of displays uses a single bit to select black or white (or green or orange, depending on the display technology). On these *monochrome* displays and in memory pixrects one bit deep, a 1 indicates *foreground* and a 0 *background*. No further interpretation is applied to memory. The default interpretation on Sun displays is a white background and a black foreground.

Other displays use several bits to identify a color or gray level. Typically, though not necessarily, the pixel value is used as an index into a *colormap*, where colors may be defined with higher precision than in the pixel. A common arrangement is to use an 8-bit pixel to choose one of 256 colors, each of which is defined in 24 bits, 8 each of red, green and blue. Pixrect depths less than or equal to 16 are supported.

### 2.1.2. Geometry Structs

As a preliminary to the discussion of pixrects, it is convenient to define a few structs which collect useful geometric information.

The struct that defines a position in coordinates $(x, y)$ is:

```
struct pr_pos {
    int     x, y;
};
```

Leaving a pixrect undefined for the moment, this struct defines a point within a specified pixrect:

```
struct pr_prpos {
    struct      pixrect *pr;
    struct      pr_pos pos;
};
```

It contains a pointer to the pixrect and a position within it.

The following struct defines the width and height of an area:

```
struct pr_size {
    int     x, y;
};
```

The following struct defines a sub-area within a pixrect:

```
struct pr_subregion {
    struct      pixrect *pr;
    struct      pr_pos pos;
    struct      pr_size size;
};
```

It contains a pointer to the pixrect, an origin for the area, and its width and height.

### 2.1.3. The Pixrect Struct

A particular pixrect is described by a *pixrect* struct. This combines the definition of a rectangular array of pixels and the means of accessing operations for manipulating those pixels:

```
struct pixrect {
    struct      pixrectops *pr_ops;
    struct      pr_size pr_size;
    int         pr_depth;
    caddr_t     pr_data;
};
```

The width and height of the rectangle are given in *pr_size*, and the number of bits in each pixel in *pr_depth*. For programmers more comfortable referring to "width" and "height," there are also two convenient macros:

```
#define pr_width  (pr_size.x)
#define pr_height (pr_size.y)
```

All other information about the pixrect (in particular, the location and values of pixels), is data private to it. Pixels are manipulated only by the set of *pixrect operations* described below. These operations will generally use information accessed through *pr_data* to accomplish their tasks.

(This restriction is relaxed somewhat in the case of pixrects whose pixels are stored in memory; this provides an escape to mechanisms outside the pixrect facility for constructing and converting pixrects of differing types. Memory pixrects are described in *Memory Pixrects*.)

## 2.2. Operations on Pixrects

Procedures are provided to perform the following operations on pixrects:

- create and destroy them (*open, region* and *destroy*)
- read and write the values of single pixels (*get* and *put*)
- use RasterOp functions to affect multiple pixels in a single operation:
  - write from a source to a destination pixrect (*rop*)
  - write from a source to a destination under control of a mask (*stencil*)
  - replicate a constant source pattern throughout a destination (*replrop*)
  - write a batch of sources to different locations in a single destination (*batchrop*)
  - draw a straight line of a single source value (*vector*)
- read and write a colormap (*getcolormap, putcolormap*)
- select particular bit-planes for manipulation on a color pixrect (*getattributes, putattributes*)

Some of these operations are the same for all pixrects, and are implemented by a single procedure. These device-independent procedures are called directly by pixrect clients. Other operations must be implemented differently for each device on which a pixrect may exist. Each pixrect includes a pointer (in its *pr_ops*) to a *pixrectops* structure, that holds the addresses of the particular device-dependent procedures appropriate to that pixrect. This allows clients to access those procedures in a device-independent fashion, by calling through the procedure pointer, rather than naming the procedure directly. To facilitate this indirection, the pixrect facility provides a set of macros which look like simple procedure calls to generic operations, and expand to invocations of the corresponding procedure in the pixrectops structure.

The description of each operation will specify whether it is a true procedure or a macro, since some of the arguments to macros are expanded multiple times, and could cause errors if the arguments contain expressions with side effects. (In fact, two sets of parallel macros are provided, which differ only in whether their arguments use the geometry structs defined above. Each is described with the operation.)

## 2.2.1. The Pixrectops Struct

The pixrectops struct is a collection of pointers to the device-dependent procedures for a particular device:

```
struct pixrectops {
    int         (*pro_rop)();
    int         (*pro_stencil)();
    int         (*pro_batchrop)();
    int         (*pro_nop)();
    int         (*pro_destroy)();
    int         (*pro_get)();
    int         (*pro_put)();
    int         (*pro_vector)();
    struct      pixrect *(*pro_region)();
    int         (*pro_putcolormap)();
    int         (*pro_getcolormap)();
    int         (*pro_putattributes)();
    int         (*pro_getattributes)();
};
```

All other operations are implemented by device-independent procedures.

## 2.2.2. Conventions for Naming Arguments to Pixrect Operations

In general, the following conventions are used in naming the arguments to pixrect operations:

Table 2-1: Argument Name Conventions

| Argument | Meaning |
|---|---|
| *d* | destination |
| *s* | source |
| *x* and *y* | left and top origins |
| *w* and *h* | width and height |

## 2.2.3. Creation and Destruction of Pixrects

Pixrects are created by the procedures *pr_open* and *mem_create*, by the procedures accessed by the macro *pr_region*, and at compile-time by the macro *mpr_static*. Pixrects are destroyed by the procedures accessed by the macro *pr_destroy*. *Mem_create* and *mpr_static* are discussed under *Memory Pixrects* below; the rest of these are described here.

### 2.2.3.1. Open: Create a Primary Display Pixrect

The properties of a non-memory pixrect are described by a UNIX device. Thus, when creating the first pixrect for a device you need to open it by a call to:

```
struct pixrect *pr_open(devicename)
    char        *devicename;
```

The default device name for your display is */dev/fb* (*fb* stands for *framebuffer*). Any other device name may be used provided that it is a display device, the kernel is configured for it, and it has pixrect support, such as, */dev/bwone0*, */dev/bwtwo0*, */dev/cgone0*.

*Pr_open* does not work for creating a pixrect whose pixels are stored in memory; that function is served by the procedure *mem_create*, discussed under *Memory Pixrects* below.

*Pr_open* returns a pointer to a pixrect struct which covers the entire surface of the named device. If it cannot, it returns NULL, and displays an error on standard error.

### 2.2.3.2. Region: Create a Secondary Pixrect

Given an existing pixrect, it is possible to create another pixrect which refers to some or all of the same pixels on the same device. This is called a *secondary pixrect*, and is created by a call to the procedures invoked by the macros *pr_region* and *prs_region*:

```
#define struct pixrect *pr_region(pr, x, y, w, h)
    struct      pixrect *pr;
    int         x, y, w, h;

#define struct pixrect *prs_region(subreg)
    struct      pr_subregion subreg;
```

The existing pixrect is addressed by *pr*; it may be a pixrect created by *pr_open*, *mem_create* or *mpr_static* (a *primary* pixrect); or it may be another secondary pixrect created by a previous call to a region operation. The rectangle to be included in the new pixrect is described by *x*, *y*, *w* and *h* in the existing pixrect; (*x*, *y*) in the existing pixrect will map to (0, 0) in the new one. *Prs_region* does the same thing, but has all its argument values collected into the single struct *subreg*. Each *region* procedure returns a pointer to the new pixrect. If it fails, it returns NULL, and displays an error on standard error.

If an existing secondary pixrect is provided in the call to the region operation, the result is another secondary pixrect referring to the underlying primary pixrect; there is no further connection between the two secondary pixrects. Generally, the distinction between primary and secondary pixrects is not important; however, no secondary pixrect should ever be used after its primary pixrect is destroyed.

### 2.2.3.3. Close / Destroy: Release a Pixrect's Resources

The following macros invoke device-dependent procedures to destroy a pixrect, freeing resources that belong to it:

```
#define pr_close(pr)
   struct      pixrect *pr;

#define pr_destroy(pr)
   struct      pixrect *pr;

#define prs_destroy(pr)
   struct      pixrect *pr;
```

The procedure returns 0 if successful, –1 if it fails. It may be applied to either primary or secondary pixrects. If a primary pixrect is destroyed before secondary pixrects which refer to its pixels, those secondary pixrects are invalidated; attempting any operation but *destroy* on them is an error. The three macros are identical; they are all defined for reasons of history and stylistic consistency.

## 2.2.4. Single-Pixel Operations

The next two operations are used to manipulate the value of a single pixel.

### 2.2.4.1. Get: Retrieve the Value of a Single Pixel

The following macros invoke device-dependent procedures to retrieve the value of a single pixel:

```
#define pr_get(pr, x, y)
   struct      pixrect *pr;
   int         x, y;

#define prs_get(srcprpos)
   struct      pr_prpos srcprpos;
```

*Pr* indicates the pixrect in which the pixel is to be found; *x* and *y* are the coordinates of the pixel. For *prs_get*, the same arguments are provided in the single struct *srcprpos*. The value of the pixel is returned as a 32-bit unsigned integer; if the procedure fails, it returns –1.

### 2.2.4.2. Put: Store a Value into a Single Pixel

The following macros invoke device-dependent procedures to store a value in a single pixel:

```
#define pr_put(pr, x, y, value)
   struct      pixrect *pr;
   int         x, y, value;

#define prs_put(dstprpos, value)
   struct      pr_prpos dstprpos;
   int         value;
```

*Pr* indicates the pixrect in which the pixel is to be found; *x* and *y* are the coordinates of the pixel. For *prs_put*, the same arguments are provided in the single struct *dstprpos*. *Value* is truncated on the left if necessary, and stored in the indicated pixel. If the procedure fails, it returns –1.

## 2.2.5. Constructing an Op Argument

The multi-pixel operations described in the next section all use a uniform mechanism for specifying the operation which is to produce destination pixel values. This operation is given in the *op* argument and includes several components.

Generally, *op* identifies a RasterOp. This is a logical function of two or three inputs; it computes the value of each pixel in the destination as a function of the previous value of that destination pixel, of a corresponding source pixel, and possibly a corresponding pixel in a mask.

Two other facilities are also specified in the *op* argument:

- a single, constant, source value may be specified as a *color* in *op*, and
- the *clipping* which is normally performed by every pixrect operation may be turned off by setting the PIX_DONTCLIP flag in the *op*.

We describe these three components of the *op* argument in order.

### 2.2.5.1. Specifying a RasterOp Function

Four bits of the *op* are used to specify one of the 16 distinct logical functions which combine monochrome source and destination pixels to give a monochrome result. This encoding is generalized to pixels of arbitrary depth by specifying that the function is applied to corresponding bits of the pixels in parallel. This emphasizes that the pixrects must be of the same depth. Some functions are much more common than others; the most useful are identified in the table *Useful Combinations of RasterOps*.

A convenient and intelligible form of encoding the function into four bits is supported by the following definitions:

```
#define PIX_SRC 0x18
#define PIX_DST 0x14
#define PIX_NOT(op)    (0x1E & (~op))
```

PIX_SRC and PIX_DST are defined constants, and PIX_NOT is a macro. Together, they allow a desired function to be specified by performing the corresponding logical operations on the appropriate constants. (The explicit definition of PIX_NOT is required to avoid inverting non-function bits of *op*).

A particular application of these logical operations allows definition of *set* and *clear* operations. The definition of the *set* operation that follows is always true, and hence sets the result:

```
#define PIX_SET (PIX_SRC | PIX_NOT(PIX_SRC))
```

The definition of the *clear* operation is always false, and hence clears the result:

```
#define PIX_CLR (PIX_SRC & PIX_NOT(PIX_SRC))
```

Other common RasterOp functions are defined in the following table:

Table 2-2: Useful Combinations of RasterOps

| Op with Value | Result |
|---|---|
| PIX_SRC | write (same as source argument) |
| PIX_DST | no-op (same as destination argument) |
| PIX_SRC \| PIX_DST | paint (OR of source and destination) |
| PIX_SRC & PIX_DST | mask (AND of source and destination) |
| PIX_NOT(PIX_SRC) & PIX_DST | erase (AND destination with negation of source) |
| PIX_NOT(PIX_DST) | invert area (negate the existing values) |
| PIX_SRC ^ PIX_DST | inverting paint (XOR of source and destination) |

## 2.2.5.2. Ops with a Constant Source Value

In certain cases, it is desirable to specify an infinite supply of pixels, all with the same value. This is done by using NULL for the source pixrect, and encoding a color in bits 5 - 31 of the *op* argument. The following macro supports this encoding:

```
#define PIX_COLOR(color)((color)<<5)
```

If no color is specified in an *op*, 0 appears by default; it remains necessary for the source pixrect specification to be NULL before this value is actually used.

Note that the color is not part of the *function* component of an *op* argument; it should never be part of an argument to PIX_NOT.

## 2.2.5.3. Controlling Clipping in the RasterOp

Pixrect operations normally clip to the bounds of the operand pixrects. Sometimes this can be done more efficiently by the client at a higher level. If the client can guarantee that only pixels which ought to be visible will be written, it may instruct the pixrect operation to bypass clipping checks, thus speeding their operation. This is done by setting the following flag in the *op* argument:

```
#define PIX_DONTCLIP    0x1
```

The result of a pixrect operation is undefined if PIX_DONTCLIP is set and the operation goes out of bounds.

Note that the PIX_DONTCLIP flag is not part of the *function* component of an *op* argument; it should never be part of an argument to PIX_NOT.

### 2.2.5.4. Examples of Complete Op Argument Specification

A very simple *op* argument will specify that source pixels be written to a destination, clipping as they go:

```
op = PIX_SRC;
```

A more complicated example will be used to affect a rectangle (known to be valid) with a constant red color defined elsewhere. (The function is syntactically correct; it's not clear how useful it is to XOR a constant source with the negation of the OR of the source and destination):

```
op = (PIX_SRC ^ PIX_NOT(PIX_SRC | PIX_DST) ) | PIX_COLOR(red) | PIX_DONTCLIP
```

## 2.2.6. Multi-Pixel Operations

The following operations all apply to multiple pixels at one time: *rop, stencil, replrop, batchrop,* and *vector*. With the exception of *vector*, they refer to rectangular areas of pixels. They all use a common mechanism, the *op* argument described in the previous section, to specify how pixels are to be set in the destination.

### 2.2.6.1. Rop: RasterOp Source to Destination

Device-dependent procedures invoked by the following macros perform the indicated raster operation from a source to a destination pixrect:

```
#define pr_rop(dpr, dx, dy, dw, dh, op, spr, sx, sy)
    struct      pixrect *dpr, *spr;
    int         dx, dy, dw, dh, op, sx, sy;

#define prs_rop(dstregion, op, srcprpos)
    struct      pr_subregion dstregion;
    int  op;
    struct      pr_prpos srcprpos;
```

*Dpr* addresses the destination pixrect, whose pixels will be affected; (*dx, dy*) is the origin (the upper-left pixel) of the affected rectangle; *dw* and *dh* are the width and height of that rectangle. *Spr* specifies the source pixrect, and (*sx, sy*) an origin within it. *Spr* may be NULL, to indicate a constant source specified in the *op* argument, as described above; in this case *sx* and *sy* are ignored. *Op* specifies the operation which is performed; its construction is described in preceding sections.

For *prs_rop*, the *dpr, dx, dy, dw* and *dh* arguments are all collected in a *pr_subregion* struct, defined above under *Geometry Structs*.

Raster operations are clipped to the source dimensions, if those are smaller than the destination size given. *Rop* procedures return –1 if they fail, 0 if they succeed.

Source and destination pixrects generally mus be the same depth. The only exception allows depth–1 pixrects to be sources to a destination of any depth. In this case, source pixels = 0 are interpreted as 0 and source pixels = 1 are written as the maximum value which can be stored in a destination pixel.

### 2.2.6.2. Stencil: RasterOps through a Mask

Device-dependent procedures invoked by the following macros perform the indicated raster operation from a source to a destination pixrect only in areas specified by a third (stencil) pixrect:

```
#define pr_stencil(dpr,dx,dy,dw,dh,op,stpr,stx,sty,spr,sx,sy)
    struct      pixrect *dpr, *stpr, *spr;
    int         dx,dy,dw,dh,op,stx,sty,sx,sy;

#define prs_stencil(dstregion, op, stenprpos, srcprpos)
    struct      pr_subregion dstregion;
    int         op;
    struct      pr_prpos stenprpos, srcprpos;
```

*Stencil* is identical to *rop* except that the source pixrect is written through a stencil pixrect which functions as a spatial write-enable mask. The stencil pixrect must have depth equal to 1. The indicated raster operation is applied only to destination pixels where the stencil pixrect is non-zero. Other destination pixels remain unchanged. The rectangle from (*sx,sy*) in the source pixrect *spr* is aligned with the rectangle from (*stx,sty*) in the stencil pixrect *stpr*, and written to the rectangle at (*dx,dy*) with width *dw* and height *dh* in the destination pixrect *dpr*. The source pixrect *spr* may be NULL, in which case the color specified in *op* is painted through the stencil. Clipping restricts painting to the intersection of the destination, stencil and source rectangles.

### 2.2.6.3. Replrop: Replicating the Source Pixrect

Often the source for a raster operation consists of a pattern that is used repeatedly, or replicated to cover an area. If a single value is to be written to all pixels in the destination, the best way is to specify that value in the *color* component of a *rop* operation. But when the pattern is larger than a single pixel, a mechanism is needed for specifying the basic pattern, and how it is to be laid down repeatedly on the destination. The *pr_replrop* procedure replicates a source pattern repeatedly to cover a destination area:

```
pr_replrop(dpr, dx, dy, dw, dh, op, spr, sx, sy)
    struct      pixrect *dpr, *spr;
    int         dx, dy, dw, dh, op, sx, sy;

#define prs_replrop(dsubreg, dp, sprpos)
    struct      pr_subregion dsubreg;
    struct      pr_prpos sprpos;
```

*Dpr* indicates the destination pixrect. The area affected is described by the rectangle defined by *dx, dy, dw, dh*. *Spr* indicates the source pixrect, and the origin within it is given by *sx, sy*. The corresponding *prs_replrop* macro generates a call to *pr_replrop*, expanding its *dsubreg* into the five destination arguments, and *sprpos* into the three source arguments. *Op* specifies the operation to be performed, as described above under *Constructing Op Arguments*.

The effect of *replrop* is the same as though an infinite pixrect were constructed using copies of the source pixrect laid immediately adjacent to each other in both dimensions, and then a *rop* was performed from that source to the destination. For instance, a standard gray pattern may be painted across a portion of the screen by constructing a pixrect that contains exactly one tile of the pattern, and by using it as the source pixrect.

The alignment of the pattern on the destination is controlled by the source origin given by *sx, sy*. If these values are 0, then the pattern will have its origin aligned with the position in the destination given by *dx, dy*. The most common other alignment is used to preserve a global alignment with the destination, for instance, to repair a portion of a gray. In this case, the source pixel which should be aligned with the destination position is the one which has the same coordinates as that destination pixel, *modulo* the size of the source pixrect. *Replrop* will perform this modulus operation for its clients, so it suffices in this case to simply copy the destination position (*dx, dy*) into the source position (*sx, sy*).

## 2.2.6.4. Batch RasterOp: Multiple Source to the Same Destination

Applications such as displaying text perform the same operation from a number of source pixrects to a single destination pixrect in a fashion that is amenable to global optimization. Device-dependent procedures invoked by the following macros perform raster operations on a sequence of sources to successive locations in a common destination pixrect:

```
struct batchitem {
    struct      pixrect *bi_pr;
    struct      pr_pos bi_pos;
};

#define pr_batchrop(dpr, dx, dy, op, items, n)
    struct      pixrect *dpr;
    int         dx, dy, op, n;
    struct      batchitem items[ ];

#define prs_batchrop(dstpos, op, items, n)
    struct      pr_prpos dstpos;
    int         op, n;
    struct      batchitem items [ ];
```

The sequence of sources used by a *batchrop* procedure is an array of *batchitem* structures. Each item specifies a source pixrect and an *advance* in *x* and *y*. The whole of each source pixrect is used, unless it needs to be clipped to fit the destination pixrect: the elements of *bi_pos* are used to update the destination position, not as an origin in the source pixrect.

*Batchrop* procedures take a destination, specified by *dpr, dx* and *dy*, or by *dstpos* in the case of *prs_batchrop*; an operation specified in *op*, as described in *Constructing Op Arguments* above, and an array of *batchitems* addressed by the argument *items*, and whose length is given in the argument *n*.

The destination position is initialized to the position given by *dx* and *dy*. Then, for each *batchitem*, the offsets given in *bi_pos* are added to the previous destination position, and the operation specified by *op* is performed on the source pixrect and the corresponding rectangle whose origin is at the current destination position. Note that the destination position is updated for each item in the batch, and these adjustments are cumulative.

The most common application of *batchrop* procedures is in painting text; additional facilities to support this application are described below under *Text Facilities for Pixrects*. Note that the definition of *batchrop* procedures supports variable-pitch and rotated fonts, and non-roman writing systems, as well as simpler text.

### 2.2.6.5. Vector: Draw a Straight Line

Device-dependent procedures invoked by the following macros draw a vector of unit width between two points in the indicated pixrect:

```
#define pr_vector(pr, x0, y0, x1, y1, op, value)
    struct      pixrect *pr;
    int         x0, y0, x1, y1, op, value;

#define prs_vector(pr, pos0, pos1, op, value)
    struct      pixrect *pr;
    struct      pr_pos      pos0, pos1;
    int         op, value;
```

*Vector* procedures draw a vector in the pixrect indicated by *pr*, with endpoints at (*x0, y0*) and (*x1, y1*), or at *pos0* and *pos1* in the case of *prs_vector*. Portions of the vector lying outside the pixrect are clipped as long as PIX_DONTCLIP is 0 in the *op* argument. The *op* argument is constructed as described above under *Constructing Op Arguments*; and *value* specifies the resulting value of pixels in the vector. If the color in *op* is non-zero, it takes precedence over the *value* argument.

## 2.2.7. Colormap Access

A *colormap* is a table which translates a pixel value into 8-bit intensities in red, green, and blue. For a pixrect of depth *n*, the corresponding colormap will have $2^n$ entries. The two most common cases are depth-1 (monochrome with two entries) and depth-8 (with 256 entries). Memory pixrects do not have colormaps.

### 2.2.7.1. Get Colormap

The following macros invoke device-dependent procedures to read all or part of a colormap into arrays in memory:

```
#define pr_getcolormap(pr, index, count, red, green, blue)
    struct              pixrect *pr;
    int                 index, count;
    unsigned char       red [ ], green[ ], blue[ ];

#define prs_getcolormap(pr, index, count, red, green, blue)
    struct              pixrect *pr;
    int                 index, count;
    unsigned char       red [ ], green[ ], blue[ ];
```

These two macros have identical definitions; both are defined to allow consistent use of one set of names for all operations.

*Pr* identifies a pixrect whose colormap is to be read; the *count* entries starting at *index* are read into the three arrays.

For monochrome pixrects the same value is written to corresponding elements of the *red, green* and *blue* arrays. These array elements will have their bits either all cleared, indicating *black*, or all set, indicating *white*. By default, the 0th (*background*) element is white, and the 1st ((*foreground*) element is black.

### 2.2.7.2. Put Colormap

The following macros invoke device-dependent procedures to store from memory into all or part of a colormap:

```
#define pr_putcolormap(pr, index, count, red, green, blue)
    struct          pixrect *pr;
    int             index, count;
    unsigned char   red [ ], green[ ], blue[ ];

#define prs_putcolormap(pr, index, count, red, green, blue)
    struct          pixrect *pr;
    int             index, count;
    unsigned char   red [ ], green[ ], blue[ ];
```

These two macros have identical definitions; both are defined to allow consistent use of one set of names for all operations.

The *count* elements starting at *index* (zero origin) in the colormap for the pixrect identified by *pr* are loaded from corresponding elements of the three arrays.

For monochrome pixrects, the only value considered is *red*[0]. If this value is 0, then the pixrect will be set to a dark background and light foreground. If the value is non-zero, the foreground will be dark, e.g. black-on-white. Monochrome pixrects are dark-on-light by default.

*Note:* Full functionality of the colormap is not supported for depth–1 pixrects. Colormap changes to depth–1 pixrects apply only to subsequent operations whereas a colormap change to a color device instantly changes all affected pixels on the display pixrect.

### 2.2.7.3. Provision for Inverted Video Pixrects

Video inversion is accomplished by manipulation of the colormap of a pixrect. The colormap of a depth–1 pixrect has two elements. The following procedures provide video inversion control:

```
pr_blackonwhite(pr, min, max)
    struct      pixrect *pr;
    int         min, max;

pr_whiteonblack(pr, min, max)
    struct      pixrect *pr;
    int         min, max;

pr_reversevideo(pr, min, max)
    struct      pixrect *pr;
    int         min, max;
```

In each procedure, *pr* identifies the pixrect to be affected; *min* is the lowest index in the colormap, specifying the background color, and *max* is the highest index, specifying the foreground color. These will most often be 0 and 1 for monochrome pixrects; the more general definitions allow colormap-sharing schemes, such as the one described below in *Colormap Sharing*, in the chapter *Overlapped Windows: Imaging Facilities*.

"Black-on-white" means that zero (background) pixels will be painted at full intensity, which is usually white. *Pr_blackonwhite* sets all bits in the entry for colormap location *min* and clears all

bits in colormap location *max*.

"White-on-black" means that zero (background) pixels will be painted at minimum intensity, which is usually black. *Pr_whiteonblack* clears all bits in colormap location *min* and sets all bits in the entry for colormap location *max*.

Reversevideo exchanges the *min* and *max* color intensities.

These procedures are ignored for memory pixrects.

## 2.2.8. Attributes for Bitplane Control

In a color pixrect, it is often useful to define bitplanes which may be manipulated independently; operations on one plane leave the other planes of an image unaffected. This is normally done by assigning a plane to a constant bit position in each pixel. Thus, the value of the *i*th bit in all the pixels defines the *i*th bitplane in the image. It is sometimes beneficial to restrict pixrect operations to affect a subset of a pixrect's bitplanes. This is done with a bitplane mask. A bitplane mask value is stored in the pixrect's private data and may be accessed by the attribute operations.

### 2.2.8.1. Get Attributes

Device-dependent procedures invoked by the following macros retrieve the mask which controls which planes in a pixrect are affected by other pixrect operations:

```
#define pr_getattributes(pr, planes)
    struct      pixrect *pr;
    int         *planes;

#define prs_getattributes(pr, planes)
    struct      pixrect *pr;
    int         *planes;
```

*Pr* identifies the pixrect; its current bitplanes mask is stored into the word addressed by *planes*. If *planes* is NULL, no operation is performed.

The two macros are identically defined; both are provided to allow consistent use of the same style of names.

### 2.2.8.2. Put Attributes

Device-dependent procedures invoked by the following macro manipulate a mask which controls which planes in a pixrect are affected by other pixrect operations:

```
#define pr_putattributes(pr, planes)
    struct      pixrect *pr;
    int         *planes;

#define prs_putattributes(pr, planes)
    struct      pixrect *pr;
    int         *planes;
```

*Pr* identifies the pixrect to be affected; its mask is set so that only the planes identified by 1-bits in the value of *planes* will be read or written by subsequent pixrect operations. If *planes* is NULL, no operation is performed.

The two macros are identically defined; both are provided to allow consistent use of the same style of names.

*Planes* may be used to enforce that no pixel values outside of a pixrects colormap section are written. In other words, the *planes* argument is a bitplane write-enable mask. Only those bits of the pixel corresponding to a 1 in the same bit position of **planes* will be affected by pixrect operations. For example, if **planes* = 1 in a destination pixrect, subsequent operations will only modify bit 0 of the destination pixels.

*Note:* If any *planes* are masked off by a call to *pr_putattributes*, no further read or write access to those planes is possible until a subsequent call to *pr_putattributes* unmasks them.

## 2.3. Text Facilities for Pixrects

Displaying text is an important task in many applications, so pixrect-level facilities are provided to address it directly. These facilities fall into two main categories: a standard format for describing fonts and character images, with routines for processing them; and a set of routines which take a string of text and a font, and handle various parts of painting that string in a pixrect.

### 2.3.1. Pixfonts and Pixchars

The following two structs are used to describe fonts and character images for pixrect-level text facilities:

```
struct pixchar {
    struct      pixrect *pc_pr;
    struct      pr_pos pc_home;
    struct      pr_pos pc_adv;
};

struct pixfont {
    struct      pr_size pf_defaultsize;
    struct      pixchar pf_char[256];
};
```

A *pixfont* contains an array of *pixchars*, indexed by the character code; it also contains the size (in pixels) of its characters when they are all the same. (If the size of a font's characters varies in one dimension, that value in *pf_defaultsize* will not have anything useful in it; however, the other may still be useful. Thus, for non-rotated variable-pitch fonts, *pf_defaultsize.y* will still indicate the unleaded interline spacing for that font.)

*Note:* The definition of a *pixfont* is expected to change.

The *pixchar* defines the format of a single character in a font. The actual image of the character is stored in a pixrect (a separate pixrect for each character) addressed by *pc_pr*. Characters that do not have a displayable image will have NULL in their entry in *pc_pr*. *Pc_home* is the origin of that image (its upper left corner) relative to the character origin. Characters are normally placed relative to a *baseline*, which is the lowest point on characters without descenders.

The leftmost point on a character is normally its origin, but kerning or mandatory letter spacing may move the origin right or left of that point. *Pc_adv* is the amount the destination position is changed by this character; that is, the amounts in *pc_adv* added to the current origin will give the origin for the next character. While normal text only advances horizontally, rotated fonts may have a vertical advance. Both are provided for in the font.

## 2.3.2. Operations on Pixfonts

Before a process may use a font, it must ensure that font has been loaded into virtual memory; this is done with *pf_open*:

```
struct pixfont *pf_open(name)
    char        *name;
```

This procedure opens the file with the given *name*. The file should be a font file as described in *vfont*(5): The file is converted to pixfont format, allocating memory for its associated structs and reading in the data for it from disk. A NULL is returned if the font cannot be opened.

The procedure:

```
struct pixfont *pf_default()
```

performs the same function for the system default font, normally a fixed-pitch, 16-point sans serif font with upper-case letters 12 pixels high. If the environment parameter DEFAULT_FONT is set, its value will be taken as the name of the font file to be opened by *pf_default*.

*Note*: *pf_open* and *pf_default* load a new copy of the font every time they are called, even if the font has already been loaded. To conserve memory, clients may use *pw_pfsysopen*, described in *Overlapped Windows: Imaging Facilities*, or take care only to open a font once in a process.

When a process is finished with a font, it should call *pf_close* to free the memory associated with it:

```
pf_close(pf)
    struct      pixfont *pf;
```

*Pf* should be the font handle returned by a previous call to *pf_open* or *pf_default*.

## 2.3.3. Pixrect Text Display

Characters are written into a pixrect with the *pf_text* procedure:

```
pf_text(where, op, font, text)
    struct      pr_prpos where;
    int         op;
    struct      pixfont *font;
    char        *text;
```

*Where* is the destination for the start of the text (nominal left edge, baseline; see *Pixfonts*); *op* is the raster operation to be used in writing the text, as described in *Constructing Op Arguments*; *font* is a pointer to the font in which the text is to be displayed; and *text* is the actual null-terminated string to be displayed.

The following procedure paints "transparent" text: it doesn't disturbing destination pixels in blank areas of the character's image:

```
pf_ttext(where, op, font, text)
    struct      pr_prpos where;
    int         op;
    struct      pixfont *font;
    char        *text;
```

The arguments to this procedure are the same as for *pf_text*. The characters' bitmaps are used as a stencil, and the color specified in op is squirted through the stencil.

(For monochrome pixrects, the same effect can be achieved by using PIX_SRC | PIX_DST as the function in the *op*; this procedure is required for color pixrects.)

Auxiliary procedures used with *pf_text* include:

```
struct pr_size pf_textbatch(where,lengthp, font, text)
    struct      batchitem where[];
    int         *lengthp;
    struct      pixfont *font;
    char        *text;

struct pr_size pf_textwidth(len, font, text)
    int         len;
    struct      pixfont *font;
    char        *text;
```

*Pf_textbatch* is used internally by *pf_text*; it constructs an array of batchitems and records its length, as required by batchrop (see *Batch Raster Op*). *Where* should be the address of an array to be filled in, and *lengthp* should point to a maximum length for that array. *Text* addresses the null-terminated string to be put in the batch, and *font* the pixfont to be used to display it. On its return, **lengthp* will have been modified to be the number of batchitems actually used for *text*.

*Pf_textwidth* returns a *pr_size* which contains the total dimension of the string of the first *len* characters in text, when formatted in the indicated font.

## 2.4. Memory Pixrects

Pixrects which store their pixels in memory, rather than displaying them on some display, have several special properties. Like all other pixrects, their dimensions are visible in the *pr_size* and *pr_depth* elements of their pixrect struct, and the device-dependent operations appropriate to manipulating them are available through their *pr_ops*. Beyond this, however, the format of the data which describes the particular pixrect is also public: *pr_data* will hold the address of a *mpr_data* struct, described below. There is also a public procedure, *mem_create*, which dynamically allocates a new memory pixrect, and a macro, *mpr_static*, which can be used to generate an initialized memory pixrect in the code of a client program. Thus, a client may construct and manipulate memory pixrects using non-pixrect operations.

### 2.4.1. The Mpr_data Struct

The *pr_data* element of a memory pixrect points to an *mpr_data* struct, which contains the information needed to deal with a memory pixrect:

```
struct mpr_data {
    int         md_linebytes;
    short       *md_image;
    struct      pr_pos md_offset;
    short       md_primary;
    short       md_flags;
};
#define MP_DISPLAY
#define MP_REVERSEVIDEO
```

*Linebytes* is the number of bytes stored in a row of the primary pixrect. This is the difference in the addresses between two pixels at the same *x*-coordinate, one row apart. Because a secondary pixrect may not include the full width of its primary pixrect, this quantity cannot be computed from the width of the pixrect — see *Region*. The actual pixels of a memory pixrect are stored someplace else in memory, usually an array, which *md_image* points to; the format of that area is described in the next section. The creator of the memory pixrect must ensure that *md_image* contains an even address. *Md_offset* is the *x-y* position of the first pixel of this pixrect in the array of pixels addressed by *md_image*. *Md_primary* is 1 if the pixrect is primary and had its image allocated dynamically (e.g. by *mem_create*). In this case, *md_image* will point to an area not referenced by any other primary pixrect. This flag is interrogated by the *destroy* routine: if it is 1 when that routine is called, the pixrect's image memory will be freed.

(*Md_flags* & MP_DISPLAY) is non-zero if this memory pixrect is in fact a display device. Otherwise, it is 0. (*Md_flags* & MP_REVERSEVIDEO) is 1 if *reversevideo* is enabled for the display device. *Md_flags* is present to support memory-mapped display devices like the Sun–2 black-and-white video device.

## 2.4.2. Pixel Layout in Memory Pixrects

In memory, the upper-left corner pixel is stored at the lowest address. This address should be even. That first pixel is followed by the remaining pixels in the top row, left-to-right. Pixels are stored in successive bits without padding or alignment. For pixels more than 1 bit deep, it is possible for a pixel to cross a byte boundary. However, rows are rounded up to 16-bit boundaries. After any padding for the top row, pixels for the row below are stored, and so on through the whole rectangle.

## 2.4.3. Creating Memory Pixrects

### 2.4.3.1. Mem_create

A new primary pixrect is created by a call on the procedure *mem_create*:

```
struct      pixrect *mem_create(w, h, depth)
    int     w, h, depth;
```

*W, h,* and *depth* specify the width and height in pixels, and depth in bits, of the new pixrect. Sufficient memory to hold those pixels is allocated and cleared to 0, new *mpr_data* and *pixrect* structs are allocated and initialized, and a pointer to the pixrect is returned. If this can not be done, the return value is NULL.

### 2.4.3.2. Static Memory Pixrects

A memory pixrect may be created at compile time by using the *mpr_static* macro:

```
#define mpr_static(name, w, h, d, image)
    char        *name;
    int         w, h, d;
    short       *image;
```

where *name* is a token to identify the generated data objects; *w*, *h*, and *d* are the width and height in pixels, and depth in bits of the pixrect; and *image* is the address of an even-byte aligned data object that contains the pixel values in the format described above.

The macro generates two structs:

```
struct      mpr_data name_data ;
struct      pixrect name ;
```

The *mpr_data* is initialized to point to all of the image data passed in; the pixrect then refers to *mem_ops* and to *name*_data.

*Note:* Contrary to its name, this macro generates structs whose storage class is *extern*.

# Chapter 3

# Overlapped Windows: Imaging Facilities

This chapter and the following two deal with the *sunwindow* layer of the window system, which provides facilities for managing windows with overlap and concurrency. This chapter is specifically concerned with generating images in such an environment. Chapter 4 deals with control of the windows, manipulating their size, location, and other structural characteristics. Chapter 5 describes the facilities for serializing multiple input streams and distributing them appropriately to multiple windows. The term "sunwindow layer" comes from the name of the library that contains its implementation.

At this level of the system, a window is treated as a *device*: it is named by an entry in the */dev* directory; it is accessed by the *open*(2) system call; and the usual handle on the window is the *file descriptor* (or *fd*) returned from that call.

For this chapter, however, a window may be considered as simply a rectangular area with contents maintained by some process. Multiple windows, maintained by independent processes, may coexist on the same screen; SunWindows allows them to *overlap*, sharing the same (*x*, *y*) coordinates, and proceeding concurrently, while maintaining their separate identities.

Window system facilities may also be used to construct a non-overlapped environment; the window system facilities required are much the same as for constructing on overlapping environment.

## 3.1. Window Issues: Controlled Display Generation

Multiple windows on a display introduce two new issues, which may be broadly characterized as: 1) preventing the window from painting where or when it shouldn't, and 2) ensuring that it does paint whenever and wherever it should. The first includes *clipping* and *locking*; the latter covers *damage repair* and *fixups*.

### 3.1.1. Clipping and Locking

*Clipping* constrains a window to draw only within the boundaries of its portion of the screen. This area is subject to changes beyond the control of a window's process — another window may be opened on top of the first, covering part of its contents, or a window may be shrunk to make room for another alongside it. Thus, it is convenient for the window system to maintain up-to-date information on which portions of the screen belong to which windows, and for the windows to consult that information whenever they are about to draw on the screen.

*Locking* prevents window processes from interfering with each other in several ways:

- Raster hardware may require several operations to complete a change to the display; one process' use of the hardware should be protected from interference by others during this critical interval.
- Changes to the arrangement of windows must be prevented while a process is painting, lest an area be removed from a window as it is being painted.
- A software cursor that the window process does not control (the kernel is usually responsible for the cursor) may have to be removed so that it does not interfere with the window's image.

Clipping and locking are described in more detail in *Locking and Clipping.*

### 3.1.2. Damage Repair and Fixups

A window whose image does not appear entirely as it should on the screen is said to be *damaged.* A common cause of damage is being first overlaid, and then uncovered, by another window. When a window is damaged, a portion of the window's image must be *repaired.* Note that the requirement for repairing damage may arise at any time; it is completely outside the window's control.

When a process performs some operation which includes reading a portion of its window, for instance copying a part of the image from one region to another to implement scrolling, it may find the source pixels obscured. This necessitates a *fixup,* in which that portion of the image is regenerated, similar to repairing damage. Unlike damage generation, the need to do some fixup is provoked only in response to an action of the window's process, e.g., scrolling.

### 3.1.3. Retained Windows

Either form of regeneration may be done by recomputing the image; this approach is reasonable for applications like text where there is some underlying representation from which the display can be recomputed easily. For images which require considerable computation, SunWindows provides a *retained* window, whose image is maintained in memory as well as on the display. Such a window may have its image recopied to the display as needed to repair damage. The mechanism for making a window *retained* is described in *Pixwins.*

### 3.1.4. Colormap Sharing

On color displays, colormap entries are a constrained resource. When shared among multiple applications, colormap usage requires arbitration. For example, consider the following applications running on the same display at the same time in different windows:

- Application program X needs 64 colors for rendering VLSI images.
- Application program Y needs 32 shades of gray for rendering black and white photographs.
- Application program Z needs 256 colors (assume this is the entire colormap) for rendering full color photographs.

Colormap usage control is handled as follows:

- To determine how X and Y figure out what portion of the colormap they should use so they don't access each others' entries, SunWindows provides a resource manager that allocates a *colormap segment* to each window from the *shared colormap.* To reduce duplicate colormap segments, they are named and can be shared among cooperating processes.

- To hide concerns about knowing the correct offset to the start of a colormap segment from routines that access the image, SunWindow initializes the image of a window with the colormap segment offset. This effectively hides the offset from the application.
- To accommodate Z if its large colormap segment request cannot be granted, Z's colormap is loaded into the hardware, replacing the shared colormap, whenever input is directed towards Z's window. Z's request is not denied even though it is not allocated its own segment in the shared colormap.
- To control the blanking that occurs when colormap swapping causes all but Z's image to disappear. Given an unfortunate choice of colors, SunWindow ensures that the background (colormap segment entry 0) and foreground (colormap segment entry size–1) for all segments in shared colormap are the same. This colormap content restriction has the affect of eliminating blanking.

### 3.1.5. Process Structure

In SunWindows, access to the screen is performed in each user process, instead of in a single, central, fully debugged screen management process. This increases the possibility of an incorrect user process damaging the display area of other application processes. Several compensating factors justify this approach:

- Clients may access this open system at whichever level is most convenient. Clients who require the ultimate efficiency of direct screen access need not sacrifice the window management functions of the window system.
- Leaving processing in user processes promotes efficiency in both implementation and execution: making and testing extensions and modifications is much easier in user code than in the kernel.

### 3.1.6. Imaging with Windows

A detailed discussion of imaging with windows follows. We begin with a description of the basic data structures that are used in this level of Sunwindows. These are a primitive geometric facility, the *rect*, for describing rectangles, and the basic structure, the *pixwin*, that describes a window on the screen with its associated state and operation vectors.

Following is a brief discussion of the simple process of creating and destroying *pixwins*. This is followed by a detailed description of the approach to locking and clipping, which leads naturally into a discussion of library routines that access a *pixwin's* pixels. Detecting and repairing damage is treated next.

### 3.1.7. Libraries and Header Files

The procedures described in this chapter are provided in the *sunwindow* library (*/usr/lib/libsunwindow.a*). The header file */usr/include/sunwindow/window_hs.h* contains all the *includes* that are required by a program using the facilities described in this chapter.

## 3.2. Data Structures

Here are some data structures used in the implementation of pixwins. Be sure you understand *rects* before proceding. Descriptions of the data structure internals are also provided for additional information.

### 3.2.1. Rects

Throughout Sunwindows, images are dealt with in rectangular chunks; where complex shapes are required, they are built up out of groups of rectangles. The basic description of a rectangle is the *rect* struct, defined in the header file */usr/include/sunwindow/rect.h*. The same file contains definitions of several useful macros and procedures for dealing with *rects*.

Where a window is partially obscured, its visible portion generally cannot be described by a simple rectangle; instead a list of non-overlapping rectangular fragments which together cover the visible area is used. This *rectlist* is declared, along with its associated macros and procedures in the file */usr/include/sunwindow/rectlist.h*.

At this point we only discuss the *rect* struct and its most useful macros; a full description of both *rects* and *rectlists* is in Appendix A.

```
#define coord short

struct rect {
    coord       r_left;
    coord       r_top;
    short       r_width,
    short       r_height;
};
```

In the context of a window, the rectangle lies in a coordinate system whose origin is in the upper left-hand corner, and whose dimensions are given in pixels. Two macros determine an edge not given explicitly in the *rect*. These macros are:

```
#define rect_right(rp)
#define rect_bottom(rp)

struct rect *rp;
```

These macros return the coordinate of the last pixel within the rectangle on the right or bottom, respectively.

### 3.2.2. Pixwins

*Pixwins* are the basic imaging elements of the overlapped window system. The window layer of the system uses pixwins to represent pixrects on a window surface. The pixwin thus describes the window image and a set of routines to operate on the window.

A client of the window system has a rectangular window in which it displays information for the user. Because of overlapping, however, it is not always possible to display information in all parts of a client's window. Parts of an image may have to be displayed at some point long after they were generated, as a portion of the window is uncovered. The clipping and repainting necessary to preserve the identity of the rectangular image across interference with other objects on the screen is handled by manipulations on pixwins.

The pixwin struct is defined in */usr/include/sunwindow/pixwin.h*:

```
struct pixwin {
    struct      pixrectops *pw_ops;
    caddr_t     pw_opshandle;
    int         pw_opsx;
    int         pw_opsy;
    struct      rectlist pw_fixup;
    struct      pixrect *pw_pixrect;
    struct      pixrect *pw_prretained;
    struct      pixwin_clipops *pw_clipops;
    struct      pixwin_clipdata *pw_clipdata;
    char        pw_cmsname[20];
};
```

The *pixwin* refers to a portion of some device, typically a display; the device is identified by *pw_pixrect.*

If the image displayed in the *pixwin* required a large effort to compute, it will be worth saving a backup copy of the whole image, making the window a *retained* window. This is done by creating an appropriate *memory pixrect* as described in *Memory Pixrects*, and storing a pointer to it in *pw_prretained.*

Portions of the image which could not be accessed by an operation which attempted to read pixels from the *pixwin* are indicated by *pw_fixup.*

*Pw_ops* is a pointer to a vector of operations in screen access macros to call either the *pixwin* software level or as an optimization, the *pixrect* software directly. The structure *pixrectops* was discussed in *Pixrectops.* The *pw_opshandle* is the data handle passed to the operations of *pw_ops. Pw_opsx* and *pw_opsy* are additional offset information that screen access macros use. These three fields are dynamically altered based on locking and clipping status.

*Pw_clipdata* is a collection of information of special interest to locking and clipping. *Pw_clipops* points to a vector of operations which are used in locking and clipping. The declarations of these last two structs are discussed more fully in *Pixwin_clipdata Struct, Pixwin_clipops Struct,* and subsequent sections.

*Pw_cmsname* the identifier of the colormap segment that this pixwin is currently using. This value should only be accessed via *pw_setcmsname* and *pw_getcmsname* procedures described below.

### 3.2.3. Pixwin_clipdata Struct

```
struct pixwin_clipdata {
    int         pwcd_windowfd;
    short       pwcd_state;
    struct      rectlist pwcd_clipping;
    int         pwcd_clipid;
    int         pwcd_damagedid;
    int         pwcd_lockcount;
    struct      pixrect *pwcd_prmulti;
    struct      pixrect *pwcd_prsingle;
    struct      pixwin_prlist *pwcd_prl;
    struct      rectlist pwcd_clippingsorted[RECTS_SORTS];
    struct      rect *pwcd_regionrect;
};

#define PWCD_NULL           0
#define PWCD_MULTIRECTS     1
#define PWCD_SINGLERECT     2
#define PWCD_USERDEFINE     3

struct pixwin_prlist {
    struct      pixwin_prlist *prl_next;
    struct      pixrect *prl_pixrect;
    int         prl_x, prl_y;
};
```

*Pwcd_windowfd* is a file descriptor for the window being accessed. Within the owning process, it is the standard handle on a window. A description of the interplay between windows and *pixwins* continues in *Pixwin Creation and Destruction.* The portions of the window's area accessible through the *pixwin* are described by the *pwcd_clipping rectlist. Pwcd_regionrect,* if not NULL, points to a *rect* that is intersected with *pwcd_clipping* to further restrict the portions of the window's area accessible through the *pixwin. Pwcd_clipid* and *pwcd_damagedid* identify the most recent *rectlists* retrieved for a window. *Pwcd_lockcount* is a reference count used for nested locking, as described in *Locking* below. Copies of this *pwcd_clipping,* sorted in directions convenient for copy operations, are stored in *pwcd_clippingsorted.*

*Pwcd_state* can be one of the following:

Table 3-1: Clipping State

| State | Meaning |
|---|---|
| PWCD_NULL | no part of window visible |
| PWCD_MULTIRECTS | must clip to multiple rectangles |
| PWCD_SINGLERECT | need clip to only one rectangle |
| PWCD_USERDEFINE | the client program will be responsible for setting up the clipping |

*Pwcd_prmulti* is the *pixrect* for drawing when there are multiple rectangles involved in the clipping. *Pwcd_prsingle* is the *pixrect* for clipping when there is only one rectangle visible.

*Pwcd_prl* is a list of *pixrects* that may be used for clipping when there are multiple rectangles involved. For vector drawing, these clippers *must* be used maintain stepping integrity across abutting rectangle boundaries. The *prl_x* and *prl_y* fields in the *pixwin_prlist* structure are offsets from the window origin for the associated *prl_pixrect.*

## 3.2.4. Pixwin_clipops Struct

```
struct pixwin_clipops {
    int         (*pwco_lock)(),
    int         (*pwco_unlock)(),
    int         (*pwco_reset)(),
    int         (*pwco_getclipping)();
};
```

The *pw_clipops* struct is a vector of pointers to system-provided procedures that implement correct screen access. These are accessed through macros described in *Locking and Clipping.*

## 3.3. Pixwin Creation and Destruction

To create a *pixwin*, the window to which it will refer must already exist. This task is accomplished with procedures like *win_getnewwindow* and *win_setrect*, described in *Window Manipulation*, or, at a higher level, *tool_create* and *tool_createsubwindow*, described in *Suntool: Tools and Subwindows.* The *pixwin* is then created for that window by a call to *pw_open*:

```
struct pixwin *pw_open(fd)
    int         fd;
```

*Pw_open* takes a file descriptor for the window on which the *pixwin* is to write. A pointer to a *pixwin* struct is returned. At this point the *pixwin* describes the exposed area of the window. If the client wants a *retained pixwin, pw_prretained* should be set to point to an appropriately-sized memory *pixrect* after *pw_open* returns.

When a client is finished with a window, it should be released by a call to:

```
pw_close(pw)
    struct      pixwin *pw;
```

*Pw_close* frees any dynamic storage associated with the *pixwin*, including its *pw_prretained* pixrect if any. If the *pixwin* has a lock on the screen, it is released.

### 3.3.1. Region Creation

One can use pixwins to clip rectangular regions within a window's own rectangular area. The *region* operation creates a new pixwin that refers to an area within an existing one:

```
struct pixwin *pw_region(pw, x, y, w, h)
    struct      pixwin *pw;
    int         x, y, w, h;
```

The pixwin which is to serve as the source is addressed by *pw*; *x, y, w* and *h* describe the rectangle to be included in the new pixwin. The upper left pixel in the returned pixwin is at coordinates (0,0); this pixel has coordinates (*x, y*) in the source pixwin.

## 3.4. Locking and Clipping

Before a window process reads from or writes to the screen, it must satisfy several conditions:

- It should obtain exclusive use of the display hardware,
- The position of windows on the screen should be frozen,
- The window's description of what portions of its window are visible should be up-to-date, and
- The window should confine its activities to those visible areas.

The first three of these requirements is met by *locking*; the last amounts to *clipping* the image the window will write to the bounds of its *exposed* area. All are handled implicitly by the access routines described in *Accessing a Pixwin's Pixels*. Some clients will use those routines, but for efficiency's sake, lock explicitly around a body of screen access operations.

### 3.4.1. Locking

The *pw_lock* macro:

```
pw_lock(pw, r)
    struct      pixwin *pw;
    struct      rect *r;
```

uses the lock routine pointed to by the window's *pw_clipops* to acquire a lock for the user process that made this call. *Pw* addresses the *pixwin* to be used for the ouput; *r* is the rectangle in the window's coordinate system that bounds the area to be affected. *Pw_lock* blocks if the lock is unavailable, for example, if another process currently has the display locked.

Lock operations for a single *pixwin* may be nested; inner lock operations merely increment a count of locks outstanding, *pwcd_lockcount* in the window's *pw_clipdata* struct. Their affected rectangles must lie within the original lock's.

A similar macro is:

```
pw_unlock(pw)
   struct      pixwin *pw;
```

which decrements the lock count. If this brings it to 0, the lock is actually released.

Since locks may be nested, it is possible for a client procedure to find itself especially in error handling with a lock which may require an indefinite number of *unlock*s. To handle this situation cleanly, another routine is provided. The following macro sets *pw*'s lockcount to 0 and release its lock:

```
pw_reset(pw)
   struct      pixwin *pw;
```

Like *pw_lock* and *pw_unlock*, *pw_lock* calls a routine addressed in the *pixwin*'s *pixwin_clipops* struct, in this case the one addressed by *pwco_reset*.

Acquisition of a lock has the following effects:

- If the cursor is in conflict with the affected rectangle, it is removed from the screen. While the screen is locked, the cursor will not be moved in such a way as to disrupt any screen accessing.
- Access to the display is restricted to the process acquiring the lock.
- Modification of the database that describes the positions of all the windows on the screen is prevented.
- The id of the most recent clipping information for the window is retrieved, and compared with that stored in *pwcd_clipid* in the window's *pw_clipdata*. If they differ, the routine addressed by *pwco_getclipping* is invoked, to make all the fields in *pw_clipdata* accurately describe the area which may be written into.
- Once the correct clipping is in hand, the *pwcd_state* variable's value determines how to set *pw_ops*, *pw_opshandle*, *pw_opsz* and *pw_opsy*. This setting is done in anticipation of further screen access operations being done before a subsequent unlock. These values can often be set to bypass the *pixwin* software by going directly to the *pixrect* level.

Locking is both moderately expensive as it involves two system calls, and capable of impacting other processes. Clients with a recognizable group of screen updates to do can gain noticeably by surrounding the group with lock – unlock brackets; then the locking overhead will only be incurred once. An example of such a group might be a line of text, or a series of vectors which have all been computed.

While it has the screen locked, a process should *not*:

- do any significant computation unrelated to displaying its image;
- invoke any system calls, including other I/O, which might cause it to block; or
- invoke any pixwin calls except *pw_unlock* and those described in *Accessing a Pixwin's Pixels*. In any case, the lock should not be held longer than about a quarter of a second, even following all these guidelines.

As a deadlock resolution approach, when a display lock is held for more than 10 seconds, the lock is broken. However, the offending process is not notified by signal; the idea is that a process shouldn't be aborted for this infraction. A message is displayed on the console.

## 3.4.2. Clipping

Output to a window is clipped to the window's *pwcd_clipping rectlist*; this is a series of rectangles which, taken together, cover the valid area that this window may write to. There are two routines which set the *pixwin*'s clipping:

```
pw_exposed(pw)
   struct      pixwin *pw;

pw_damaged(pw)
   struct      pixwin *pw;
```

*Pw_damaged* is discussed in *Damage*. *Pw_exposed* is the normal routine for discovering what portion of a window is visible. It retrieves the *rectlist* describing that area into the *pixwin*'s *pwcd_clipping*, and stores the id identifying it in *pwcd_clipid*. It also stores its own address in the *pixwin*'s *pwco_getclipping*, so that subsequent lock operations will get the correct area description.

Clipping, even more than locking, should normally be left to the library output routines. For the intrepid, the strategy these routines follow is briefly sketched here; the *rectlist* data structures and procedures in Appendix A are required reading.

Some procedure will set the *pixwin*'s *pwcd_clipping* so that it contains a *rectlist* describing the region which may be painted. This is done by a lock operation which makes a call through **pwco_getclipping*, or an explicit call to one of *pw_open*, *pw_donedamaged*, *pw_exposed* or *pw_damaged*. This *rectlist* is essentially a list of rectangular fragments which together cover the area of interest. As an image is generated, portions of it which lie outside the rectangle list must be masked off, and the remainder written to the window through a *pixrect*.

The clipping aid *pwcd_prmulti* is set up to be a *pixrect* which clips for the entire rectangular area of the window. Any clipping using this *pixrect* must utilize the information in *pwcd_clipping* to do the actual clipping to multiple rectangles.

*Pwcd_prl* is set up to parallel each of the rectangles in *pwcd_clipping*. Thus, if one draws to each of the *pixrects* in this data structure, the image will be correctly clipped. *Pwcd_state* is set by examining the makeup of the *pwcd_clipping*. If *pwcd_state* is PWCD_SINGLERECT, a *pixrect* is set up in *pwcd_prsingle* also. When this case exists, after *pw_lock* and before *pw_unlock*, most screen accesses will directly access the *pixrect* level of software. Thus, in this common case, screen access is as fast in the window system as it is on the raw *pixrect* software outside of the window system. Also, *pwcd_prsingle* is set up with a zero height and width *pixrect* when *pwcd_state* is PWCD_NULL.

As an escape, none of the *pixrect* setup described above takes place when *pwcd_state* is PWCD_USERDEFINE. This means that clipping is the responsibility of higher level software.

A client may write to the display with an operation which specifies no clipping (op | PIX_DONTCLIP). This means that it is doing the clipping at a higher level. Note that clipping data is only valid during the time the client may write to the screen, that is when the window's owner process holds a lock on the screen. If the clipping is done wrong, it is possible to damage another window's image.

## 3.5. Accessing a Pixwin's Pixels

Procedures described in this section provide all the normal facilities for output to a window and should be used unless there are special circumstances. Each contains a call to the standard lock procedure, described in *Locking*. Each takes care of clipping to the *rectlist* in *pw_clipping*. Since the routines are used both for painting new material in a window and for repairing damage, they make no assumption about what clipping information should be gotten. Thus, there should be some previous call to either *pw_open*, *pw_donedamaged*, *pw_exposed* or *pw_damaged*, to initialize *pwo_getclipping* correctly.

The procedures described in this section will maintain the memory *pixrect* for a retained pixwin. That is, they check the window's *pw_prretained*, and if it is not null, perform their operation on that data in memory, as well as on the screen.

### 3.5.1. Write Routines

```
pw_write(pw, xd, yd, width, height, op, pr, xs, ys)
    struct      pixwin *pw;
    int         op, xd, yd, width, height, xs, ys;
    struct      pixrect *pr;

pw_writebackground(pw, xd, yd, width, height, op)
```

Pixels are written to the pixwin *pw* in the rectangle defined by *xd*, *yd*, *width*, and *height*, using rasterop function *op* (as defined in *Constructing Op Arguments for Rop and Batchrop Pixrectops*). They are taken from the rectangle with its origin at *xs*, *ys* in the source pixrect pointed to by *pr*. *Pw_writebackground* simply supplies a null *pr* which indicates that an infinite source of pixels, all of which are set to zero, is used. The following draws a pixel of *value* at (*x*, *y*) in the addressed *pixwin*:

```
pw_put(pw, x, y, value)
    struct      pixwin *pw;
    int         x, y, value;
```

The next draws a vector of pixel *value* from (x0, y0) to (x1, y1) in the addressed *pixwin* using rasterop *op*:

```
pw_vector(pw, x0, y0, x1, y1, op, value)
    struct      pixwin *pw;
    int         op, x0, y0, x1, y1, value;

pw_replrop(pw, xd, yd, width, height, op, pr, xs, ys)
    struct      pixwin *pw;
    int         op, xd, yd, width, height;
    struct      pixrect *pr;
    int         xs, ys;
```

This procedure uses the indicated raster op function to replicate a pattern (found in the source *pixrect*) into a destination in a pixwin. For a full discussion of the semantics of this procedure, refer to the description of the equivalent procedure *pr_replrop* in *Pixel Data and Operations*.

The following two routines:

```
pw_text(pw, x, y, op, font, s)
   struct    pixwin *pw;
   int       x, y, op;
   struct    pixfont *font;
   char      *s;

pw_char(pw, x, y, op, font, c)
   struct    pixwin *pw;
   int       x, y, op;
   struct    pixfont *font;
   char      c;
```

write a string of characters and a single character respectively, to a *pixwin*, using rasterop *op* as above. *Pw_text* and *pw_char* are distinguished by their own coordinate system: the destination is given as the left edge and *baseline* of the first character. The left edge does not take into account any *kerning* (character position adjustment depending on its neighbors), so it is possible for a character to have some pixels to the left of the x-coordinate. The baseline is the y-coordinate of the lowest pixel of characters without descenders, 'L' or 'o' for example, so pixels will frequently occur both above and below the baseline in a string. *Font* may be NULL in which case the *system font* is used.

The system font is the same as the font returned from *pf_default*. In addition, the system font is reference counted and shared between software packages. To get the system font call *pw_pfsysopen*:

```
struct pixfont *pw_pfsysopen()
```

When you are done with the system font call *pw_pfsysclose*:

```
pw_pfsysclose()
```

*Note:* A font to be used in *pw_text* is required to have the same *pc_home.y* and character height for all characters in the font.

The following routine:

```
pw_ttext(pw, x, y, op, font, s)
   struct    pixwin *pw;
   int       x, y, op;
   struct    pixfont *font;
   char      *s;
```

is just like *pw_text* except that it writes *transparent* text. Transparent text writes the shape of the letters without disturbing the background behind it. This is most useful with color pixwins. Monochrome pixwins can use *pw_text* and a PIX_SRC|PIX_DST op, which is faster.

Applications such as displaying text perform the same operation on a number of pixrects in a fashion that is amenable to global optimization. The batchrop procedure is provided for these situations:

```
pw_batchrop(pw, dx, dy, op, items, n)
   struct    pixwin *pw;
   int       dx, dy, op, n;
   struct    batchitem items[ ];
```

*Pw_batchrop* is exactly analogous to *pr_batchrop* described in *Pixel Data and Operations*. Refer there for a detailed explanation of *pw_batchrop*.

Stencil ops are like raster ops except that the source pixrect is written through a stencil pixrect which functions as a spatial write enable mask. The indicated raster operation is applied only to destination pixels where the stencil pixrect is non-zero. Other destination pixels remain unchanged.

```
pw_stencil(dpw, dx, dy, dw, dh, op, stpr, stx, sty, spr, sx, sy)
    struct      pixwin *dpw;
    struct      pixrect *stpr, *spr;
    int         dx,dy,dw,dh,op,stx,sty,sx,sy;
```

*Pw_stencil* is exactly analogous to *pr_stencil* described in *Pixel Data and Operations*. Refer there for a detailed explanation of *pw_stencil*.

## 3.5.2. Read and Copy Routines

The following routines use the window as a source of pixels. They may find themselves thwarted by trying to read from a portion of the *pixwin* which is hidden, and therefore has no pixels. When this happens, *pw_fixup* in the *pixwin* structure will be filled in by the system with the description of the source areas which could not be accessed. The client must then regenerate this part of the image into the destination. Retained *pixwins* will always return *rl_null* in *pw_fixup* because the image is refreshed from *pw_prretained*. The following returns the value of the pixel at (*x, y*) in the addressed pixwin:

```
pw_get(pw, x, y)
    struct      pixwin *pw;
    int         x, y;
```

Pixels are read from the *pixwin* into a pixrect by:

```
pw_read(pr, xd, yd, width, height, op, pw, xs, ys)
    struct      pixwin *pw;
    int         op, xd, yd, width, height, xs, ys;
    struct      pixrect *pr;
```

Pixels are read from the rectangle defined by *xs, ys, width, height*, in the pixwin pointed to by *pw*, using rasterop function *op*. The pixels are stored in the rectangle with its origin at *xd, yd* in the *pixrect* pointed to by *pr*.

Copy is used when both source and destination are *pixwins*:

```
pw_copy(dpw, xd, yd, width, height, op, spw, xs, ys)
    struct      pixwin *dpw, *spw;
    int         op, xd, yd, width, height, xs, ys;
```

*Note:* Currently *dpw* and *spw* must be the same pixwin.

## 3.5.3. Bitplane Control

For pixwins on color display devices, one must be able to restrict access to certain bitplanes.

```
pw_putattributes(pw, planes)
    struct      pixwin *pw;
    int         *planes;
```

*Planes* is a bitplane access enable mask. Only those bits of the pixel corresponding to a 1 in the same bit position of **planes* will be affected by pixwin operations. *Pw_putattributes* sets the access enable mask of *pw*. If the *planes* argument is NULL, that attribute value will not be written.

*Note:* Use *pw_putattributes* with care; it changes the internal state of the pixwin until *pw_putattributes* is next called. Don't forget to restore the internal state once through accessing in this special mode.

```
pw_getattributes(pw, planes)
    struct      pixwin *pw;
    int         *planes;
```

retrieves the value of the access enable mask into **planes*.

## 3.6. Damage

When a portion of a client's window becomes visible after having been hidden, it is *damaged*. This may arise from several causes. For instance, an overlaying window may have been removed, or the client's window may have been stretched to give it more area. The client is notified that such a region exists by the signal SIGWINCH; this simply indicates that something about the window has changed in a fashion that probably requires repainting. It is possible that the window has shrunk, and no repainting of the image is required at all, but this is a degenerate case. It is then the client's responsibility to *repair* the damage by painting the appropriate pixels into that area. The following section describes how to do that.

### 3.6.1. Handling a SIGWINCH Signal

There are several stages to handling a SIGWINCH. First, in almost all cases, the procedure that catches the signal should *not* immediately try to repair the damage indicated by the signal. Since the signal is a software interrupt, it may easily arrive at an inconvenient time, halfway through a window's repaint for some normal cause, for instance. Consequently, the appropriate action in the signal handler is usually to set a flag which will be tested elsewhere. Conveniently, a SIGWINCH is like any other signal; it will break a process out of a *select* system call, so it is possible to awaken a client that was blocked, and with a little investigation, discover the cause of the SIGWINCH. See the *select*(2) system call and refer to the *tool_select* mechanism in *Tool Processing* for an example of this approach.

Once a process has discovered that a SIGWINCH has occurred and arrived at a state where it's safe to do something about it, it must determine exactly what has changed, and respond appropriately. There are two general possibilities: the window may have changed size, and/or a portion of it may have been uncovered.

*Win_getsize* (described in *Window Manipulation*) can be used to inquire the current dimensions of a window. The previous size must have been remembered, for instance from when the window was created or last adjusted. These two sizes are compared to see if the size has changed. Upon noticing that its size has changed, a window containing other windows may wish to

rearrange the enclosed windows, for example, by expanding one or more windows to fill a newly opened space.

Whether a size change occurred or not, the actual images on the screen must be fixed up. It is possible to simply repaint the whole window at this point — that will certainly repair any damaged areas — but this is often a bad idea because it typically does much more work than necessary.

Therefore, the window should retrieve the description of the damaged area, repair that damage, and inform the system that it has done so: The *pw_damaged* procedure:

```
pw_damaged(pw)
   struct      pixwin *pw;
```

is a procedure much like *pw_exposed*. It fills in *pwcd_clipping* with a *rectlist* describing the area of interest, stores the id of that *rectlist* in the *pixwin*'s *opsdata* and in *pwcd_damagedid* as well. It also stores its own address in *pwco_getclipping*, so that a subsequent lock will check the correct *rectlist*. All the clippers are set up too. Colormap segment offset initialization is done, as described in *Surface Preparation*.

Now is the time for the client to repaint its window — or at least those portions covered by the damaged *rectlist*; if the regeneration is relatively expensive, that is if the window is large, or its contents complicated, it may be worth restricting the amount of repainting *before* the clipping that the *rectlist* will enforce. This means stepping through the rectangles of the *rectlist*, determining for each what data contributed to its portion of the image, and reconstructing only that portion. See Appendix A for details about *rectlists*.

For retained pixwins, the following call can be used to copy the image from the backup pixrect to the window:

```
pw_repairretained(pw)
   struct      pixwin *pw;
```

When the image is repaired, the client should inform the window system with a call to:

```
pw_donedamaged(pw)
   struct      pixwin *pw;
```

*Pw_donedamaged* allows the system to discard the *rectlist* describing this damage. It is possible that more damage will have accumulated by this time, and even that some areas will be repainted more than once, but that will be rare.

After calling *pw_donedamaged*, the *pixwin* describes the entire visible area of the window.

A process which owns more than one window can receive a SIGWINCH for any of them, with no indication of which window generated it. The only solution is to fix up *all* windows. Fortunately, that should not be overly expensive, areas are completely and exactly specified by the returned value for *pw_damaged*.

## 3.7. Colormap Manipulation

Pixwins provide an interface to a basic colormap sharing mechanism. Portions of the colormap, *colormap segments*, are named and can be shared among cooperating processes. Use of a colormap segment, as opposed to the entire colormap, is essentially invisible to clients. Routines that access a pixwin's pixels do not distinguish between windows which use colormap segments and those which use the entire colormap.

### 3.7.1. Initialization

*Pw_open* and *pw_region* both create and return a pixwin. If a colormap segment is already defined for the window of the pixwin, this is the colormap segment used in the new pixwin. However, if the window has no colormap segment defined for it, a monochrome colormap segment is setup for the pixwin by default. The default segment is defined in */usr/include/sunwindow/cms_mono.h.*

### 3.7.2. Background and Foreground

Every colormap segment has two distinguished values, its *background* and *foreground.* The background color is defined as the value at the first position of a colormap segment. The foreground color is defined as the value at the last position of a colormap segment (the colormap segment's size minus 1).

The foreground is important in terms of color/monochrome compatibility. Any source color, other than 0, that is written on a monochrome pixrect is translated to the foreground color.

*Pw_open* and *pw_region* set the background and foreground of the returned pixwin to be those of the overall screen (see *win_screenget*). In addition, if the screen is defined as being inverted, the background and foreground are reversed. For reasons involving image blanking, invisible cursors, merged boundaries, color/monochrome compatibility, relative colormap segment placement, and so on, this pre-emption is vital to the overall integrity of color displays doing colormap sharing. Monochrome displays have many of these problems, although they are less severe than on color displays.

Here are handy utilities to set two specific colormap segment entries:

```
pw_reversevideo(pw, min, max)
    struct      pixwin *pw;
    int         min, max;

pw_blackonwhite(pw, min, max)
    struct      pixwin *pw;
    int         min, max;

pw_whiteonblack(pw, min, max)
    struct      pixwin *pw;
    int         min, max;
```

*Min* and *max* are the first and last entries in the colormap, respectively. If *min* is the background and *max* is the foreground and *pw* is a color pixwin, these calls do nothing.

### 3.7.3. A New Colormap Segment

For a different colormap segment for a pixwin, a new name must be created. If the colormap segment's usage is to be static in nature, by all means try to use a shared colormap segment definition. The colormap segment definitions that could be shared with other windows are in */usr/include/sunwindow/cms_*.h.* These are *cms.h, cms_rgb.h, cms_grays.h, cms_mono.h,* and *cms_rainbow.h.* Even if no other program shares your colormap segment, at least multiple instances of your program could share it. Remember that colormap entries are scarce.

If this new colormap segment should not be shared by another window then the name should be unique. A common way to generate a unique name is to append your process id to a more meaningful string that describes the usage of the colormap segment.

```
pw_setcmsname(pw, cmsname)
    struct      pixwin *pw;
    char        cmsname[20];
```

*Cmsname* is the name that *pw* will call its window's colormap segment. Just setting the name has the effect of resetting the colormap segment to a NULL entry. Usually, the very next call after *pw_setcmsname* should be *pw_putcolormap* as described in the next section.

Colormap segments are associated with windows, not pixwins. Each window can have only one colormap segment. Pixwins provide an interface for managing that one colormap segment. Since more than one pixwin may exist per window, care should be taken to avoid changing the colormap segment definition out from underneath another pixwin on the same window.

```
pw_getcmsname(pw, cmsname)
    struct      pixwin *pw;
    char        cmsname[20];
```

The colormap segment name of *pw* is copied into *cmsname.*

## 3.7.4. Colormap Access

```
pw_putcolormap(pw, index, count, red, green, blue)
    struct              pixwin *pw;
    int                 index, count;
    unsigned char       red [ ], green[ ], blue[ ];
```

The *count* elements of the pixwin's colormap segment starting at *index* (zero relative) are loaded with the first *count* values in the three arrays. A colormap has three components each indexed by a given pixel value to produce an RGB color. Monochrome pixwins assume red equals green equals blue. Pixrects of depth 8 have colormaps with 256 (2 to the eighth) entries. Background and foreground values are forced to the values defined by the screen.

```
pw_getcolormap(pw, index, count, red, green, blue)
    struct              pixwin *pw;
    int                 index, count;
    unsigned char       red [ ], green[ ], blue[ ];
```

finds out the state of the colormap segment. The arguments are analogous to those of *pw_putcolormap.*

The utility:

```
pw_cyclecolormap(pw, cycles, index, count)
    struct      pixwin *pw;
    int         cycles, index, count;
```

is handy for taking a portion of *pw*'s colormap segment, starting at *index* for *count* entries, and rotating those entries among themselves for *cycles*. A cycle is defined as the *count* shifts it takes one entry to move through every position once.

## 3.7.5. Surface Preparation

In order for a client to ignore the offset of his colormap segment the image of the pixwin must be initialized to the value of the offset. This *surface preparation* is done automatically by pixwins under the following circumstances:

- The routine *pw_damaged* does surface preparation on the area of the pixwin that is damaged.
- The routine *pw_putcolormap* does surface preparation over the entire exposed portion of a pixwin if a new colormap segment is being loaded for the first time.

For monochrome displays, nothing is done during surface preparation. For color displays, when the surface is prepared, the low order bits (colormap segment size minus 1) are not modified. This mean that surface preparation does not clear the image. Initialization of the image (often clearing) is still the responsibility of client code.

There is a case in which surface preparation must be done explicitly by client code. When window boundaries are knowingly violated (see *win_grabio*), as in the case of pop-up menus, the following procedure must be called to prepare each rectangle on the screen that is to be written upon:

```
pw_preparesurface(pw, rect)
    struct      pixwin *pw;
    struct      rect *r;
```

*Rect* is relative to *pw*'s coordinate system. Most commonly, a saved copy of the area to be written is made so that it can be restored later.

# Chapter 4

# Window Manipulation

This chapter describes the *sunwindow* facilities for creating, positioning, and controlling windows. It continues the discussion begun in *Overlapped Windows: Imaging Facilities*, on the *sunwindow* level that allows displaying images on windows which may be overlapped.

The structure that underlies the operations described in this chapter is maintained within the window system, and is accessible to the client only through system calls and their procedural envelopes, it will not be described here. The window is presented to the client as a *device*; it is represented, like other devices, by a *file descriptor* returned by *open*. It is manipulated by other I/O calls, such as *select*, *read*, *ioctl*, and *close*. *Write* to a window is not defined, since all the facilities of the previous chapter on *Overlapped Windows: Imaging Facilities* are required to display output on a window.

The header file */usr/include/sunwindow/window_hs.h* includes the header files needed to work at this level of the window system.

## 4.1. Window Data

The information about a window maintained by the window system includes:

- two rectangles which refer to alternative *sizes* and *positions* for the window on the screen;
- a series of links that describe the window's position in a hierarchical database, which determines its *overlapping* relationships to other windows;
- clipping information used in the processing described in *Overlapped Windows: Imaging Facilities*;
- the image used to track the mouse when it is in the window;
- the id of the process which should receive SIGWINCH signals for the window (this is the *owner* process);
- a mask that indicates what user input actions the window should be notified of;
- another window, which is given any input events that this window does not use; and
- 32 bits of data private to the window client.

## 4.2. Window Creation, Destruction, and Reference

As mentioned above, windows are *devices*. As such, they are special files in the */dev* directory with names of the form "*/dev/winn*", where *n* is a decimal number. A window is created by opening one of these devices, and the window name is simply the filename of the opened device.

### 4.2.1. A New Window

The first process to open a window becomes its *owner*. A process can obtain a window it is guaranteed to own by calling:

```
int win_getnewwindow()
```

This finds the first unopened window, opens it, and returns a file descriptor which refers to it. If none can be found, it returns -1. A file descriptor, often called the *windowfd*, is the usual handle for a window within the process that opened it.

When a process is finished with a window, it may close it. This is the standard *close*(3) system call with the window's file descriptor as argument. As with other file descriptors, a window left open when its owning process terminates will be closed automatically by the operating system.

Another procedure is most appropriately described at this point, although in fact clients will have little use for it. To find the next available window, *win_getnewwindow* uses:

```
int win_nextfree(fd)
    int         fd;
```

where *fd* is a file descriptor it got by opening */dev/win0*. The return value is a *window number*, as described in *References to Windows* below; a return value of WIN_NULLLINK indicates there is no available unopened window.

### 4.2.2. An Existing Window

It is possible for more than one process to have a window open at the same time; *Providing for Naive Programs* presents one plausible scenario for using this capability. The window will remain open until all processes which opened it have closed it. The coordination required when several processes have the same window open is described in *Providing for Naive Programs*.

### 4.2.3. References to Windows

Within the process which created a window, the usual handle on that window is the file descriptor returned by *open* and *win_getnewwindow*. Outside that process, the file descriptor is not valid; one of two other forms must be used. One form is the *window name* (e.g., /dev/win12); the other form is the *window number*, which corresponds to the numeric component of the window name. Both of these references are valid across process boundaries. The *window number* will appear in several contexts below.

Procedures are supplied for switching the various window identifiers back and forth. *Win_numbertoname* stores the filename for the window whose number is *winnumber* into the buffer addressed by *name*:

```
win_numbertoname(winnumber, name)
    int         winnumber;
    char        *name;
```

*Name* should be WIN_NAMESIZE long as should all the name buffers in this section.

*Win_nametonumber* returns the window number of the window whose name is passed in *name*:

```
int win_nametonumber(name)
    char        *name;
```

Given a window file descriptor, *win_fdtoname* stores the corresponding device name into the buffer addressed by *name*:

```
win_fdtoname(windowfd, name)
    int         windowfd;
    char        *name;
```

The following returns the window number for the window whose file descriptor is *windowfd*:

```
int win_fdtonumber(windowfd)
    int         windowfd;
```

## 4.3. Window Geometry

Once a window has been opened, its size and position may be set. The same routines used for this purpose are also helpful for adjusting the screen positions of a window at other times, when user-interface actions indicate that it is to be moved or stretched, for instance. The basic procedures are:

```
win_getrect(windowfd, rect)
    int         windowfd;
    struct      rect *rect;

win_getsize(windowfd, rect)
    int         windowfd;
    struct      rect *rect;

short win_getheight(windowfd)
    int         windowfd;

short win_getwidth(windowfd)
    int         windowfd;
```

*Win_getrect* stores the rectangle of the window whose file descriptor is the first argument into the rect addressed by the second argument; the origin is relative to that window's parent. *Setting Window Links* explains what is meant by a window's "parent."

*Win_getsize* is similar, but the rectangle is self-relative — that is, the origin is (0,0).

*Win_getheight* and *win_getwidth* return the single requested dimension for the indicated window. *Win_setrect* copies the rect argument's data into the rect of the indicated window:

```
win_setrect(windowfd, rect)
    int         windowfd;
    struct      rect *rect;
```

This changes its size and/or position on the screen. The coordinates are in the coordinate system of the window's parent.

```
win_getsavedrect(windowfd, rect)
    int         windowfd;
    struct      rect *rect;

win_setsavedrect(windowfd, rect)
    int         windowfd;
    struct      rect *rect;
```

A window may have an alternate size and location; this facility is useful for *icons*, for example (see *Full Screen Access*). The alternate rectangle may be read with *win_getsavedrect*, and written with *win_setsavedrect*. As with *win_getrect* and *win_setrect*, the coordinates are relative to the window's parent.

## 4.4. The Window Hierarchy

Position in the window database determines the nesting relationships of windows, and therefore their overlapping and obscuring relationships. Once a window has been opened and its size set, the next step in creating a window is to define its relationship to the other windows in the system. This is done by setting links to its neighbors, and inserting it into the window database.

### 4.4.1. Setting Window Links

The window database is a strict hierarchy. Every window (except the root) has a parent; it also has 0 or more *siblings* and *children*. In the terminology of a family tree, *age* corresponds to *depth* in the layering of windows on the screen: parents underlie their offspring, and older windows underlie younger siblings which intersect them on the display. Parents also enclose their children, which means that any portion of a child's image that is not within its parent's rectangle is clipped. Depth determines overlapping behavior: the *uppermost* image for any point on the screen is the one that gets displayed. Every window has links to its parent, its older and younger siblings, and to its oldest and youngest children.

Windows may exist outside the structure which is being displayed on a screen; they are in this state as they are being set up, for instance.

The links from a window to its neighbors are identified by *link selectors*; the value of a link is a *window number*. An appropriate analogy is to consider the *link selector* as an array index, and the associated *window number* as the value of the indexed element. To accommodate different viewpoints on the structure there are two sets of equivalent selectors defined for the links:

```
WL_PARENT       == WL_ENCLOSING
WL_OLDERSIB     == WL_COVERED
WL_YOUNGERSIB   == WL_COVERING
WL_OLDESTCHILD  == WL_BOTTOMCHILD
WL_YOUNGESTCHILD == WL_TOPCHILD
```

A link which has no corresponding window, a child link of a "leaf" window, for instance, has the value WIN_NULLLINK.

When a window is first created, all its links are null. Before it can be used for anything, at least the parent link must be set. If the window is to be attached to any siblings, those links should be set in the window as well. The individual links of a window may be inspected and changed by the following procedures.

*Win_getlink* returns a window number.

```
int win_getlink(windowfd, link_selector)
  int       windowfd, link_selector;
```

This number is the value of the selected link for the window associated with *windowfd.*

```
win_setlink(windowfd, link_selector, value)
  int       windowfd, link_selector, value;
```

*Win_setlink* sets the selected link in the indicated window to be *value,* which should be another window number. The actual window number to be supplied may come from one of several sources: if the window is one of a related group, all created in the same process, file descriptors will be available for the other windows. Their window numbers may be derived from the file descriptors via *win_fdtonumber.* The window number for the parent of a new window or group of windows is not immediately obvious, however. The solution is a convention that the WINDOW_PARENT environment parameter will be set to the filename of the parent. See *Passing Parameters to a Tool* for an example of this environment parameter's usage.

## 4.4.2. Activating the Window

Once a window's links have all been defined, the window is inserted into the tree of windows and attached to its neighbors by a call to

```
win_insert(windowfd)
  int       windowfd;
```

This call causes the window to be inserted into the tree, and all its neighbors to be modified to point to it. This is the point at which the window becomes available for display on the screen.

Every window should be inserted after its rectangle(s) and link structure have been set, but the insertion need not be immediate: if a subtree of windows is being defined, it is appropriate to create the window at the root of this subtree, create and insert all of its descendants, and then, when the subtree is fully defined, insert its root window. This activates the whole subtree in a single action, which typically will result in a cleaner display interaction.

Once a window has been inserted in the window database, it is available for input and output. At this point, it is appropriate to call *pw_open* and access the screen.

## 4.4.3. Modifying Window Relationships

Windows may be rearranged in the tree. This will change their overlapping relationships. For instance, to bring a window to the top of the heap, it should be moved to the "youngest" position among its siblings. And to guarantee that it is at the top of the display heap, each of its ancestors must likewise be the youngest child of *its* parent.

To accomplish such a modification, the window should first be removed:

```
win_remove(windowfd)
  int       windowfd;
```

After the window has been removed from the tree, it is safe to modify its links, and then reinsert it.

A process doing multiple window tree modifications should lock the window tree before it begins. This prevents any other process from performing a conflicting modification. This is done with a call to:

```
win_lockdata(windowfd)
    int         windowfd;
```

After all the modifications have been made and the windows reinserted, the lock is released with a call to:

```
win_unlockdata(windowfd)
    int         windowfd;
```

Most routines described in this chapter, including the four above, will block temporarily, if another process either has the database locked, or is writing to the screen, and the window adjustment has the possibility of conflicting with the window that is being written.

As a method of deadlock resolution, SIGXCPU is sent to a process that spends more that 10 seconds of real time inside a window data lock, and the lock is broken.

## 4.5. User Data

Each window has 32 bits of uninterpreted client data associated with it. This is not touched by the basic window system; typically the client uses it to store flags. Higher levels of the system may implement minimal inter-window status-sharing through this facility. This data is manipulated with the following procedures:

```
win_getuserflags(windowfd)
    int         windowfd;

win_setuserflags(windowfd, flags)
    int         windowfd;
    int         flags;

win_setuserflag(windowfd, flag, value)
    int         windowfd;
    int         flag;
    int         value;
```

*Win_getuserflags* returns the user data. *Win_setuserflags* stores its *flags* argument into the window struct. *Win_setuserflag* uses *flag* as a mask to select one or more flags in the data word, and sets the selected flags on or off as *value* is TRUE or FALSE.

## 4.6. Minimal-Repaint Support

This section has strong connections to the preceding chapter and to Appendix A on *Rects and Rectlists*. Readers should refer to both from here.

Moving windows about on the screen may involve repainting large portions of their image in new places. Often, the existing image can be copied to the new location, saving the cost of regenerating it. Two procedures are provided to support this function:

```
win_computeclipping(windowfd)
    int         windowfd;
```

causes the window system to recompute the *exposed* and *damaged* rectlists for the windows on the screen while withholding the SIGWINCH that will tell each owner to repair damage.

*Win_partialrepair:*

```
win_partialrepair(windowfd, r)
    int         windowfd;
    struct      rect *r;
```

tells the window system to remove the rectangle *r* from the damaged area for the window identified by *windowfd*. This operation is a no-op if *windowfd* has damage accumulated from a previous window database change, but has not told the window system that it has repaired that damage.

Any window manager can use these facilities according to the following strategy:

- The old exposed areas for the affected windows are retrieved and cached. (*pw_exposed*)
- The window database is locked and manipulated to accomplish the rearrangement. (*win_lockdata, win_remove, win_setlink, win_setrect, win_insert* ...)
- The new area is computed, retrieved, and intersected with the old. (*win_computeclipping, pw_exposed, rl_intersection*)
- Pixels in the intersection are copied, and those areas are removed from the subject window's damaged area. (*pw_lock, pr_copy, win_partialrepair*)
- The window database is unlocked, and any windows still damaged get the signals informing them of the reduced damage which must be repaired.

## 4.7. Multiple Screens

Multiple displays may be simultaneously attached to a workstation, and clients may want windows on all of them. Therefore, the window database is a forest, with one tree of windows for each display. Thus, there is no overlapping of window trees that belong to different screens. For displays that share the same mouse device, the physical arrangement of the displays can be passed to the window system, and the mouse cursor will pass from one screen to the next as though they were continuous.

```
struct singlecolor {
    u_char      red, green, blue;
};

struct screen {
    char        scr_rootname[SCR_NAMESIZE];
    char        scr_kbdname[SCR_NAMESIZE];
    char        scr_msname[SCR_NAMESIZE];
    char        scr_fbname[SCR_NAMESIZE];
    struct      singlecolor scr_foreground;
    struct      singlcolor scr_background;
    int         scr_flags;
    struct      rect scr_rect;
};

#define SCR_NAMESIZE                20
#define SCR_SWITCHBKGRDFRGRD        0x1
```

The *screen* structure describes a client's notion of the display screen. There are also fields indicating the input devices associated with the screen. *Scr_rootname* is the device name of the window which is at the base of the window display tree for the screen; the default is "*/dev/win0*". *Scr_kbdname* is the device name of the keyboard associated with the screen; the default is "*/dev/kbd*". *Scr_msname* is the device name of the mouse associated with the screen; the default is "*/dev/mouse*". *Scr_fbname* is the device name of the frame buffer on which the screen is displayed; the default is "*/dev/fb*". *Scr_kbdname, scr_msname* and *scr_fbname* can have the string "NONE" if no device of the corresponding type is to be associated with the screen. *Scr_foreground* is three RGB color values that define the foreground color used on the frame buffer; the default is {colormap size–1, colormap size–1, colormap size–1}. *Scr_background* is three RGB color values that define the background color used on the frame buffer; the default is {0, 0, 0}. The default values of the background and foreground yield a black on white image. *Scr_flags* contains boolean flags; the default is 0. SCR_SWITCHBKGRDFRGRD is a flag that directs any client of the background and foreground data to switch their positions, thus providing a video reversed image (usually yielding a white on black image). *Scr_rect* is the size and position of the screen on the frame buffer; the default is the entire frame buffer surface.

*Win_screennew*

```
int win_screennew(screen)
    struct      screen *screen;
```

opens and returns a window file descriptor for a root window. This new root window resides on the new screen which was defined by the specifications of **screen*. Any zeroed field in **screen* tells *win_screennew* to use the default value for that field (see above for defaults). Also, see the description of *win_initscreenfromargv* below. If –1 is returned, an error message is displayed to indicate that there was some problem creating the screen.

There can be as many screens as there are frame buffers on your machine and *dtop* devices configured into your kernel. The kernel calls screen instances *desktops* or *dtops*.

*Win_screenget*:

```
win_screenget(windowfd, screen)
   int         windowfd;
   struct      screen *screen;
```

fills in the addressed struct *screen* with information for the screen with which the window indicated by *windowfd* is associated.

*Win_screendestroy*:

```
win_screendestroy(windowfd)
   int         windowfd;
```

causes each window owner process (except the invoking process) on the screen associated with *windowfd* to be sent a SIGTERM signal.

*Win_setscreenpositions* informs the window system of the logical layout of multiple screens:

```
win_setscreenpositions(windowfd, neighbors)
   int         windowfd, neighbors[SCR_POSITIONS];

#define SCR_NORTH      0
#define SCR_EAST       1
#define SCR_SOUTH      2
#define SCR_WEST       3

#define SCR_POSITIONS  4
```

This enables the cursor to cross to the appropriate screen. *Windowfd*'s window is the root for its screen; the four slots in *neighbors* should be filled in with the window numbers of the root windows for the screens in the corresponding positions. No diagonal neighbors are defined, since they are not strictly neighbors.

*Win_getscreenpositions* fills in *neighbors* with *windowfd*'s screen's neighbors:

```
win_getscreenpositions(windowfd, neighbors)
   int         windowfd, neighbors [SCR_POSITIONS];
```

*Win_setkbd*:

```
int win_setkbd(windowfd, screen)
   int         windowfd;
   struct      screen *screen;
```

is used to change the keyboard associated with *windowfd*'s screen. Only the data relative to the keyboard is used (i.e., *screen-> scr_kbdname*).

*Win_setms*:

```
int win_setms(windowfd, screen)
   int         windowfd;
   struct      screen *screen;
```

is used to change the mouse associated with *windowfd*'s screen. Only the data relative to the mouse is used (i.e., *screen->scr_msname*).

*Win_initscreenfromargv*:

```
int win_initscreenfromargv(screen, argv)
   struct      screen *screen;
   char        **argv;
```

can be used to do a standard command line parse of *argv* into **screen*. **Screen* is first zeroed. The syntax is:

[-d *display device*] [-m *mouse device*] [-k *keyboard device*] [-l] [-f *red green blue*] [-b *red green blue*]

See *suntools*(1) for semantics and details.

## 4.8. Cursor and Mouse Manipulations

This section describes the interface to the mouse and the cursor that follows the mouse. Both of which are maintained by the window system internals.

### 4.8.1. Cursors

The cursor is the image which tracks the mouse on the screen:

```
struct cursor {
    short       cur_xhot, cur_yhot;
    int         cur_function;
    struct      pixrect *cur_shape;
};

#define CUR_MAXIMAGEWORDS16
```

*Cur_shape* points to a memory pixrect which holds the actual image for the cursor. The window system supports a *cur_shape.pr_data->md_image* up to CUR_MAXIMAGEWORDS words.

The "hot spot" defined by (*cur_xhot, cur_yhot*) associates the cursor image, which has height and width, with the mouse position, which is a single point on the screen. The hot spot gives the mouse position an offset from the upper-left corner of the cursor image.

Most cursors have a hot spot whose position is dictated by the image shape, the tip of an arrow, the center of a bullseye, the center of a cross-hair. Cursors can also be used as a status feedback mechanism, an hourglass to indicate that some processing is occurring for instance. This type of cursor should have the hot spot located in the middle of its image so the user has a definite spot for pointing and does not have to guess where the hot spot is.

The function indicated by *cur_function* is a rasterop (as described in *Constructing Op Arguments for Rop and Batchrop Pixrectops*), which will be used to paint the cursor. PIX_SRC | PIX_DST is generally effective on light backgrounds, for example in text, but invisible over solid black. PIX_SRC ^ PIX_DST is a reasonable compromise over many different backgrounds, although it does poorly over a gray pattern.

```
win_getcursor(windowfd, cursor)
    int         windowfd;
    struct      cursor *cursor;
```

stores a copy of the cursor that is currently being used on the screen into the buffer addressed by *cursor*.

*Win_setcursor*:

```
win_setcursor(windowfd, cursor)
    int         windowfd;
    struct      cursor *cursor;
```

sets the cursor and function that will be used whenever the mouse position is within the indicated window.

If a window process does not want a cursor displayed, the appropriate mechanism is to set the cursor to one whose dimensions are both 0.

### 4.8.2. Mouse Position

Determining the mouse's current position is treated under *Input to Application Programs*. We note here that the standard procedure for a process to track the mouse is to arrange to receive an input event every time the mouse moves; and in fact, the mouse position is passed with *every* user input a window receives.

The mouse position can be reset under program control; that is, the cursor can be moved on the screen, and the position that is given for the mouse in input events can be reset without the mouse on the table top being physically moved:

```
win_setmouseposition(windowfd, x, y)
    int         windowfd, x, y;
```

puts the mouse position at (*x, y*) in the coordinate system of the window indicated by *windowfd*. The result is a jump from the previous position to the new one without touching any points between. Input events occasioned by the move, window entry and exit and cursor changes, will be generated. This facility should be used with restraint, as users are likely to lose a cursor that moves independently of their control.

Occasionally it is necessary to discover which window underlies the cursor, usually because a window is handling input for all its children. The procedure used for this purpose is:

```
int win_findintersect(windowfd, x, y)
    int         windowfd, x, y;
```

where *windowfd* is the calling window's file descriptor, and (*x, y*) define a screen position in that window's coordinate space. The returned value is a window number. *X* and *y* may lie outside the bounds of the window.

## 4.9. Providing for Naive Programs

There is a large class of applications that are relatively unsophisticated about the window system, but want to run in windows anyway. For example, a simple-minded graphics program may want a window in which to run, but doesn't want to know about all the details of creating and positioning it. This section describes a way of allowing for these applications.

### 4.9.1. Which Window to Use

SunWindows defines an important environment parameter, WINDOW_GFX. By convention, WINDOW_GFX is set to a string that is the device name of a window in which graphics programs should be run. This window is already opened and installed in the window tree. Routines exist to read and write this parameter:

```
int we_getgfxwindow(name)
    char        *name

we_setgfxwindow(name)
    char        *name
```

*We_getgfxwindow* returns a non-zero value if it cannot find a value.

## 4.9.2. The Blanket Window

A good way to take over an existing window is to create a new window that becomes attached to and covers the existing window. Such a covering window is called a *blanket* window. The covered window will be called the *parent* window in this subsection because of its window tree relationship with a blanket window. *Note:* It's a bad idea to take over an existing window using *win_setowner*.

Using the parent window name from the environment parameter WINDOW_GFX (described above), *open*(2) the parent window. Get a new window to be used as the blanket window using *win_getnewwindow*. Now call:

```
int win_insertblanket(blanketfd, parentfd)
    int         blanketfd, parentfd;
```

A non-zero return value indicates success. As the parent window changes size and position the blanket window will automatically cover the parent.

To remove the blanket window from on top of the parent window call:

```
win_removeblanket(blanketfd)
    int         blanketfd;
```

If the process that created the blanket window dies before *win_removeblanket* can be called, the blanket window will automatically be removed and destroyed upon automatic closure of the window device. This automatic closure happens because the only open file descriptor on it will be in the creating process.

A non-zero return value from *win_isblanket* indicates that *blanketfd* is indeed a blanket window.

```
int win_isblanket(blanketfd)
    int         blanketfd;
```

## 4.10. Window Ownership

*Note:* Do not use the two routines in this section for *temporarily* taking over another window. These routines are included for backwards compatibility reasons.

SIGWINCH signals are directed to the process that *owns* the window, the owner normally being the process that created the window. The following procedures may read from and write to the window:

```
int win_getowner(windowfd)
    int         windowfd;

win_setowner(windowfd, pid)
    int         windowfd, pid;
```

*Win_getowner* returns the process id of the indicated window owner. If the owner doesn't exist, zero is returned. *Win_setowner* makes the process identified by *pid* the owner of the window indicated by *windowfd*. *Win_setowner* causes a SIGWINCH to be sent to the new owner.

## 4.11. Error Handling

Except as explicitly noted, the procedures described in this section do not return error codes. The standard error reporting mechanism inside the *sunwindow* library is to call a procedure that displays a message, typically identifying the *ioctl* call that detected the error. After the message display, the calling process resumes execution.

This default error handling routine may be replaced by calling:

```
int             (*win_errorhandler(win_error))()
    int         (*win_error)();
```

The *win_errorhandler* procedure takes the address of one procedure, the new error handler, as an argument and returns the address of another procedure, the old error handler, as a result. Any error handler procedure should be a function that returns an **int**.

```
win_error(errnum, winopnum)
    int         errnum, winopnum;
```

*Errnum* will be –1 indicating that the actual error number is found in the global *errno*. *Winopnum* is the *ioctl* number that defines the window operation that generated the error. See *Error Message Decoding* in *Programming Notes* in the appendix.

# Chapter 5

# Input to Application Programs

This chapter continues the description of the *sunwindow* level of the Sun window system. Here we discuss how user input is made available to application programs. Unless otherwise noted, the structures and procedures discussed in this section are found in the header file */usr/include/sunwindow/win_input.h.*

The window system provides facilities which meet two distinct needs regarding input to an application program:

- A uniform interface to multiple input devices allows programs to deal with varying keyboards and positioning devices, ignoring complexities due to facilities which the programs do not use.
- Several different keyboards are available with Sun systems; they differ in the number and arrangement of keys. At a minimum, some clients will require ASCII characters, one per keystroke. More sophisticated clients will assign special values to non-standard keys (such as "META" characters in the range 0x80 and above). Some clients will assign functions to particular keys on the keyboard, and will distinguish key-down from key-up events.
- The standard positioning device on a Sun is the mouse, which reports a location and the state of three buttons. Alternatively, some clients may use a tablet and stylus, or in place of the stylus, a "puck" with as many as 10 buttons on it.
- In some client systems, the time between input events is significant; for example, when smoothing a user's stylus trace, or assigning special meaning to multiple clicks of a button within a short period.

The window system allows clients with only the simplest requirements to ignore all the complications, while providing more sophisticated clients the facilities they require. The mechanism for accomplishing this is called the *virtual input device.* This mechanism with its input events is described in *Virtual Input Device.*

- The second major section of this chapter describes how user inputs are collected from multiple sources, serialized, and distributed among multiple consumers. Multiple clients are able to accept inputs concurrently, and a slow consumer does not affect other clients' ability to receive their inputs. Type-ahead and mouse-ahead are fully supported.
    - Client programs operate under the illusion that they have the user's full attention, leaving the window system to handle the multiplexing. Therefore, a client sees precisely those input events that the user has directed to that application.
    - Conversely, the client may require inputs from multiple devices, where the exact sequences across all those devices is significant. The order of mouse and function key events is likely to be significant, for instance. This is provided for via a single unified input stream, rather than requiring polling of multiple streams, which would be

unacceptable in a multi-processed environment.

- The distribution of input events takes into account the window's indication of what events it is prepared to handle; other events are redirected, allowing a division of labor among the various components of a system.

## 5.1. The Virtual Input Device

This section describes the virtual device which generates user input, and how the input is presented to the client process. The device appears as an extended keyboard, different from existing keyboards, but incorporating the common features of most of them. It also incorporates a *locator* which indicates a screen position, and a clock which reports a time in seconds and microseconds.

### 5.1.1. Uniform Input Events

Each user action generates an *input event*, which is reported in a uniform format regardless of the event. An event is reported in the following struct:

```
struct inputevent {
    short       ie_code;
    short       ie_flags;
    short       ie_shiftmask;
    short       ie_locx;
    short       ie_locy;
    struct      timeval ie_time;
};
```

*Ie_code* identifies the source of the event, as a switch position on a Virtual Input Device. The exact definition of the codes is given in *Event Codes*. In general, the input events fall into one of three classes: events that generate a single ASCII character; events related to locator motion and window geometry; and events identified with invocation of a special function, usually involving the depression or release of a single special button on the mouse or keyboard. These classes are known as ASCII, pseudo, and function events, respectively.

The information provided by the code in *ie_code* is interpreted according to event flags in *ie_flags*. (See *Event Flags* below.)

The remaining elements of the struct provide general status information which may be useful on any event:

*ie_shiftmask* is used to report the state of certain shift-keys that is, to modify the meaning of other events.

*ie_locx* and

*ie_locy* provide the position of the locator in the window's coordinate system at the time the event occurred.

*ie_time* provides a timestamp for the event, in the format of a system *timeval*, as defined in */usr/include/sys/time.h*.

## 5.1.2. Event Codes

Event codes can take on any value in the range from 0 to 65535 inclusive. Of the codes defined in the header file, 256 are assigned to the ASCII event class and the other 128 are partitioned between the pseudo and function event classes. The following constants define the number of codes and the first and last code in the latter two classes:

```
#define VKEY_CODES    128
#define VKEY_FIRST    32512
#define VKEY_LAST     VKEY_FIRST+ VKEY_CODES-1
```

### 5.1.2.1. ASCII Events

The event codes in the range 0 to 511 inclusive are assigned to the ASCII event class. This class is further sub-divided:

```
#define ASCII_FIRST 0
#define ASCII_LAST 127
```

In particular, striking a key which has an obvious ASCII meaning causes the Virtual Input Device to enqueue for the client an event whose code is the 7-bit ASCII character corresponding to that key. Such a key with an obvious ASCII meaning is one in the main typing array labelled with a single letter of the alphabet. This is independent of the physical keyboard actually used. A slight complication occurs because of the presence of both upper- and lower-case characters in ASCII: if the user "shifts" the physical keyboard by depressing the CAPS-LOCK, SHIFT-LOCK, or SHIFT key the *ie_code* contains the shifted ASCII character corresponding to the struck key.

For physical keystations that are mapped to cursor control keys, the current implementation transmits a series of events with codes that correspond to the ANSI X3.64 7-bit ASCII encoding for the cursor control function. For physical keystations that are mapped to function keys, the current implementation transmits a series of events with codes that correspond to an ANSI X3.64 user-definable escape sequence. For further details, see *kbd*(5).

```
#define META_FIRST 128
#define META_LAST 255
```

Event codes from 128 to 255 inclusive are generated when the client has META translation enabled and the user strikes a key that would generate a 7-bit ASCII code while the META key is also depressed. In this case, the event code is the 7-bit ASCII code added to META_FIRST.

### 5.1.2.2. Function Events

Event codes in the function class correspond to button strikes that do not result in generation of an event code in the ASCII class.

In the function class are the event codes associated with locator buttons:

```
#define BUT(i)
```

A physical locator often has up to 10 buttons connected to it. Alternatively, even though the physical locator does not have any buttons physically available on it, it may have buttons on another device assigned to it. A light pen is an example of such a locator. In either case, each of the $n$ buttons (where $0 < n <= 10$) associated with the Virtual Input Device's locator are

assigned an event code; the *ith* button is assigned the code *BUT(i)*. Thus a 3-button mouse reports x and y and buttons 1 – 3.

In the function class are the event codes associated with keyboard function keys that don't generate single ASCII charaters:

```
#define KEY_LEFT(i)
#define KEY_RIGHT(i)
#define KEY_TOP(i)
#define KEY_BOTTOMLEFT
#define KEY_BOTTOMRIGHT
```

The function keys in the Virtual Input Device define an idealized standard layout that groups keys by location: 16 left, 16 right, 16 top and 2 bottom. While the actual position on the keyboard may be different, it is convenient to provide some grouping for the large number of function keys. The mapping to physical keys on various keyboards is defined in */usr/include/sundev/kbd.h* and discussed in *kbd*(5).

### 5.1.2.3. Pseudo Events

```
#define VKEY_FIRSTPSEUDO
#define VKEY_LASTPSEUDO
```

Event codes in the pseudo class are events that involve locator movement instead of physical button striking. The physical locator constantly provides an (x, y) coordinate position in pixels; this position is transformed by the Virtual Input Device to the coordinate system of the window receiving an event. In order to watch actual locator movement (or lack thereof), the client must be enabled for the events with codes.

```
#define LOC_MOVE
#define LOC_MOVEWHILEBUTDOWN
#define LOC_STILL
```

A LOC_MOVE is reported only when the locator actually moves. Since fast motions may yield non-adjacent locations in consecutive events, the locator tracking mechanism reports the current position at a set sampling rate currently 40 times per second.

LOC_MOVEWHILEBUTDOWN is like LOC_MOVE but happens only when a button on the locator is down.

A single LOC_STILL event is reported when the locator has been still for a moment, currently 1/5 of a second.

Clients can be notified when the locator has entered or exited a window via the event codes:

```
#define LOC_WINENTER
#define LOC_WINEXIT
```

## 5.1.3. Event Flags

Only one event flag is currently defined:

```
#define IE_NEGEVENT
```

indicates the event was "negative." Positive events include depression of any button or key,

including buttons on the locator, motion of the locator device while it is available to this client, and entry of the cursor into a window. The only currently defined negative event is the release of a depressed button. Stopping of the locator and locator exit from the window are positive events, distinct from locator motion and window entry. This asymmetry allows a client to be informed of these events without the performance penalty associated with receiving all negative events and then discarding all but these two.

Two macros are defined to inquire about the state of this flag:

```
#define win_inputnegevent(ie)
#define win_inputposevent(ie)
   struct      inputevent *ie;
```

These are TRUE or FALSE if the IE_NEGEVENT bit is 1 or 0 respectively in the input event pointed to by *ie*.

### 5.1.4. Shift Codes

*Ie_shiftmask* contains a set of bit flags which indicate an interesting state when an input event occurs. The most obvious example is the state of the Shift or Control keys when some other key is pressed. Eventually, clients will be able to declare any Virtual Input switch as an "interesting" shift switch. For now, only the following bits are reported:

```
#define  CAPSMASK      0x0001
#define  SHIFTMASK     0x000E
#define  CTRLMASK      0x0030
#define  UPMASK        0x0080
```

These are defined in */usr/include/sundev/kbd.h*, and described in *kbd*(5).

## 5.2. Reading Input Events

A library routine exists for reading the next input event for a window:

```
int input_readevent(fd, ie)
   int         fd;
   struct      inputevent *ie;
```

This fills in the indicated struct, and returns 0 if all went well. In case of error, it sets the global variable *errno*, and returns –1; the client should check for this case.

A window can be set to do blocking or non-blocking reads via a standard *fcntl* system call, as described in *fctnl*(2) and *fcntl*(5). A window is defaulted to blocking reads. The blocking status of a window can be determined by the *fcntl* system call.

The recommended normal style for handling input uses blocking I/O and the *select*(2) system call to await both input events and signals such as SIGWINCH. This allows a signal handler to merely set a flag, and leave substantial processing to be performed synchronously when the *select* returns. The *tool_select* mechanism described in chapter 7 illustrates this approach. Using blocking I/O and *read*(2) without a prior *select* forces the client to process SIGWINCHes entirely in the asynchronous interrupt handler. This necessitates extra care to avoid race conditions and other asynchronous errors.

Non-blocking I/O may be useful in a few circumstances. For example, when tracking the mouse with an image which requires significant computation, it may be desirable to ignore all but the last in a queued sequence of motion events. This is done by reading the events, but not processing them until a non-motion event is found, or until all events are read. Then the most recent mouse location is displayed, but not all the points covered since the last display. When all events have been read and the window is doing non-blocking I/O, *input_readevent* returns -1 and the global variable *errno* is set to EWOULDBLOCK.

## 5.3. Input Serialization and Distribution

With the exception of some of the pseudo event codes, the Virtual Input Device described in preceding sections is not logically tied to the Sun window system; the scheme could be used by any system desiring that form of unification. This section is more specific to the window system, since it discusses how events are selected and distributed among the various windows which might use them.

Each user input event is formatted into an *inputevent*, which is then assigned to some recipient. There are three ways a process gets to receive an input event:

- Most commonly, it reads the window which lies under the cursor, and that window has an *input mask* which matches the event. Input masks are described in *Input Masks*. If several windows are layered under the cursor, the event is tested first against the input mask of the topmost window.
- If the event does not match the input mask of one window, other windows will be given a chance at it, as described below.
- Much less frequently, a window will be made the recipient of *all* input events; this is discussed under *win_grabio*, in section 5.3.2 below.

Each window designates another window to be offered events which the first will not accept. By default this is the window's parent; another backstop may be designated in a call to *win_setinputmask*, described in the next section. If an event is offered unsuccessfully to the root window, it is discarded. Windows which are not in the chain of designated recipients never have a chance to accept the event.

If a recipient is found, the locator coordinates are adjusted to the coordinate system of the recipient, and the event is appended to the recipient's input stream. Thus, every window sees a single stream of input events, in the order in which the events happened (and time-stamped, so that the intervals between events can also be computed), and including only the events that window has declared to be of interest.

### 5.3.1. Input Masks

The input masks facilitate two things:

- Events can be accepted or rejected by classes; for instance, a process may want only ASCII characters.
- The times when events are accepted can be controlled, minimizing the processing required to accept and ignore uninteresting events. For instance, a process may track the mouse only when it is inside one of its windows, or when one of the mouse buttons is down.

Clients specify which input events they are prepared to process by setting the input mask for each window being read.

```
struct inputmask {
    short       im_flags;
    char        im_inputcode[IM_CODEARRAYSIZE];
    short       im_shifts;
    short       im_shiftcodes[IM_SHIFTARRAYSIZE];
};
#define IM_CODEARRAYSIZE  (VKEY_CODE/((sizeof char)*BITSPERBYTE))
#define IM_SHIFTARRAYSIZE ((sizeof short)*BITSPERBYTE)
```

*Im_flags* specifies the handling of related groups of input events:

```
#define IM_UNENCODED
```

indicates that no translation of physical device events should be performed. The Virtual Input Device should not intervene between the window and the user input. In this case, the most significant byte of *ie_code* in an input event is the id number of the device that generated the event, and the least significant byte contains the physical keystation number of the keystation that the user struck. The current device ids are those assigned to the supported keyboards and the id assigned to the mouse

```
#define MOUSE_DEVID 127
```

For unencoded mouse input, the least significant byte of the event code is identical to the least significant byte of the corresponding encoded input event. Note that unencoded pseudo events are associated with the physical locator; that is, a button-push on a tablet puck will generate a different code from a corresponding button-push on a mouse.

```
#define IM_ASCII
```

indicates that the Virtual Input Device translation should occur.

```
#define IM_ANSI
```

indicates that the process wants keystrokes to be interpreted as ANSI characters and escape sequences: normal ASCII characters are represented by their ASCII code in *ie_code*, described in *Uniform Input Events*. Function keys with a standard interpretation, such as the cursor control keys, are represented by a sequence of input events, whose *ie_codes* are ASCII characters starting with ESC. See *kbd*(5) for further details.

```
#define IM_POSASCII
```

indicates that the client only wants to be notified of positive events for ASCII class events, even though IM_NEGEVENT is enabled.

*Note:* The current implementation automatically enables both IM_ANSI and IM_POSASCII when IM_ASCII is specified.

Requesting a particular function event in addition turns off any ANSI escape-coding for that function event.

```
#define IM_META
```

indicates that META-translation should occur. This means ASCII events that occur while the META key is depressed are reported with codes in the META range. Note that IM_META does not make sense unless IM_ASCII is enabled.

```
#define IM_NEGEVENT
```

indicates that the client wants to be notified of negative events as well as positive ones. See *Event Flags* for a discussion of positive and negative events.

*Im_inputcode* is an array of bit flags indexed by biased event codes. A 1 in the *ith* position of the bit array indicates that the event with code VKEY_FIRST+ *i* should be reported. This filter applies in both IM_UNENCODED and IM_ASCII modes.

There are two routines which are of interest here.

```
win_setinputmask(windowfd, acceptmask, flushmask, designee)
    int         windowfd;
    struct      inputmask *acceptmask, *flushmask;
    int         designee;
```

sets the input mask for the window identified by *windowfd*. *Acceptmask* addresses the new mask — events it passes will be reported to this window after the call to *win_setinputmask*.

*Flushmask* specifies a set of events which should be flushed from this window's input queue. These are events which were accepted by the previous mask, and have already been generated, but not read, by this window. This is a dangerous facility; type-ahead and mouse-ahead will often be lost if it is used. The most obvious application is for confirmations, but these can be better implemented by requiring the confirmation within a short time-out.

*Note:* If *flushmask* is non-NULL, the current implementation flushes all events from the queue, not just those specified in *flushmask*.

*Designee* is the window number, which specifies the next potential recipient for events rejected by this window. If it is set to WIN_NULLLINK (defined in */usr/include/sunwindow/win_struct.h*), it is interpreted as designating the window's parent.

*Note:* Changing masks in response to some input should be done with caution. There will be a lapse of time between the event which persuades the client it wants a new mask and the time the system interprets the resulting call to *win_setinputmask*. Events which occur in this interval will be passed or discarded according to the old input mask. Thus, it is probably not appropriate to wait for a button down before requesting the corresponding button-up; the button-up may arrive and be discarded before the mask is changed. It's less dangerous to wait until a button goes down to start tracking the mouse, since the client will be caught up as soon as the first motion event arrives. But even here, it's better to ask for the LOC_MOVEWHILEBUTDOWN event, and never change the mask.

The input mask for a window is read with

```
win_getinputmask(windowfd, im, designee)
    int         windowfd;
    struct      inputmask *im;
    int         *designee;
```

The input mask for the window identified by *windowfd* is copied into the buffer addressed by im. The number of the window that is the next possible recipient of input is copied into the int addressed by designee.

We return to *win_input.h* for these routines useful for manipulating input masks. The first three are macros:

```
#define win_setinputcodebit(im,code)
    struct      inputmask *im;
    char        code;
```

sets the bit indexed by *code* in the input mask addressed by *im* to 1.

```
#define win_unsetinputcodebit(im,code)
    struct      inputmask *im;
    char        code;
```

resets the bit to zero. The routine:

```
#define win_getinputcodebit(im, code)
    struct      inputmask *im;
    char        code;
```

returns non-zero if the bit indexed by *code* in the input mask addressed by *im* is set.

```
input_imnull(mask)
    struct      inputmask *mask;
```

is a procedure which initializes an input mask to all zeros. It is critical to initialize the input mask explicitly when the mask is defined as a local procedure variable.

### 5.3.2. Seizing All Inputs

Normally, input events are directed to the window which underlies the cursor at the time the event occurs. Two procedures modify that behavior. A window may temporarily seize all inputs by calling:

```
win_grabio(windowfd)
    int         windowfd;
```

The caller's input mask still applies, but it receives input events from the whole screen; no window other than the one identified by *windowfd* will be offered an input event or allowed to write on the screen after this call.

```
win_releaseio(windowfd)
    int         windowfd;
```

undoes the effect of a *win_grabio*, restoring the previous state.

## 5.4. Event Codes Defined

In the following table are collected together all of the special event code names discussed above. These names define values which appear in the *ie_code* field of an *inputevent*. As the system evolves, the particular value bound to a name is likely to change, thus event codes should be compared to the symbolic names below, not to the current values of those names.

```
#define ASCII_FIRST                 (0)
#define ASCII_LAST                  (127)
#define META_FIRST                  (128)
#define META_LAST                   (255)

#define VKEY_CODES                  (128)
#define VKEY_FIRST                  (32512)

#define VKEY_FIRSTPSEUDO            (VKEY_FIRST)
#define LOC_MOVE                    (VKEY_FIRSTPSEUDO+ 0)
#define LOC_STILL                   (VKEY_FIRSTPSEUDO+ 1)
#define LOC_WINENTER                (VKEY_FIRSTPSEUDO+ 2)
#define LOC_WINEXIT                 (VKEY_FIRSTPSEUDO+ 3)
#define LOC_MOVEWHILEBUTDOWN        (VKEY_FIRSTPSEUDO+ 4)
#define VKEY_LASTPSEUDO             (VKEY_FIRSTPSEUDO+ 15)

#define VKEY_FIRSTFUNC              (VKEY_LASTSHIFT+ 1)

#define BUT_FIRST                   (VKEY_FIRSTFUNC)
#define BUT(i)                      ((BUT_FIRST)+ (i)-1)
#define BUT_LAST                    (BUT_FIRST+ 9)

#define KEY_LEFTFIRST               ((BUT_LAST)+ 1)
#define KEY_LEFT(i)                 ((KEY_LEFTFIRST)+ (i)-1)
#define KEY_LEFTLAST                ((KEY_LEFTFIRST)+ 15)

#define KEY_RIGHTFIRST              ((KEY_LEFTLAST)+ 1)
#define KEY_RIGHT(i)                ((KEY_RIGHTFIRST)+ (i)-1)
#define KEY_RIGHTLAST               ((KEY_RIGHTFIRST)+ 15)

#define KEY_TOPFIRST                ((KEY_RIGHTLAST)+ 1)
#define KEY_TOP(i)                  ((KEY_TOPFIRST)+ (i)-1)
#define KEY_TOPLAST                 ((KEY_TOPFIRST)+ 15)

#define KEY_BOTTOMLEFT              ((KEY_TOPLAST)+ 1)
#define KEY_BOTTOMRIGHT             ((KEY_BOTTOMLEFT)+ 1)

#define VKEY_LASTFUNC               (VKEY_FIRSTFUNC+ 101)

#define VKEY_LAST                   VKEY_FIRST+ VKEY_CODES-1
```

There are 3 synonyms for the common case of a 3-button mouse:

```
#define MS_LEFT         BUT(1)
#define MS_MIDDLE       BUT(2)
#define MS_RIGHT        BUT(3)
```

# Chapter 6

# Suntool: Tools and Subwindows

This chapter introduces the third and highest level of SunWindows, *suntools*. It discusses how to write a tool: it covers creation and destruction of a tool and its subwindows, the strategy for dividing work among them, and the use of routines provided to accomplish that work.

At the *suntools* level, the lower-level facilities are actually used to build user interfaces. This chapter also describes a model for building applications, a number of components that implement commonly-needed portions of such applications, an executive and operating environment that supports that model, and some general-purpose utilities that can be used in this and similar environments.

We refer to an application program that is a client of this SunWindows level as a *tool*. *Tool* covers the one or more processes that do the actual application work. This term also refers to the collection of typically several windows through which the tool interacts with the user. Simple tools might include a calculator, a bitmap editor, and a terminal emulator. Sun Microsystems provides a few ready-built tools, several of which are illustrated in Appendix B. Others may be developed to to suit particular needs.

Common SunWindows tool components and their functions include:

- An executive framework that supplies the usual "main loop" of a program, and which coordinates the activities of the various subwindows;
- A standard *tool window* that frames the *subwindows* of the tool, identifying it with a name stripe at the top and borders around the subwindows. Each tool window can adjust its size and position, including layering, and subwindow boundary movement.
- Several standard subwindows that can be instantiated in the tool;
- A standard scheme for laying out those subwindows; and
- A facility that provides a default *icon*, which is a small form the tool takes to be unobtrusive but still identifiable.

The *suntools* program initializes and oversees the window environment. It provides for:

- automatic startup of a specified collection of tools;
- dynamic invocation of standard tools;
- management of the window, called the *root* window, which underlies all tools and paints a simple solid color;
- the user interface for leaving the window system.

Users desiring another interface to these functions can replace the *suntools* program, while retaining specific tools.

The procedures that support the facilities described in this chapter and the following two are in the *suntool* library, */usr/lib/libsuntool.a*). These procedures and their data structures are declared in a number of distinct header files, all of which can be included in */usr/include/suntool/tool_hs.h.*

## 6.1. Tools Design

A typical tool is built as a *tool window*, and contained within that, a set of *subwindows*, which incorporate most of the user interface to the tool's facilities. Each subwindow is a "window" in the sense described in *Window Manipulation*; the subwindows form a tree rooted at the tool window, and the various tool windows are all children of the *root* window associated with the screen.

### 6.1.1. Non-Pre-emptive Operation

In general, tools should be designed to function in a *non-pre-emptive* style: they should wait without consuming resources until given something to do, perform the task expeditiously, and promptly return control to the user. If some task requires extensive processing, a separate process should be forked to run it without blocking the user interface.

This non-pre-emptive style implies that the tool is built as a set of independent procedures, which are invoked as appropriate by a standardized control structure. The basic advice to client programs is, "Wait right there; we'll let you know as soon as we have something for you to do." From a programming point of view, the main function that the tool mechanism provides is the provision of the control structure to implement this non-pre-emptive programming style. The tool window and its subwindows all have the same interface to this control mechanism.

### 6.1.2. Division of Labor

The tool window performs a few functions directly. These are the user interface functions, which are common to all tools.

Subwindows are the workhorses of the *suntool* environment, but most of the work they do is specific to their own tasks, and of little interest here. It is important to understand that a subwindow corresponds to a data type: there will be many instantiations of particular subwindows, quite possibly several in a single tool.

Various types of subwindows are developed as separate packages that can be assembled at a high level. In addition to programmer convenience, this approach promotes a consistent user interface across applications.

The remainder of this chapter divides a tool's existence into two large areas: creation and destruction, and tool-specific aspects of processing.

## 6.2. Tool Creation

All of the following processing must be done as a tool is started:

- Parameters for this invocation of the tool must be passed to it. Every tool must be given the name of its *parent window*; other parameters that may be given to the tool include a position for it on the screen, whether it should be open or iconic, specification of data files,

such as fonts, to be used in this invocation, and initializations to be performed.

- The tool should be given its own process and process group. In contrast to the usual procedure in which a program is invoked under the shell, the parent process should generally be allowed to proceed before the child exits.
- The tool window should be created with space allocated for it and its various options defined; similarly, its subwindows should be created and positioned in the tool window.
- The UNIX signal system should be initialized to pass appropriate signals to the tool.
- The tool's window should be installed into the display structure.
- Finally, the tool may start its normal processing.

### 6.2.1. Passing Parameters to the Tool

There are at least three ways parameters may be passed to a tool that is starting up:

- Command-line arguments.
- Relatively stable options may be stored in a file like a user profile.
- Environment parameters may be used for well-established values. They have the valuable property that they can communicate information across several layers of processes, not all of which have to be involved.

The first two parameters passing mechanisms need no special attention here, since they are used just as in non-window UNIX programs. However, SunWindows itself uses a few environment variables for tool startup. WINDOW_PARENT is set to a string that is the device name of a window's parent; for a tool, this will usually be the name of the root window of the window system. WINDOW_INITIALDATA is set to the coordinates of two rectangles plus one flag. The rectangles are the regions for the window while open and closed, and the flag is a boolean that is non-zero if the tool should start out iconic.

```
we_setparentwindow(windevname)
    char        *windevname;
```

sets WINDOW_PARENT to *windevname*.

```
int we_getparentwindow(windevname)
    char        *windevname;
```

gets the value of WINDOW_PARENT into *windevname*. The length of this string should be at least WIN_NAMESIZE characters long, a constant found in */usr/include/sunwindow/win_struct.h*. A non-zero return value means that the WINDOW_PARENT parameter couldn't be found.

The process that is starting the tool should set WINDOW_INITIALDATA before it forks (*wmgr_forktool* does this; see *Suntools: User Interface Utilities*). After the fork, the newborn tool may interrogate these variables. The routines to do this are in the library */usr/lib/libsunwindow.a*.

```
we_setinitdata(rnormal, riconic, iflag)
    struct      rect *rnormal, *riconic;
    int         iflag;
```

sets the environment variable in the parent process, and

```
we_getinitdata(rnormal, riconic, iflag)
   struct      rect *rnormal, *riconic;
   int         *iflag;
```

reads those values in the child process. A non-zero return value means that the WINDOW_INITIALDATA parameter couldn't be found.

A procedure is provided for unsetting WINDOW_INITIALDATA for tools that are going to provide windows for other processes to run in. This procedure prevents a wayward child process from being confused by the an incorrectly set variable:

```
we_clearinitdata()
```

## 6.2.2. Forking the Tool

A tool will normally have its own process. The creation of that process does not differ significantly from the normal paradigm for process creation. If it is to be started by a menu command or some other procedural interface, it is appropriate for the creating process to do the fork and return from the procedure call. When the child process dies, the parent process should catch the SIGCHLD signal and clean up. See the *wait*(2) system call. SIGCHLD indicates to a parent process that a child process has changed state.

## 6.2.3. Creating the Tool Window

The pair of procedures *tool_create* and *tool_createsubwindow* carry out the main work of creating a tool with its subwindows. These take a series of parameters that define the object to be created, and return a pointer to an object that encapsulates the information about the tool or its subwindow. That pointer is then passed to a number of other routines that manipulate the object; the client is usually not concerned with the exact definition of the structure.

These create routines include a large part of the processing described in the earlier parts of this manual, so that client programmers need not necessarily concern themselves much with the details of *pixrects* and *pixwins*.

A tool is created by a call to:

```
struct tool *tool_create(name, flags, normalrect, icon)
   char        *name;
   short       flags;
   struct      rect *normalrect;
   struct      icon *icon;

#define  TOOL_NAMESTRIPE          0x01
#define  TOOL_BOUNDARYMGR         0x02
```

*name* is the name of the tool. This is what will be displayed in the tool's name stripe if TOOL_NAMESTRIPE is set in the flag's argument. It also appears on the default icon.

*flags* has the flags TOOL_NAMESTRIPE and/or TOOL_BOUNDARYMGR set as those properties are desired. (TOOL_BOUNDARYMGR enables boundaries that the user can move between subwindows.)

*Normalrect* describes the initial position and size of the tool in its normal open state in the coordinate system of the tool's parent, which is typically the window for the screen.

*icon* is a pointer to an *icon* struct, if the client wants a special icon.

*Normalrect* and the *icon* may be defaulted by passing NULL for their arguments. The default icon is described, along with considerations for making custom icons, in *Suntool: User Interface Utilities*; the choice is strictly a matter of convenience vs. ambition. A tool's starting position should almost always be left NULL; it could be the result of WE_GETINITDATA that is going into *normalrect*.

Creating the tool does not cause it to appear on the screen; a separate step is used for that purpose after the full tool structure is constructed, as described in *Tool Installation*. Most tool programmers can skim the following information to *Subwindow Initialization* and ignore the details of the *tool* and *toolsw* data structures.

## 6.2.4. The Tool Struct

The tool struct is defined in */usr/include/suntool/tool.h*. It is:

```
struct tool {
    short       tl_flags;
    int         tl_windowfd;
    char        *tl_name;
    struct      icon *tl_icon;
    struct      toolio tl_io;
    struct      toolsw *tl_sw;
    struct      pixwin *tl_pixwin;
    struct      rect tl_rectcache;
}
```

*Tl_flags* holds state information. Currently, there are 6 defined flags:

```
#define TOOL_NAMESTRIPE          0x01
#define TOOL_BOUNDARYMGR         0x02
#define TOOL_ICONIC              0x04
#define TOOL_SIGCHLD             0x08
#define TOOL_SIGWINCHPENDING     0x10
#define TOOL_DONE                0x20
```

Their actions are as follows:

TOOL_NAMESTRIPE
: indicates that the tool is to be displayed with a black stripe holding its name at the top of its window.

TOOL_BOUNDARYMGR
: enables the option that allows the user to move inter-subwindow boundaries.

TOOL_ICONIC
: indicates the current state of the tool: 1 = small (*iconic*); 0 = normal (*open*).

TOOL_SIGCHLD and

TOOL_SIGWINCHPENDING
: mean that the tool has received the indicated signal and has not yet performed the processing to deal with it.

TOOL_DONE
indicates the tool should exit the *tool_select* notification loop.

The last three flags are used during *tool_select* processing described below and should be considered private to the tool implementation.

*Tl_windowfd* holds the file descriptor for a tool's window. This is used for both input and output. It also identifies the window for manipulations on the window database, such as modifying its position or shape. *Windowfd*s' uses are discussed in chapters 3 through 5.

*Tl_name* addresses the string that can be displayed in the tool's namestripe and default icon.

*Tl_rectcache* holds a rectangle that indicates the size of the tool's window. Because the rectangle is in the tool's coordinate system, the origin will always be (0, 0). This size information is cached so that the tool can tell when its size has changed by comparing the cached rect with the current rect.

*Tl_icon* holds a pointer to the icon struct for this tool.

*Tl_pixwin* addresses the window's pixwin, which is the structure through which the tool accesses the display.

*Tl_sw* points to the first and oldest of the tool's subwindows. The following section discusses these structs.

The tool uses *tl_io* to control notification of input and window change events to itself. *Toolio Structure* details this structure type. During tool creation, the fields of this structure are set up with values to do default tool processing.

## 6.2.5. Subwindow Creation

After the tool is created, its subwindows are added to it.

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

#define TOOL_SWEXTENDTOEDGE -1
```

makes a new subwindow, adds it to the list of subwindows for the indicated *tool*, and returns a pointer to the new *toolsw* struct. The *width* and *height* parameters are hints to the layout mechanism indicating what size the windows should be if there is enough room to accommodate them. There are no guarantees about maintaining subwindow size because changing window sizes can ruin any scheme. TOOL_SWEXTENDTOEDGE may be passed for *width* and/or *height*; it allows the subwindow to stretch with its parent in either or both directions. *Subwindow Layout* details the subwindow layout. The *name* is currently unused; it may eventually support the capability to refer to subwindows by name.

The remaining subwindow initialization requires reference to the data structure:

```
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;
};
```

The subwindows of a tool are chained on a list with *ts_next* in one subwindow pointing to the next in line, until the list is terminated with a null pointer.

Like the tool window, each subwindow must have an associated open window device; *tool_createsubwindow* stores the file descriptor in *ts_windowfd*.

*Ts_name*, *ts_width* and *ts_height* are exactly as in the call to *tool_createsubwindow*.

The tool uses *ts_io* to control notification of input and window change events to the subwindow. Upon subwindow creation, the *ts_io* structure has null values in it that need to be set. This is normally done by the *create* routine for a standard subwindow type. *Toolio Structure* details this structure.

*Ts_destroy* gets called when the tool is being destroyed by *tool_destroy* so that the subwindow may terminate cleanly.

*Ts_data* provides 32 bits of uninterpreted data private to the subwindow implementation. Typically, it will be a pointer to information for this instance of the subwindow. That is, all subwindows of the same type will share common interrupt handlers and layout characteristics. Window contents and other information specific to one particular window will all be accessed through this pointer. This is discussed at more length in *Requirements for Subwindows* in Chapter 7.

## 6.2.6. Subwindow Layout

By default, subwindows are laid out in their tool's area in a simple left-to-right, top-to-bottom fashion, in the order they are created. A subwindow is placed as high as it can be, and in that space, as far to the left as it can be.

Subwindows that should be arranged in a more controlled fashion may be rearranged after they have all been created, using the rectangle manipulation facilities described in *Window Geometry*. Three functions return numbers useful to tools doing their own subwindow layout explicitly:

```
short tool_stripeheight(tool)
    struct      tool *tool;
```

returns the height in pixels of the tool's name stripe.

```
short tool_borderwidth(tool)
    struct      tool *tool;
```

returns the width in pixels of the tool's outside border.

```
short tool_subwindowspacing(tool)
    struct      tool *tool;
```

returns the number of pixels that should be left as a margin between subwindows of a tool, currently the same as the outside border of the tool.

## 6.2.7. Subwindow Initialization

By the time *tool_createsubwindow* has returned, the subwindow is already inserted in the tree growing out of the tool window; however, the subwindow will not perform any interesting function until *ts_io* and *ts_data* have been initialized. Normally, *tool_createsubwindow* is not directly called. Instead, the tool subwindow creation procedure for a subwindow type is called. The subwindow specific routine will call *tool_createsubwindow* and then initialize *ts_io* and *ts_data*.

## 6.2.8. Tool Installation

Once the tool is created and its subwindows have been created and installed, the software interrupt system should be turned on via a call to *signal* as described in *Window Change Notifications*. At least SIGWINCH should be caught; if there are inferior processes in any of the subwindows, SIGCHLD should be added with any others as appropriate. Finally, the tool is installed into the display window tree by a call to:

```
tool_install(tool)
    struct      tool *tool;
```

At this point, the tool is operating; in fact, it will probably shortly receive a SIGWINCH asynchronously to paint its window(s) for the first time.

## 6.2.9. Tool Destruction

Explicitly destroying a tool as it reaches the end of its processing allows the system to reclaim resources and remove the windows gracefully. The procedure to invoke this cleanup is:

```
tool_destroy(tool)
    struct      tool *tool;
```

*Tool_destroy* will destroy every subwindow of the indicated tool as part of its processing, so the subwindows need not be destroyed explicitly. Each subwindow's *ts_destroy* procedure gets called, so they can clean up gracefully. The pointer passed to *tool_destroy* must never be dereferenced after that call, since it is no longer valid.

A single subwindow can be destroyed by an explicit call to:

```
tool_destroysubwindow(tool, subwindow)
    struct      tool *tool;
    struct      toolsw *subwindow;
```

A tool may use this procedure to change its subwindows, while continuing to run.

## 6.3. Tool Processing

The main loop of a normal tool is encapsulated inside a call to:

```
tool_select(tool, waitprocessesdie)
    struct      tool *tool;
    int         waitprocessesdie;
```

This procedure is the notification distributer used for event-driven program control flow. When some input event, timeout or signal interrupt is detected inside *tool_select*, a call to a notification handler is made, passing in the *toolio* structures of the tool and its subwindows. When the handler returns, *tool_select* awaits another event. The *waitprocessesdie* argument is discussed below in *Child Process Management.*

### 6.3.1. Toolio Structure

The *toolio* data structure in each *toolsw* structure holds what is needed for a subwindow to wait for something to happen in the *tool_select* call. The *tool* structure uses the *toolio* data structure within itself to wait for input too. It is defined in */usr/include/suntool/tool.h.*

```
struct toolio {
    int         tio_inputmask,
    int         tio_outputmask,
    int         tio_exceptmask;
    struct      timeval *tio_timer;
    int         (*tio_handlesigwinch) ();
    int         (*tio_selected) ();
};
```

*Tio_inputmask, tio_outputmask, tio_exceptmask* and *tio_timer* fields are analogous to the last four arguments to the *select* system call. *Tio_inputmask* has the bit "1<<*f*" set for each file descriptor *f* on which a window wants to wait for input. Similarly, *tio_outputmask* and *tio_exceptmask* indicate an interest in *f* being ready for writing and having an exceptional condition pending, respectively. There are currently no "exceptional conditions" implemented; this field provides compatibility with the *select* system call.

If *tio_timer* is a non-zero pointer, it specifies a maximum interval to wait for one of the file descriptors in the masks to require attention. If *tio_timer* is a zero pointer, an infinite timeout is assumed. To effect a poll, the *tio_timer* argument should be non_zero, pointing to a *timeval* structure with all zero fields.

*Toolio* also contains pointers to the procedures that are called when the tool has received some notification. *Tio_handlesigwinch* addresses the procedure that responds to the SIGWINCH signal. This procedure handles repaint requests and window size changes. The general form for such a procedure is:

```
sigwinch_handler(data)
    caddr_t     data;
```

Such procedures take a single argument *data* whose type is context-dependent. For a tool this data is a pointer to the *tool* structure. For a subwindow this data is the *ts_data* value in the *toolsw* structure.

*Tio_selected* addresses the procedure which responds to notifications from the *select* system call. The procedure's calling sequence is:

```
io_handler(data, ibits, obits, ebits, timer)
    caddr_t     data;
    int         *ibits,
    int         *obits,
    int         *ebits,
    struct      timeval **timer;
```

In such procedures, the *data* argument is like that of the SIGWINCH handlers described above. The three integer pointers indicate which file descriptors are ready for reads (**ibits*), writes (**obits*), or exception-handling (**ebits*). If *timer* is NULL, this window was not waiting on any timeout. If **timer* points to a valid struct *timeval* then this window is waiting for a timeout. If both the *(*timer)->tv_sec* and *(*timer)->tv_usec* are zero, the timeout has just happened for this window and should be serviced. The data in the file descriptor masks is not defined if a timeout has occurred.

Before returning from a procedure of this type, the masks and timer must be reset by storing through the pointers passed in the arguments; the values should be consistent with the discussion of the masks and timer pointer above. You may not want to reset the timer if you are using it as a countdown timer, and it still has time remaining on it.

## 6.3.2. File Descriptor and Timeout Notifications

*Tool_select* generates three composite masks from each of the three *toolio* structures in the tool. The input mask is special in that if all the masks in a particular *toolio* structure are zero, an entry in the composite input mask is made for the associated window anyway. *Tool_select* also determines the shortest timeout that any of the windows is waiting on. The composite masks and shortest timeout are passed to the *select* system call.

When the *select* system call returns normally, windows that have a match between their masks and the mask of ready file descriptors that have timed out, are notified via their *tio_selected* procedure. The *tio_selected* procedures are called with the complete ready masks, not just the intersection of its own masks and the ready masks. However, a *tio_selected* procedure is called with its own window's timer value.

Each window that has been selected as a result of the *select* system call is notified. The order of notification is not defined. Problems will arise if there are multiple non-cooperating windows waiting on the same device.

It should be noted that timers in this implementation are only approximate. When the *select* system call returns and a timeout hasn't occurred, the *select* is assumed to have been instantaneous. Also, the time taken up with handling notifications is not deducted from the timers.

## 6.3.3. Window Change Notifications

Clients of the tool interface must catch the SIGWINCH signal. A signal catcher can be set up via the *signal(3)* library call. That catcher is then responsible for notifying the tool package that the signal has arrived. This is done by calling:

```
tool_sigwinch(tool)
    struct      tool *tool;
```

This procedure simply sets the TOOL_SIGWINCHPENDING flag in *tool*. The receipt of any signal has the side effect of causing the *select* system call in *tool_select* to return abnormally.

The TOOL_SIGWINCHPENDING flag is noticed and the tool's *tio_handlesigwinch* procedure is called. The default *tio_handlesigwinch* procedure does some processing, which may include changing the subwindow layout, and eventually calls all its subwindows' *tio_handlesigwinch* procedures.

### 6.3.4. Child Process Maintenance

*Tool_select* also gathers up dead children processes of the tool. The *waitprocessesdie* argument to *tool_select* is provided for tools which have separate processes behind some of their subwindows. Such tools must explicitly catch SIGCHLD, the signal that indicates to a parent process that a child process has changed state. Then the signal handler, parallel to a SIGWINCH catcher and *tool_sigwinch*, should call:

```
tool_sigchld(tool)
    struct      tool *tool;
```

This call causes *tool_select* to try to gather up a dead child process via a *wait3* system call (see *wait*(2)). When as many child processes have been gathered up as indicated by the *waitprocessesdie* argument to *tool_select*, *tool_select* returns.

### 6.3.5. Changing the Tool's Image

During processing, a call to:

```
tool_display(tool)
    struct      tool *tool;
```

redisplays the entire tool. This is useful if some change has been made to the image of the tool itself, for instance if its name or its icon's image have been changed. Normal repaints in response to size changes or damage should not use this procedure. They will be taken care of by SIGWINCH events and their handlers.

### 6.3.6. Terminating Tool Processing

During the time that *tool_select* is acting as the main loop of the program, a call to:

```
tool_done(tool)
    struct  tool *tool;
```

causes the flag TOOL_DONE to be set in *tool*. *Tool_select* notices this flag, and then returns gracefully.

### 6.3.7. Replacing Toolio Operations

Since the *toolio* structure contains procedure pointers in variables, it is possible to customize the behavior of a window by replacing the default values.

Icons that respond to user inputs or that update their image in response to timer or other events, may be implemented by replacing the tool's *tool_selected* procedure. A different subwindow layout scheme may be implemented in a replacement procedure for *tio_handlesigwinch*. Note that these modifications do not require changes to existing libraries; the address of the

substitute routine is simply stored in the appropriate slot at run-time. However, the substitute routine must either do all of the processing handled by the original library routine, or the substitute routine should do its special processing and then call the original library routine.

# Chapter 7

# Suntool: Subwindow Packages

This chapter describes *subwindow packages*, the building blocks for constructing a *tool*. It presents a guide for building new subwindow packages of general utility and describes the available standard subwindow packages for use with *suntools*. Refer to *Suntool: Tools and Subwindows* for a description of the overall structure of tools and the general notion of a subwindow.

Subwindows, as presented here, are designed to be independent of the particular framework in which they are used. That is, a subwindow is a merger of window handling and application processing which should be valid in frameworks other than the *tool* structure and *suntool* environment described in the preceding chapter. The design avoids any dependence on those constructs. Thus, a subwindow package can be used in another user interface system written on top of the *sunwindow* basic window system. However, subwindow packages all provide a utility for creating a subwindow in the *tool* context.

## 7.1. Minimum Standard Subwindow Interface

This section describes the minimum programming interface one should define when writing a new subwindow package. A subwindow implementation should provide all the facilities described here. This section presents the arguments to the following standard procedures. Each subwindow package need only document any additional arguments passed to its *create/init* procedures. There is a set of naming conventions that provides additional consistency between subwindow package interfaces.

For the purpose of example, we use *foo* as the prefix. Other prefixes used in existing subwindow packages include *tty*, *gfx* and *msg*.

Each subwindow package has a structure definition that contains all the data required by a single instance of the subwindow.

```
struct foosubwindow {
    int         fsw_windowfd;
    struct      pixwin *fsw_pixwin;
    ...
};
```

The structure definition typically has a *pixwin* for screen access and a window handle for identification as part of this data. The information that the subwindow's procedures need should be stored in this data structure; this may entail redundantly storing some data that is in the associated containing data structure, such as the *toolsw* struct. Having an object per subwindow allows multiple instantiations of a subwindow package in a single-user process. The following struct creates new instances of a foo subwindow:

```
struct foosubwindow *foosw_init(windowfd, ...)
    int         windowfd;
```

*Windowfd* is to be a foo subwindow. The "..." indicates that many subwindow packages will require additional set-up arguments. This routine typically opens a *pixwin*, sets its input mask as described in *Input to Application Programs*, and dynamically allocates and fills the subwindow's data object. If the returned value is NULL then the operation failed.

```
foosw_done(foosw)
    struct      foosubwindow *foosw;
```

destroys subwindow instance data. Once this procedure is called, the *foosw* pointer should no longer be referenced.

```
foosw_handlesigwinch(foosw)
    struct      foosubwindow *foosw;
```

This procedure handles repaint requests and must also detect and deal with changes in the window size. It is called as an eventual result of some other procedure catching a SIGWINCH.

```
foosw_selected(foosw, ibits, obits, ebits, timer)
    struct      foosubwindow *foosw;
    int         *ibits,
    int         *obits,
    int         *ebits,
    struct      timeval **timer;
```

handles event notifications. Subwindow packages that don't accept input may not have a procedure of this type. The semantics of this procedure are fully described in the preceding chapter in the section entitled *Toolio Structure.*

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

creates a struct *toolsw* that is a foo subwindow. *Foosw_createtoolsubwindow* is only applicable in the *tool* context. It is often the only call that an application program need make to set up a subwindow of a given type. *Tool* is the handle on the tool that has already been created. *Name* is the name that you want associated with the subwindow. *Width* and *height* are the dimensions of the subwindow as wanted by the *tool_createsubwindow* call. The "..." indicates that many subwindow packages will require additional arguments. These additional arguments should parallel those in *foosw_init*. If the returned value is NULL then the operation failed.

*Foosw_createtoolsubwindow* takes the window file descriptor it gets from *tool_createsubwindow*, passes it to *foosw_init*, and stores the resulting pointer in the tool subwindow's *ts_data* slot. The addresses of *foosw_handlesigwinch* and *foosw_selected* are stored in the appropriate slots of the *toolio* structure for the tool subwindow, and the address of *foosw_done* is stored in the tool subwindow's *ts_destroy* procedure slot.

Of course, most subwindow packages define functions that perform application-specific processing; the ones described here are merely the permissible minimum.

## 7.2. Empty Subwindow

The empty subwindow package simply serves as a place holder. It does nothing but paint itself gray. It expects the window it is tending to be taken over by another process as described in *Graphics Subwindow*. When the other process is done with the empty subwindow package, the caretaker process resumes control.

A private data definition that contains instance-specific data defined in */usr/include/suntool/emptysw.h* is:

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

*Em_windowfd* is the file descriptor of the window that is tended by the empty subwindow. *Em_pixwin* is the structure for accessing the screen.

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

sets up an empty subwindow in a tool window. If the returned value is NULL then the operation failed. Since *esw_createtoolsubwindow* takes care of setting up the empty subwindow, the reader may not be interested in the remainder of this section.

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

creates a new instance of an empty subwindow. *Windowfd* is the window to be tended. If the returned value is NULL then the operation failed.

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

handles SIGWINCH signals. If the process invoking this procedure is the current owner of *esw->em_windowfd*, gray is painted in the window. If it is not the current owner, it checks to see if the current owner is still alive. If the current owner is dead, this process takes over the windows again and paints gray in the window.

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

destroys the subwindow's instance data.

Processes that take over windows should follow guidelines discussed in *Overlapped Windows: Imaging Facilities* concerning the use of the *win_getowner* and *win_setowner* procedures. Preferably, the graphics subwindow interface described below should be used for this activity.

## 7.3. Graphics Subwindow

The graphics subwindow package is for programs that need a single window in which to draw. Using this subwindow package insulates programmers of this type of program from much of the complexity of the window system.

Users of this interface have the additional benefit of being able invoke their programs from outside the window system. Thus, you can write one program and have it run both inside and outside the window system. This situation is actually an illusion. What really happens when running outside the window system is that the window system is actually started up and that a single window is created in which the graphics subwindow package runs.

The graphics subwindow can also manage a retained window for the programmer. The programmer need not worry about the fact that he is in an overlapping window situation. A backup copy of the bits on the screen is maintained from which to service any repaint requests.

The graphics subwindow can be used in tool building like any of the other subwindow packages described in this chapter. However, the graphics subwindow also provides the ability for a program to run on top of an existing window by using the blanket window mechanism.

The data definition for the instance-specific data defined in */usr/include/suntool/gfxsw.h* is:

```
struct gfxsubwindow {
    int         gfx_windowfd;
    int         gfx_flags;
    int         gfx_reps;
    struct      pixwin *gfx_pixwin;
    struct      rect gfx_rect;
    caddr_t     gfx_takeoverdata;
};

#define GFX_RESTART     0x01
#define GFX_DAMAGED     0x02
```

*Gfx_windowfd* is the file descriptor of the window that is being accessed. *Gfx_reps* are the number of repetitions that continuously running (non-blocking) cyclic programs are to execute. *Gfx_pixwin* is the structure for accessing the screen. *Gfx_rect* is a cached copy of the window's current self relative dimensions. *Gfx_takeoverdata* is data private to the graphics subwindow package.

*Gfx_flags* contains bits that the client program interprets. The GFX_DAMAGED bit is set by the graphics subwindow package whenever a SIGWINCH has been received. In addition, the GFX_RESTART bit is set if the size of the window has changed or the window is not retained. The client program must examine these flags at the times described below.

GFX_DAMAGED means that *gfxsw_handlesigwinch* should be called. This flag should be examined and acted upon before looking at GFX_RESTART. GFX_RESTART is often interpreted by a graphics program to mean that the image should be scaled to a new window size and that the image should be redrawn. Many continuous programs, graphics demos for instance, redraw from the beginning of a cycle. Other event-driven programs, graphics editors and status windows, for example, redraw from their underlying data descriptions. The GFX_RESTART bit needs to be reset to 0 by the client program before actually doing any redrawing.

### 7.3.1. In a Tool Window

A graphics subwindow in a *tool* context is only applicable for event-driven programs that use the *tool_select* mechanism. Any subwindow in a tool must use this notification mechanism so that all the windows are able to cooperate in the same process.

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

sets up a graphics subwindow in a tool window. If *argv* is not zero, this array of character pointers is processed like a command line in a standard way to determine whether the window should be made retained "-r" and/or what value should be placed in *gfx_reps* "-n ####". If the returned value is NULL then the operation failed. It is the responsibility of the client to set up *toolsw->ts_io.tio_selected* if the client is to process input through the graphics subwindow.

It is also the responsibility of the client to replace *toolsw->ts_io.tio_handlesigwinch* with the client's own routine to notify the client when something about his window changes. The client *tio_handlesigwinch* will call *gfxsw_interpretesigwinch* described below.

```
gfxsw_getretained(gfxsw);
    struct      gfxsubwindow *gfxsw;
```

can be called to make a graphics subwindow retained if you choose not to do the standard command line parsing provided by *gfxsw_createtoolsubwindow*. It should be called immediately after the graphics subwindow is created. Destroying *gfxsw->gfx_prretained* has the effect of making the window no longer retained.

The procedure:

```
gfxsw_interpretesigwinch(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

is called from the client *tio_handlesigwinch* to give the graphics subwindow package a chance to set the bits in *gfxsw->gfx_flags*. The code in the client *tio_handlesigwinch* then checks the flags and responds appropriately, perhaps by calling the *gfxsw_handlesigwinch* procedure that handles SIGWINCH signals:

```
gfxsw_handlesigwinch(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

If the window is retained and the window has not changed size, this routine fixes up any part of the image that has been damaged. If the window is retained and the window has changed size, this routine frees the old retained pixrect and allocates one of the new size. If the window is not retained, the damaged list associated with the window is thrown away. The GFX_DAMAGED flag is reset to zero in this routine.

The procedure:

```
gfxsw_done(gfxsw)
    struct      gfxsubwindow *gfxsw;
```

destroys the subwindow's instance data.

## 7.3.2. Overlaying an Existing Window

The graphics subwindow provides the ability for a program to overlay an existing window. The empty subwindow described above is designed to be overlayed.

The following procedure creates a new instance of a graphics subwindow in something other than the *tool* context:

```
struct gfxsubwindow *gfxsw_init(windowfd, argv)
    int         windowfd;
    char        **argv;
```

*Windowfd* should be zero; the assumption is that there is some indication in the environment as to which window should be overlayed. See *we_getgfxwindow* in *Window Manipulation* for more information. *Argv* is like *argv* in *gfxsw_createtoolsubwindow*. In addition, arguments similar to the ones recognized by *win_initscreenfromargv* are parsed. Thus, the program can be directed to run on a particular screen. If the returned value is NULL then the operation failed.

When a screen is created from scratch, window system keyboard and mouse processing are not turned on. *Gfxsw_setinputmask* should be called instead of *win_setinputmask* when defining window input (see below) in order to enable window system keyboard and mouse processing. This mechanism is used to allow programs that listen to the standard input to still run when started from outside the window system.

*Gfx_takoverdata* in the returned *gfxsubwindow* data structure is not zero in this case. The structure of the data that this pointer refers to is private to the implementation of the graphics subwindow.

When a graphics subwindow has overlayed another window, various signal catching routines are set up if the corresponding signals have no currently defined handler routines.

The *gfxsw_catchsigwinch* procedure is set up as the signal catcher of SIGWINCH:

```
gfxsw_catchsigwinch()
```

It, in turn, calls *gfxsw_interpretesigwinch*.

The *gfxsw_catchsigtstp* procedure is set up as the signal catcher of SIGTSTP:

```
gfxsw_catchsigtstp()
```

The graphics subwindow is removed from the display tree. The pixwin of the graphics subwindow is reset. SIGSTOP is sent to the the graphics subwindow's own process.

The *gfxsw_catchsigcont* procedure is set up as the signal catcher of SIGCONT:

```
gfxsw_catchsigcont()
```

The graphics subwindow is inserted back into the display tree (presumably after *gfxsw_catchsigtstp* removed it).

Continuous programs that never use a select mechanism should examine *gfxsw->gfx_flags* in their main loop. Other programs that would like to use a select mechanism to wait for input/timeout should call:

```
gfxsw_select(gfxsw, selected, ibits, obits, ebits, timer)
    struct      gfxsubwindow *gfxsw;
    int         (*selected)(), ibits, obits, ebits;
    struct      timeval *timer;
```

as a substitute for the *tool_select*. *Selected* is the routine that is called when some input or timeout is noticed. Its calling sequence is exactly like *foosw_selected* described at the beginning of this chapter. The only difference in the semantics of this routine and *foosw_selected* is that the *gfxsw->gfx_flags* should be examined and acted upon in *selected*. *Selected* may be called with no input pending so that you are able to see the flags when they change.

*Ibits, obits, ebits* and *timer*, as well as *gfxsw* and *selected*, can be thought of as initializing an internal *toolio* structure, which is then fed to the *tool_select* mechanism.

A substitute for the *tool_done* is:

```
gfxsw_selectdone(gfxsw)
   struct      gfxsubwindow *gfxsw;
```

*Gfxsw_selectdone* is called from within the *selected* procedure passed to *gfxsw_select.*

Programs that are not using the mouse can call:

```
gfxsw_notusingmouse(gfx)
   struct      gfxsubwindow *gfxsw;
```

In certain cases, when the graphics subwindow is the only window on the display for instance, some efficiency measures can be taken. In particular, pixwin locking overhead can be reduced.

```
gfxsw_setinputmask(gfx, im_set, im_flush, nextwindownumber, usems, usekbd)
   struct      gfxsubwindow *gfxsw;
   int         nextwindownumber;
   struct      inputmask *im_set, *im_flush;
   int         usems, usekbd;
```

The calling sequence is essentially that of *win_setinputmask. Usems* being non-zero means that mouse input is wanted and so the mouse is turned on for the screen (if currently off). *Usekbd* being non-zero means that keyboard input is wanted and so the keyboard is turned on for the screen (if currently off). See *gfxsw_init* (above) for a rationale for using *Gfxsw_setinputmask* instead of *win_setinputmask.*

```
gfxsw_inputinterrupts(gfx, ie)
   struct      gfxsubwindow *gfxsw;
   struct      inputevent *ie;
```

This utility looks at **ie.* If **ie* is a character that (on a tty) normally does process control (interrupts the process, dumps core, stops the process, terminates the process), it does the similar action. This routine is meant to be a primitive substitute for tty process control while using the window input mechanism.

## 7.4. Message Subwindow

The message subwindow package displays simple ASCII strings.

A private data definition that contains instance-specific data defined in */usr/include/suntool/msgsw.h* is:

```
struct msgsubwindow {
   int         msg_windowfd;
   char        *msg_string;
   struct      pixfont *msg_font;
   struct      rect msg_rectcache;
   struct      pixwin *msg_pixwin;
};
```

*Msg_windowfd* is the file descriptor of the window that is the message subwindow. *Msg_string* is the string being displayed using *msg_font.* Only printable characters and blanks are properly dealt with, *not* carriage returns, line feeds or tabs. The implementation uses *msg_rectcache* to help determine if the size of the subwindow has changed. *Msg_pixwin* is the structure that accesses the screen.

```
struct toolsw *msgsw_createtoolsubwindow(tool, name, width, height, string, font)
    struct      tool *tool;
    char        *name;
    short       width, height;
    char        *string;
    struct      pixfont *font;
```

is the call that sets up a message subwindow in a tool window. *String* is the string being displayed using *font*. If the returned value is NULL then the operation failed. Since *msgsw_createtoolsubwindow* takes care of the set-up of the message subwindow, the reader may not be interested in the remainder of this section, except for *msgsw_setstring*.

The following struct creates a new instance of a message subwindow:

```
struct messagesubwindow *msgsw_init(windowfd, string, font)
    int         windowfd;
    char        *string;
    struct      pixfont *font;
```

*Windowfd* identifies the window to be used. *String* is the string being displayed using *font*. If the returned value is NULL then the operation failed.

```
msgsw_setstring(msgsw, string)
    struct      messagesubwindow *msgsw;
    char        *string;
```

changes the existing *msgsw->msg_string* to *string* and redisplays the window.

```
msgsw_display(msgsw)
    struct      messagesubwindow *msgsw;
```

redisplays the window.

```
msgsw_handlesigwinch(msgsw)
    struct      messagesubwindow *msgsw;
```

is called to handle SIGWINCH signals. It repairs the damage to the window if the window hasn't changed size. If the window has changed size, the string is reformatted into the new size.

```
msgsw_done(msgsw)
    struct      messagesubwindow *msgsw;
```

destroys the subwindow's instance data.

## 7.5. Option Subwindow

An option subwindow (*optionsw*) presents a mouse-and-display-oriented user interface for setting parameters and invoking commands in an application program. 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 of which corresponds to one parameter. Existing item types include labels, booleans, enumerated choices, text parameters, and command buttons. *Note:* New item types and extensions to these existing types are contemplated.

The program *optiontool* is provided as a simple example of the features discussed here. Familiarity with the behavior of the program, and with its source file */usr/suntool/src/optiontool.c*, are

helpful in reading this section. See the source code for the icontool in appendix B for a good example.

The declarations for the *optionsw* package are found in the header file */usr/include/suntool/optionsw.h*. The file */usr/include/suntool/tool_hs.h* can be included to provide the support header files for *optionsw.h*. *Optionsw.h* includes declarations of all the public procedures, as well as the following structures and their associated defined constants. The first provides a counted buffer for a text item's value to be stored into:

```
struct string_buf {
    u_int       limit;
    char        *data;
};
```

*Data* should point to an array of chars to be used as the buffer, and *limit* should be set to the size of that buffer. Use of this structure is described with *optsw_getvalue* in *Explicit Client Reading and Writing or Item Values* below.

The second is used to identify the type as well as the value of a reference:

```
struct typed_pair {
    u_int       type;
    caddr_t     value;
};

#define IM_GRAPHIC      2
#define IM_TEXT         3
#define IM_TEXTVEC      4
```

*Type* indicates what kind of object *value* points to. The current choices are indicated in the following table:

Table 7-1: Option Image Types

| Type | Value Should Be |
|---|---|
| IM_GRAPHIC | (struct pixrect*) |
| IM_TEXT | (char *) |
| IM_TEXTVEC | (char **) |

In the TEXTVEC case, *value* points to the first element of an array of string pointers; the last element of the array should be a NULL pointer. These are currently used only in enumerated items described in *Enumerated Items*.

## 7.5.1. Option Subwindow Standard Procedures

This section describes the routines needed to conform to subwindow package norms. These routines follow the general procedures provided in *Minimum Standard Subwindow Interface*.

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

creates an option subwindow within a *tool*. The handle *toolsw->ts_data* is used for the *optsw* argument in calls to other procedures of the *optionsw* package to identify the affected window and its private data. If the returned value is NULL then the operation failed. The remainder of this section is of interest only to clients outside the *tool* system.

In contexts other than a *tool*, *optsw_init* must be called explicitly. Similarly, provisions must be made for using the rest of the routines in this section.

```
caddr_t optsw_init(fd)
    int         fd;
```

*Optsw_init* takes an *fd* that identifies the window to be used for the optionsw, and returns an opaque pointer, which identifies the created optionsw in future calls to the package. If the returned value is NULL then the operation failed.

```
optsw_handlesigwinch(optsw)
    caddr_t     optsw;
```

is called to handle SIGWINCH signals. It repairs the damage to the window, and if the window has changed size, reformats the options as described below.

```
optsw_selected(optsw, ibits, obits, ebits, timer)
    caddr_t     optsw;
    int         *ibits, *obits, *ebits;
    struct      timevalue **timer;
```

is called to handle user inputs.

The cleanup routine for an *optionsw* is:

```
optsw_done(optsw)
    caddr_t     optsw;
```

It frees all storage allocated for the subwindow and its items. Of course, the client should not attempt to use any pointer associated with the optionsw or its items after a call to this routine.

## 7.5.2. Option Items

Once an *optionsw* is created, it may be populated with option items. Each item is created by a call to the create routine for the desired type; this creates the item, adds it to the items for the *optionsw*, and returns an item handle (an opaque pointer which identifies it).

In some general aspects, all items in the *optionsw* exhibit the same behavior. The left or middle mouse button indicates an item to be manipulated; the right button is left to the menu function. Pressing one of the first two buttons gets the *optionsw*'s attention, and releasing it actually completes a user-input event to which some item may respond. While the button is held down, the cursor may be slid around over the window, and each item it passes over will indicate its readiness to respond, typically by a reverse video display. Any such indication may be canceled simply by moving the cursor off the item before letting up on the button.

Each item is identified on the screen by a *label*, which may be either text or a picture provided by the client. This label is passed to the item creation routine in a *typed_pair* struct. In the

graphic case (*type* == IM_GRAPHIC), the pixrect passed pointer is used without further consideration by the optionsw implementation — the client may even change the image after the item is created. For text labels (*type* == IM_TEXT), several defaults provide a uniform style with minimal client effort. Text labels are displayed in a bold-face version of the current font. (The current font for the option subwindow starts as the window's default font, and may be reset for each item, as described under *optsw_setfont* in *Miscellany* below.) The text of the label is modified to indicate the type of the item visually:

Boolean items are surrounded by square brackets: "**[text]**"

Commands are surrounded by parentheses: "**(text)**"

Enumerated items have a colon appended to their label, and braces surrounding the set of their values: "**text:** { choice1 choice2 choice3 }"

Text items have a colon appended to their label: "**text:** <value>"

Label items have their exact text presented in the bold face: "**text**".

The text of the label is copied by the optionsw implementation; it may not be modified by the client after the item is created.

Clients which find these defaults too restrictive are free to generate their own labels (by using *pf_text* into a memory pixrect, for example) and pass them in as type IM_GRAPHIC.

### 7.5.2.1. Boolean Items

The following procedure creates an item which maintains a boolean (TRUE or FALSE) value:

```
caddr_t optsw_bool(optsw, label, init, notify)
    caddr_t     optsw;
    struct      typed_pair *label;
    int         init;
    int         (*notify)();
```

Its *label* contains a pointer to a *typed_pair* as described above. The label is displayed in reverse video whenever the item is TRUE. The value of the item is initially set to *init*, and is toggled whenever the user selects the item. (It may also be set by a call to *optsw_setvalue*, as described below.) Whenever user action changes the value of the item, the procedure *notify* is called with the new value, as described in *Client Notification Procedures*. This argument may be NULL to indicate that no notification is desired.

### 7.5.2.2. Command Items

The following procedure creates an item that invokes the client procedure *notify* when selected by the user:

```
caddr_t optsw_command(optsw, label, notify)
    caddr_t     optsw;
    struct      typed_pair *label;
    int         (*notify)();
```

The created item has no value. All three arguments are the same as their couterparts in *optsw_bool*.

### 7.5.2.3. Enumerated Items

The following procedure creates an item in which exactly one of a set of choices is in effect at any time:

```
caddr_t optsw_enum(optsw, label, choices, flags, init, notify)
    caddr_t     optsw;
    struct      typed_pair *label;
    struct      typed_pair *choices;
    int         flags;
    int         init;
    int         (*notify)();
```

The value is interpreted as a 0-based index into the choices for the selection. *Optsw, label,* and *notify* are as above. *Choices* is a vector of images to be displayed for the choices; for now its type must be ITEM_VEC. This means that the data pointer for *choices* addresses an array of string pointers, one for each possible choice plus a NULL indicating the end of the array. *Init* is the initial value of the item; it should be at most the size of the *choices* array minus 2 (to avoid the null pointer which terminates the array). *Flags* will eventually indicate layout options, but for now should be 0.

### 7.5.2.4. Label Items

The following procedure creates an item which does nothing but paint itself. This item type may be used to include labeling information in the option subwindow.

```
caddr_t optsw_label(optsw, label)
    caddr_t     optsw;
    struct      typed_pair *label;
```

*Optsw* and *label* are as above.

### 7.5.2.5. Text Items

The following procedures create an item which holds a text value:

```
caddr_t optsw_text(optsw, label, default_value, flags, notify)
    caddr_t     optsw;
    struct      typed_pair *label;
    char        *default_value;
    int         flags;
    int         (*notify)();

    #define OPT_TEXTMASKED
```

*Optsw, label,* and *notify* are as above. *Default_value* is the initial value of the item. *Flags* specify attributes of the created item; currently, only the *masked* attribute is supported. If OPT_TEXTMASKED in flags is set, each character of the text item will be displayed as an asterisk. This feature is useful for text parameters which should not be displayed, such as passwords. The true value of the item is returned by *optsw_getvalue* described below. *Notify* is like the procedures of the other item-creation routines. It is called whenever the value of the text item is changed, except by a call to *optsw_setvalue.* Its arguments are handles for the optionsw and the

item. *Optsw_getvalue* should be used to actually retrieve the new value. This parameter to *optsw_text* may be NULL to indicate 'no notification.'

There may be multiple text items in an option subwindow. At any time, one of them "has the caret." Any keystrokes directed to the option subwindow will be directed to this item. The item that has the caret is indicated by a box around its label. Initially, this is the first text item created in the option subwindow. The user may set the caret in another item by clicking either the left or middle mouse button while the cursor is pointing at the new item's label.

The caret may also be determined and reset programmatically by calls to the following procedures:

```
caddr_t optsw_getcaret(optsw)
    caddr_t     optsw;
```

returns an item handle for the item that currently has the caret.

```
caddr_t optsw_setcaret(optsw, ip)
    caddr_t     optsw;
    caddr_t     ip;
```

sets the caret on the item indicated by *ip*, and returns *ip* if successful. Otherwise, it returns NULL. *Ip* should be a handle on a text item.

Only displayable characters will be accepted in the item (ASCII codes 040-0176 inclusive). The user's erase (character delete) and kill (line delete) characters are available for editing existing text. The first will delete the last character of the text; the latter will delete the whole string. Other characters will be discarded.

Text items will expand to fit the remainder of their option subwindow's width. This may be more polymorphism than clients desire. See the discussion under *Item Layout and Relocation* below.

*Note:* This release of text items includes the following restrictions:

- Values of text parameters are restricted to a single line of text, less than 1000 characters long. Characters which extend beyond the item's right edge will not be displayed, although they are entered and edited the same as visible characters.
- Text items may be edited only at their ends. The available operations are: add a character to the end, delete a character from the end, and delete the whole value.

While significant extension to the functionality of text items is planned, the actual interface (the external procedure definitions and data structures) are designed to accommodate those extensions without change.

## 7.5.3. Item Layout and Relocation — SIGWINCH Handling

As each item is created, its width and height are determined and stored in the item's private data. No left and top positions are assigned at this time. Later, whenever a signal is received which indicates that the size of the subwindow has changed (in particular, when the tool is first displayed, and the size grows from 0 to the initial window), a layout procedure determines positions for all the items in the window.

The default layout procedure starts in the upper-left corner of the subwindow and places items in successive positions to the right, and then in successive rows down the window. Item positions are not normally fixed; items may be repositioned if the window is later laid out again with a different size.

If an item is encountered with either of its top or left edges fixed, that specification is accepted without further consideration — it is possible to lay one item down on top of a previously positioned item, or to position it out of sight to the right or below the subwindow boundary.

Positioning of subsequent items after an item with a fixed position may be affected in three ways:

1. The top of the row in which the item appears may move down, but not up, for the rest of the items in the row.
2. Subsequent items in the same row will not be positioned to the left of the item's right edge.
3. Items in subsequent rows will not be positioned above the bottom of the fixed item.

If an item is encountered which does not have fixed width (currently, only a text item), an attempt will be made to expand the item to fill the remaining width in the option subwindow. This is done through a rather simple-minded negotiation between the general layout procedure and the flexible item. If both the position and width of the item are flexible, the result of this negotiation may not be very satisfactory to observers. In most cases, the position, the width, or both should be fixed.

At any time between an item's creation and its destruction, the client may inquire or modify its current size and position. This is done via the following two procedures:

```
optsw_getplace(optsw, ip, place)
  caddr_t       optsw;
  caddr_t       ip;
  struct        item_place *place;

optsw_setplace(optsw, ip, place, reformat)
  caddr_t       optsw;
  caddr_t       ip;
  struct        item_place *place;
  int           reformat;
```

*Optsw* is the handle returned by *optsw_init*. *Ip* is the pointer to an *opt_item* struct returned by the item's create routine. *Place* is a pointer to a *struct item_place* described below.

The *optsw_setplace* arguments are parallel to those of *optsw_getplace*. *Place* is a pointer to a struct *item_place*, which contains a rect and four boolean flags indicating that a value is to be fixed for that item. The *reformat* argument indicates that the window is to be laid out and displayed anew, taking the changed item into account. This should generally be done any time after the window has been opened, since the item is already displayed, but it may be postponed if a series of adjustments are to be made; in that case, it is appropriate to reformat only after the last item's place is set.

The following struct is also described in *optionsw.h*:

```
struct item_place {
   struct       rect rect;
   struct       {
      x : 1;
      y : 1;
      w : 1;
      h : 1;
   } fixed;
};
```

*Rect* indicates the current size and position of the item, and the four bit-fields *fixed.x*, *fixed.y*, *fixed.w*, and *fixed.h* are TRUE if the corresponding dimension may not be adjusted by the layout procedure.

For convenience in laying out string items, two functions convert character columns and lines to the appropriate pixel coordinates:

```
int optsw_coltox(optsw, col)
    caddr_t     optsw;
    int         col;

int optsw_linetoy(optsw, line)
    caddr_t     optsw;
    int         line;
```

The dimensions used in calculating these coordinates are the width of the character 'a' in the optionsw's default font and the nominal height of that font, that is, the distance between baselines of successive unleaded lines of text. Both columns and rows start at 0.

## 7.5.4. Client Notification Procedures

Most item types provide a mechanism for notifying clients that the value of an item has been changed by the user. The same general mechanism is used to specify the procedure to be invoked in response to selection of a command button.

In each case, a pointer to a procedure is passed to the item-creation routine and stored with the item. This procedure pointer may be zero, in which case there is no client notification. When appropriate, this *notification* procedure is invoked by *optionsw* code with arguments to identify the affected subwindow and item, and the new value assigned to the item. The general form for these procedures is:

```
notify(optsw, item, value)
    caddr_t     optsw;
    caddr_t     item;
    int         value;
{ ... processing to respond to item's new value.}
```

Procedures to be invoked in response to a command button-push have the same form, except there is no *value* parameter. Notification of changes to text items also omit the *value* parameter.

Note that the notification procedure is ***provided by the client*** and ***invoked by the optionsw package.***

## 7.5.5. Explicit Client Reading and Writing of Item Values

Clients may read the current value of an item by calling the procedure:

```
int optsw_getvalue(ip, dest)
    caddr_t     ip;
    caddr_t     dest;
```

*Ip* is the item handle which identifies the item whose value is sought; *dest* is the address of the destination in which the value is to be stored. For items with a numeric value, *dest* should actually be a pointer to an *int*; the value will be stored in the indicated *int*, and returned as the

value of the function. Items which have no value (commands, labels) store and return –1.

For text items, *dest* should be a pointer to a struct *string_buf*, whose *limit* is the length of the associated *data* array. *Optsw_getvalue* will store characters from the value of the indicated item into *(*dest->data)*, and return the number of characters stored. If there is room, a terminating NULL character will be written, and a later call to *optsw_getvalue* will store characters starting at the beginning of the item's value. Otherwise, the data buffer will be filled and the returned count will be equal to *dest->limit*; the next call to *optsw_getvalue* for this item will resume storing characters with the first character not reported in the previous call. Multiple calls to *optsw_getvalue* may thus be used to retrieve a long value through a short buffer. Eventually, there will be room to store a null character, and the whole value will have been reported; the next call to *optsw_getvalue* for this item will restart at the beginning of the value.

Clients may set the value of an item by calling:

```
optsw_setvalue(optsw, ip, value)
    caddr_t     optsw;
    caddr_t     ip;
    caddr_t     value;
```

*Optsw* is the opaque handle on the option subwindow; it enables repainting of the modified item. *Ip* indicates the item to be modified, *Value* should be an appropriate value for the item, which is then cast to *caddr_t*. That is, booleans and enumerateds should provide an **int** (or **unsigned**); text items should provide a (**char** *). For example, if *optsw_setvalue* is being used to change a boolean item, *value* could be:

```
(caddr_t) FALSE
```

## 7.5.6. Miscellany

Clients may inquire and set the font that is being used for displaying item labels and values. Fonts for these objects are determined at the time the object is created; different items may use different fonts. Thus, the client may create an object, change the font, create more objects which will use the new font, and then change the font back (or to a third value) for succeeding items.

```
struct pixfont *optsw_getfont(optsw)
    caddr_t     optsw;
```

returns the current font for the indicated *optsw*.

```
optsw_setfont(optsw, font)
    caddr_t     optsw;
    struct      pixfont *font;
```

sets the *optsw*'s font to be *font*.

Given an item in an optionsw, the routine:

```
optsw_nextitem(optsw, ip)
    caddr_t     optsw;
    caddr_t     ip;
```

returns a handle for the next item in sequence. If *ip* is NULL, the first item in the window will be returned; if *ip* refers to the last item in the optionsw, NULL is returned.

The routine:

```
optsw_removeitems(optsw, ip, count, reformat)
    caddr_t     optsw;
    caddr_t     ip;
    int         count;
    int         reformat:
```

removes at most *count* items from *optsw*, making them inaccessible to the user, but not destroying them. They may be restored later by a call to *optsw_restoreitems*. The subwindow is redisplayed without them if reformat is TRUE. The number of items so removed is returned; this may be less than count if the items in the subwindow are exhausted before *count* has been removed.

Starting at the item indicated by *ip*, the routine:

```
optsw_restoreitems(optsw, ip, count, reformat)
    caddr_t     optsw;
    caddr_t     ip;
    int         count;
    int         reformat:
```

restores at most *count* items in *osw* and returns the number restored. This may be left than *count* if all extant for the optionsw are exhausted, or an item which is not currently removed is encountered, first. The subwindow is redisplayed with the restored items if reformat is TRUE.

For assistance in implementing applications which use option subwindows, two routines are provided which print a formatted display of the optionsw and/or its items, to a stream of the client's choice:

```
optsw_dumpsw(stream, optsw, verbose)
    FILE        *stream;
    caddr_t     optsw;
    bool        verbose;

optsw_dumpitem(file, ip)
    FILE        *file;
    caddr_t     *ip;
```

For each procedure, the client says where to write the dump with the *stream* argument, and identifies the object to be dumped with the *optsw* or *ip* argument. If *verbose* is true, *optsw_dumpsw* will dump all the items of the optionsw.

## 7.6. Terminal Emulator Subwindow

This is the subwindow package that provides a Sun Terminal emulator.

The private data definition that contains instance-specific data defined in */usr/include/suntool/ttysw.h* is:

```
struct ttysubwindow {
                                /* Private data*/
};
```

*Note:* Only one TTY subwindow per process.

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

is the call that sets up a terminal emulator subwindow in a tool window. *Ttysw_createtoolsubwindow* takes care of setting up the terminal emulator subwindow except for the forking of the program. If the returned value is NULL then the operation failed. Thus, clients of this routine may want to ignore the remainder of this section except for the discussion of *ttysw_fork* and perhaps *ttysw_becomeconsole.*

```
struct ttysubwindow *ttysw_init(windowfd)
    int         windowfd;
```

creates a new instance of a tty subwindow. *Windowfd* is the window that is to be used. If the returned value is NULL then the operation failed.

```
ttysw_becomeconsole(ttysw)
    struct      ttysubwindow *ttysw;
```

sets up the terminal emulator to receive any output directed to the console. This should be called after calling *ttysw_init.*

```
ttysw_saveparms(ttyfd)
    int         ttyfd;
```

should be called by the screen initialization program, e.g., *suntools*(1). This saves the characteristics of the terminal *ttyfd* in an environment variable. Terminal emulation processes forked from the screen initialization process will get their characteristics from this environment variable; terminal emulation processes started directly from shells get their characteristics from the standard error tty. *Ttysw_saveparms* is needed because a screen initialization program is often started from the console, whose characteristics can change due to console redirection.

```
ttysw_handlesigwinch(ttysw)
    struct      ttysubwindow *ttysw;
```

is called to handle SIGWINCH signals. On a size change, the terminal emulator's display space is reformatted. Also, its process group is notified via SIGWINCH that the size available to it is different. Refer to *TTY-Based Programs in TTY Subwindows.* If there is display damage to be fixed up, the terminal emulator redisplays the image by using character information from its screen description.

```
ttysw_selected(ttysw, ibits, obits, ebits, timer)
    struct      ttysubwindow *ttysw;
    int         *ibits, *obits, *ebits;
    struct      timeval **timer;
```

reads input and writes output for the terminal emulator. **Ibits, *obits* and **timer* are modified by *ttysw_selected.* See the general discussion of *tio_selected* type procedures in *Minimum Standard Subwindow Interface.*

```
int ttysw_fork(ttysw, argv, inputmask, outputmask, exceptmask)
    struct      ttysubwindow *ttysw;
    char        **argv;
    int         *inputmask, *outputmask, *exceptmask;
```

forks the program indicated by **argv*. The identifier of the forked process is returned. If the returned value is -1 then the operation failed and the global variable *errno* contains the error code. There are the following possibilities:

- If **argv* is NULL, the user SHELL environment value is used. If this environment parameter is not available, */bin/sh* is used.
- If **argv* is "-c", this flag and *argv[1]* are passed to a shell as arguments. The shell then runs *argv[1]*. The argument list for this case becomes *shell/-c/argv[1]/0*.
- If **argv* is not NULL, the program named by *argv[0]* is run with the arguments given in the rest of *argv*. The argument list should be NULL terminated.

The arguments **inputmask, *ouputmask, *exceptmask* are dereferenced by *ttysw_fork* and set to the values that the terminal eumlator subwindow manager wants to wait on in a subsequent *select call.*

```
ttysw_done(ttysw)
    struct      ttysubwindow *ttysw;
```

destroys the subwindow's instance data.

## 7.6.1. TTY-Based Programs in TTY Subwindows

TTY-based programs, such as *csh, sh,* and *vi,* which use the *termcap* to determine the size of their screen, need not know about windows to run reasonably under the terminal emulator. The *termcap* library will return the current number of lines and columns of the terminal emulator. However, if the user changes his window's size while one of these programs is running, the terminal emulator and the program may disagree about what the terminal size is.

In the case of a size change, the terminal emulator sends a SIGWINCH signal to its process group. If a child process doesn't catch the signal, no harm is done because the default action for SIGWINCH is that the signal be ignored. A child process can catch the signal, and then requery the *termcap* library for the correct terminal size. Unfortunately, no TTY-based programs do this now.

The terminal emulator and the *termcap* library communicate size information through *ioctl* system calls on the pseudo-tty shared by both. The terminal emulator makes a TIOCSSIZE *ioctl* call to set the size of the pseudo-tty. The *termcap* library or some other TTY-based program makes a TIOCGSIZE *ioctl* call to get the size of the pseudo-tty. These constants and the data that they pass in the *ioctl* call are further defined in */usr/include/sys/ioctl.h.*

```
int we_getmywindow(windowname)
    char        *windowname;
```

can be called by programs running under a window system pseudo-tty to find out the terminal emulator's window name. This information is passed from the terminal emulator process to a child process through the environment variable WINDOW_ME, which is set to be the subwindow's device name, for example */dev/win5*. *We_getmywindow* reads WINDOW_ME's value into *windowname*. A return value of 0 indicates success. *Windowname* should point to at least WIN_NAMESIZE characters. This information could be the handle needed for a program to perform some sort of special window management function not provided by the default window manager.

# Chapter 8

# Suntool: User Interface Utilities

This chapter describes the programming interface to a variety of separate packages that implement the user interface of the *suntool* layer. Because these utilities are not tied to the notions of *tool* and *subwindow* as described in a previous chapter, they can be used as is, in another user interface system written on top of the *sunwindow* basic window system. For convenience, these utilities are associated directly with the *suntool* software layer.

## 8.1. Full Screen Access

To provide certain kinds of feedback to the user, it may be necessary to violate window boundaries. Pop-up menus, prompts and window management are examples of the kind of operations that do this. The *fullscreen* interface provides a mechanism for gaining access to the entire screen in a safe way. The package provides a convenient interface to underlying *sunwindow* primitives. The following structure is defined in */usr/include/suntool/fullscreen.h*:

```
struct fullscreen {
    int         fs_windowfd;
    struct      rect fs_screenrect;
    struct      pixwin *fs_pixwin;
    struct      cursor fs_cachedcursor;
    struct      inputmask fs_cachedim;
    int         fs_cachedinputnext;
};
```

*Fs_windowfd* is the window that created the *fullscreen* object. *Fs_screenrect* describes the entire screen's dimensions. *Fs_pixwin* is used to access the screen via the pixwin interface. The coordinate space of fullscreen access is the same as *fs_windowfd*'s. Thus, pixwin accesses are not necessarily done in the screen's coordinate space. Also, *fs_screenrect* is in the window's coordinate space. If, for example, the screen is 1024 pixels wide and 800 pixels high, *fs_windowfd* has its left edge at 300 and its top edge at 200, that is, both relative to the screen's upper left-hand corner, then *fs_screenrect* is {-300, -200, 1024, 800}.

The original cursor, *fs_cachedcursor*, input mask, *fs_cachedim*, and the window number of the input redirection window, *fs_cachedinputnext*, are cached and later restored when the fullscreen access object is destroyed.

```
struct fullscreen *fullscreen_init(windowfd)
    int         windowfd;
```

gains full screen access for *windowfd* and caches the window state that is likely to be changed during the lifetime of the fullscreen object. *Windowfd* is set to do blocking I/O. A pointer to this object is returned although a global pointer named *sunwindow* will keep multiple processes from gaining fullscreen access at the same time.

During the time that the full screen is being accessed, no other processes can access the screen, and all user input is directed to *fs->fs_windowfd*. Because of this, use fullscreen access

infrequently and for only short periods of time.

*Fullscreen_destroy* restores *fs*'s cached data:

```
fullscreen_destroy(fs)
    struct      fullscreen *fs;
```

It releases the right to access the full screen and destroys the fullscreen data object. *Fs->fs_windowfd*'s input blocking status is returned to its original state.

## 8.2. Icon Display Facility

This section describes an icon display facility. The icon structure is simply a stylized description of a useful class of images. Icons normally serve more to identify an object than display its contents. A typical use of an icon is to identify a currently unused but available tool. Another use might be a graphical depiction of an object, a document, database element, or resource for instance, that a user might want to point at with his mouse. The icon structure is declared in the file */usr/include/suntool/icon.h*:

```
struct icon {
    short       ic_width;
    short       ic_height;
    struct      pixrect *ic_background;
    struct      rect ic_gfxrect;
    struct      pixrect *ic_mpr;
    struct      rect ic_textrect;
    char        *ic_text;
    struct      pixfont *ic_font;
    int         ic_flags;
};

#define ICON_BKGRDPAT    0x02
#define ICON_BKGRDGRY    0x04
#define ICON_BKGRDCLR    0x08
#define ICON_BKGRDSET    0x10
```

*Ic_width* and *ic_height* describe the full size of the icon. *Ic_background* is an optional pattern with which to prepare the image background. *Ic_gfxrect* and *ic_textrect* describe two subareas of the icon (icon coordinate system relative), which may overlap. *Ic_mpr* addresses a memory pixrect as described in *Memory Pixrects*. *Ic_mpr* has the graphic portion of the icon, *ic_text* points to a string, and *ic_font* a font in which to display it. The bits of *ic_flags* are defined above and indicate different ways to prepare the background of the image before adding *ic_mpr* and the text:

ICON_BKGRDPAT
: use *ic_background*

ICON_BKGRDGRY
: use a standard gray pattern used by the background window (this background is the memory pixrect *tool_bkgrd* defined in */usr/include/suntool/tool.h*).

ICON_BKGRDCLR
: clear (white out) the image

ICON_BKGRDSET
set (solid black) the image.

The function:

```
icon_display(icon, pixwin, x, y)
    struct      icon *icon;
    struct      pixwin *pixwin;
    int         x, y;
```

displays *icon* offset (*x, y*) from the origin of *pixwin*. The background is prepared according to *icon->ic_flags*. The graphic portion of the icon is displayed next, followed by the text; thus, if they overlap, the text will come out on top.

There are no strict restrictions on the size of an icon. However, the facility becomes relatively pointless if the icon is too large. Non-uniform icons have esthetic and placement defects. Therefore, a set of standard dimensions should be provided for any particular class of icons. Here are the standards used by clients of tools defined in */usr/include/suntool/tool.h*:

```
#define TOOL_ICONWIDTH    64
#define TOOL_ICOHEIGHT    64
#define TOOL_ICONMARGIN   2

#define TOOL_ICONIMAGEWIDTH
#define TOOL_ICONIMAGEHEIGHT
#define TOOL_ICONIMAGELEFT
#define TOOL_ICONIMAGETOP

#define TOOL_ICONTEXTWIDTH
#define TOOL_ICONTEXTHEIGHT
#define TOOL_ICONTEXTLEFT
#define TOOL_ICONTEXTTOP
```

These constants put the icon in a 64-pixel square, including a two-pixel margin all around. The graphics and text regions are defined relative to the size of the icon and its margin; the graphics area covers the whole icon inside the margin, and the text overlies the bottom 3/4 of that region. The TOOL_ICONIMAGE* and TOOL_ICONTEXT* constants hold defaults for generating reasonable images when *ic_gfxrect* and *ic_textrect* respectively are initialized to them.

## 8.3. Pop-up Menus

A pop-up menu is a collection of items that a user can choose among by pointing the cursor at the desired item. It is quickly displayed in response to a button push, remains visible as long as the user holds the button down, and disappears as soon as the button is released.

Several menus can be presented at once. They appear to the user as a stack of images with the header of each menu visible, along with the items of the top menu in a vertical list. The user can bring other menus to the top by the same mechanism as choosing an item in the top menu.

A single menu is described by the following structure defined in */usr/include/suntool/menu.h*:

```
struct menu {
    int         m_imagetype;
    caddr_t     m_imagedata;
    int         m_itemcount;
    struct      menuitem *m_items;
    struct      menu *m_next;
    caddr_t     m_data;
};

#define MENU_IMAGESTRING 0x0
```

*M_imagetype* describes the data type of *m_imagedata*. *M_imagedata* is a pointer to the data displayed in the header of the menu. MENU_IMAGESTRING is the only currently defined image data type and is a character pointer. *M_next* addresses the next menu in a stack; it is NULL if this menu is the last or only one in the stack. *M_data* is private data utilized by the menu package while displaying menus. *M_items* is an array of *menuitems* whose length is *m_itemcount*.

```
struct      menuitem {
    int         mi_imagetype;
    caddr_t     mi_imagedata;
    caddr_t     mi_data;
};
```

A *menuitem* consists of a display token/data pair. *Mi_imagetype* describes the data type of *mi_imagedata*. *Mi_imagedata* is a pointer to the data displayed in this item. MENU_IMAGESTRING is the only defined image data type and is a character pointer. *Mi_data* is private to the creator of the item. Typically, it is an identifier that differentiates this item from others.

A client of the menu package constructs a stack of menus or several, for different situations by allocating menu structures and menuitem arrays and initializing all the fields in them. This involves hooking up all the data structures by setting the various pointers. An example of a menu set is found in *Sample Tools* in the *panetool* program. Button-down on the right mouse button is the standard invocation. Then when a user action initiates menu processing, the client calls:

```
struct menuitem *menu_display(menuptr, event, iowindowfd)
    struct      menu **menuptr;
    struct      inputevent *event;
    int         iowindowfd;
```

*Menuptr* is the address of a menu pointer that points to the first or "top" menu structure in a menu stack. If the user causes the stack order to be rearranged, this indirection allows the menu package to leave the new top of the stack in **menuptr* upon returning from *menu_display*. The menu package shuffles the stack's *m_next* values to rearrange the stack order. This enables the menu stack to be redisplayed in the order it was left in the last invocation.

*Event* is the *inputevent* which provoked the menu. The location information, *event->ie_locx*, *event->ie_locy*, in the event controls where the menus will be displayed. *Event->ie_code* is the event that is treated as the "menu button;" that is, the menu is displayed until this button goes up. The right mouse button is the usual menu button. The left mouse button is always used as the accelerator to bring rear menus forward. If it wasn't an explicit user action that provoked the call to *menu_display*, these three *event* fields must be loaded with the desired values

beforehand.

*Iowindowfd* is the file descriptor for the window that is displaying the menu. It is also the window that is read for user input. The *event* location values are relative to this window.

*Menu_display* currently uses the mechanism described in *Full Screen Access*. *Menu_display* temporarily modifies *iowindowfd*'s input mask to allow mouse motion and buttons to be placed on this window's input queue. All the menus in the stack are displayed, and there can only be one stack on the screen at a time. The font used for strings is that returned from *pw_pfsysopen*.

*Menu_display* returns the *menuitem*, which was under the cursor when the user released the mouse button, or NULL if the cursor was not over an item.

## 8.3.1. Prompt Facility

A prompt facility is sometimes used with menus to tell the user to proceed from his current state. Prompting can also be done without menus. The definitions for the prompt facility are found in */usr/include/suntool/menu.h*:

```
struct prompt {
    struct      rect prt_rect;
    struct      pixfont *prt_font;
    char        *prt_text;
};

#define PROMPT_FLEXIBLE -1
```

*Prt_rect* is the rectangle in which the text addressed by *prt_text* will be displayed using *prt_font*. Only printable characters and blanks are properly dealt with. Carriage returns, line feeds or tabs are not. If any of *prt_rect*'s fields are PROMPT_FLEXIBLE, that dimension is automatically chosen by the prompt mechanism to accommodate all the characters in *prt_text*.

```
menu_prompt(prompt, event, iowindowfd)
    struct      prompt *prompt,
    struct      inputevent *event;
    int         iowindowfd;
```

*Menu_prompt* displays the indicated prompt (*prompt->prt_rect* is *iowindowfd* relative), and then waits for any input event other than mouse motion. It then removes the prompt, and returns the event which ended the prompt's existence in *event*. *Iowindowfd* is the window from which input is taken while the prompt is up. The *fullscreen* access method is used during prompt display.

## 8.4. Selection Management

This section describes an interface to a *selection manager* that is used to coordinate access to a single data entity called the *current selection*. The current selection is globally accessible by any process, thus providing an inter-tool data exchange mechanism.

A common style of operation/operand command specification is a non-modal one in which the operand is specified first. In the window system, the operand is called the *selection* since it usually requires that the user select something with the pointing device. A selection is highlighted in some way and persists until an operation removes it programmatically or the user performs some action that causes the selection to be removed.

The header file */usr/include/suntool/selection.h* contains the definition necessary for using selections. The object that describes a selection is:

```
struct selection {
    int         sel_type,
    int         sel_items,
    int         sel_itembytes,
    int         sel_pubflags;
    caddr_t     sel_privdata;
};

#define SELTYPE_NULL 0
#define SELTYPE_CHAR 1
```

*Sel_type* indicates the type of the selection. Currently, SELTYPE_NULL (no selection) and SELTYPE_CHAR (ASCII characters) are the only selection types defined. *Sel_items* is the number of items in the selection data. *Sel_itembytes* is the number of bytes each item occupies in the selection data. *Sel_pubflags* is used to contain publicly understood flags that further describe the selection. *Sel_privdata* is used to contain 32 bits worth of privately understood data that is only understood between implementations of a particular selection type.

The selection structure is not to be confused with actual selection data itself, the characters in a SELTYPE_CHAR selection, for instance.

```
selection_set(sel, sel_write, sel_clear, windowfd)
    struct      selection *sel
    int         (*sel_write)();
    int         (*sel_clear)();
    int         windowfd;

sel_write(sel, file)
    struct      selection *sel;
    FILE        *file;

sel_clear(sel, windowfd)
    struct      selection *sel;
    int         windowfd;
```

*Selection_set* is used to change the current selection. *Sel* describes the selection. *Sel_write* is a procedure that is called to store information into the selection. Currently, only *selection_set* calls *sel_write*, but in the future *sel_write* might be called at any time. The *sel_write* procedure takes as arguments *sel*, the selection description handed to *selection_set*, and *file*, a standard I/O FILE pointer. The standard I/O library is used to write the selection data to *file*. *Windowfd* is the window that is making the selection.

*Sel_clear* is a procedure that the selection manager would call when it wanted the selection currently being set to be dehighlighted. This could happen when another selection had been made. *This clear feature is not currently implemented. When implemented this call could come at any time after returning from selection_set.*

```
selection_clear(windowfd)
    int         windowfd;
```

is called when *windowfd* wants to clear the current selection. Ideally, there is only one selection on the screen at a time so that the user doesn't become confused about which operand will be

affected by his next command.

Since the *sel_clear* feature is not currently implemented, it is the selection maker's decision as to when to dehilight his selection feedback. The only existing use of the selection mechanism waits for the user to move his cursor out of the window that made the selection before dehilighting it.

```
selection_get(sel_read, windowfd)
    int         (*sel_read)();
    int         windowfd;

sel_read(sel, file)
    struct      selection *sel;
    FILE        *file;
```

*Selection_get* is used to find out the current selection. *Sel_read* is a procedure that *selection_get* calls to enable the client to retrieve the selection. *Windowfd* is the window that wants to find out about the selection.

The *sel_read* procedure takes as arguments *sel*, the selection description of the current selection, and *file*, a standard I/O FILE pointer. The standard io library is used to read the selection data from *file*. *Sel_read* should check the type of the selection and make sure that it is a type with which it can deal.

## 8.5. Window Management

The procedures in this section implement common functions for managing windows.

### 8.5.1. Window Manipulation

These routines provide the standard window management user interface presented by tool windows:

```
wmgr_open(toolfd, rootfd)
   int          toolfd, rootfd;

wmgr_close(toolfd, rootfd)
   int          toolfd, rootfd;

wmgr_move(toolfd)
   int          toolfd;

wmgr_stretch(toolfd)
   int          toolfd;

wmgr_top(toolfd, rootfd)
   int          toolfd, rootfd;

wmgr_bottom(toolfd, rootfd)
   int          toolfd, rootfd;

wmgr_refreshwindow(windowfd)
   int          windowfd;
```

In each of the above routines, ***toolfd*** is a file descriptor for a tool window and ***rootfd*** is a file descriptor for the root window. *Wmgr_open* opens a tool window from its iconic state to normal size. If the window is already open, *wmgr_open* does nothing. *Wmgr_close* closes a tool window from its normal size to its iconic size. If the window is already closed, *wmgr_close* does nothing. *Wmgr_move* prompts the user to move the tool window or cancel the operation. If confirmed, the rest of the move interaction, including dragging the window and moving the bits on the screen, is done. *Wmgr_stretch* is like *wmgr_move*, but it stretches the window instead of moving it. *Wmgr_top* places the tool window on the top on the window stack. *Wmgr_bottom* places the tool window on the bottom on the window stack. *Wmgr_refreshwindow* causes ***windowfd*** and all its descendant windows to repaint.

The routine *wmgr_changerect*:

```
wmgr_changerect(feedbackfd, windowfd, event, move, noprompt)
   int          feedbackfd, windowfd;
   struct       inputevent *event;
   bool         move, noprompt;
```

implements *wmgr_move* and *wmgr_stretch*, including the user interaction sequence. *Windowfd* is moved (1) or stretched (0) depending on the value of *move*. To accomplish the user interaction, the input event is read from the ***feedbackfd*** window (usually the same as ***windowfd***). The prompt is turned off if ***noprompt*** is 1.

```
int wmgr_confirm(windowfd, text)
   int          windowfd;
   char         *text;
```

*Wmgr_confirm* implements a layer over the prompt package for a standard confirmation user interface. *Text* is put up in a prompt box. If the user confirms with a left mouse button press, then –1 is returned. Otherwise, 0 is returned.

*Note:* The up button event is not consumed.

The window management package provides menu handling code that ties all the routines in this subsection into the *wmgr_toolmenu*. This provides a convenient way of getting access to the same menu that is presented by a tool window. If you don't like the menu provided (you want to add/subtract/change menu items), define and use a new one. The routines in this section should be all you need to put together a functionally similar window manipulation interface.

```
struct menu *wmgr_toolmenu;

wmgr_setupmenu(toolfd)
    int         toolfd;

wmgr_handletoolmenuitem(menu, mi, toolfd, rootfd)
    struct      menu *menu;
    struct      menuitem *mi;
    int         toolfd, rootfd;
```

To use the default tool menu, call *wmgr_setupmenu* just before you put up *wmgr_toolmenu*. *Wmgr_setupmenu* arranges the menu items depending on the tool state (iconic vs. normal). Passing the menu item returned from *menu_display* to *wmgr_handletoolmenuitem* causes the appropriate menu action to be done. As an example, refer to the Pane Tool code provided in *panetool.c* in appendix B.

## 8.5.2. Tool Invocation

The routines in this section provide tool invocation and default position control.

```
#define WMGR_SETPOS -1

wmgr_figuretoolrect(rootfd, rect)
    int         rootfd;
    struct      rect *rect;

wmgr_figureiconrect(rootfd, rect)
    int         rootfd;
    struct      rect *rect;
```

These routines allow windows to be assigned initial positions that don't pile up on top of one another. The *rootfd* window maintains a "next slot" position for both normal tool windows and icon windows (see *wmgr_setrectalloc* below). These procedures assign the next slot to the rect if *rect->r_left* or *rect->r_top* is equal to WMGR_SETPOS. A new slot is chosen and is then available for the next window with an undefined position.

These procedures also assign a default width and height if WMGR_SETPOS is given, again for both normal (tool) and iconic rects. *Wmgr_figuretoolrect* currently assigns tool window slots that march from near the top middle of the screen towards the bottom left of the screen. It assigns a window size correct for an 80-column by 34-row terminal emulator window. *Wmgr_figureiconrect* currently assigns icon slots that march from the left bottom towards the right of the screen. It assigns icon sizes that are 64 by 64 pixels.

```
wmgr_forktool(programname, otherargs, rectnormal, recticon, iconic)
    char        *programname, *otherargs;
    struct      rect *rectnormal, *recticon;
    int         iconic;
```

is used to fork a new tool that has its normal rectangle set to *rectnormal* and its icon rectangle set to *recticon*. If *iconic* is not zero, the tool is created iconic. *Programname* is the name of the file that is to be run and *otherargs* is the command line that you want to pass to the tool. A path search is done to locate the file. Arguments that have embedded white space should be enclosed by double quotes.

## 8.5.3. Utilities

The utilities described here are some of the low level routines that are used to implement the higher level routines. They may be used to put together a window management user interface different from that provided by tools. If a series of calls is to be made to procedures that manipulate the window tree, the whole sequence should be bracketed by *win_lockdata* and *win_unlockdata*, as described in *The Window Hierarchy*.

```
wmgr_completechangerect(windowfd, rectnew, rectoriginal, parentprleft, parentprtop)
    int         windowfd;
    struct      rect *rectnew, *rectoriginal;
    int         parentprleft, parentprtop;
```

does the work involved with changing the position or size of a window's rect. This involves saving as many bits as possible by copying them on the screen so they don't have to be recomputed. *Wmgr_completechangerect* would be called after some programmatic or user action determined the new window position and size in pixels. *Windowfd* is the window being changed. *Rectnew* is the window's new rectangle. *Rectoriginal* is the window's original rectangle. *Parentprleft* and *parentprtop* are the upper-left screen coordinates of the parent of *windowfd*.

```
wmgr_winandchildrenexposed(pixwin, rl)
    struct      pixwin *pixwin;
    struct      rectlist *rl;
```

computes the visible portion of *pixwin->pw_clipdata.pwcd_windowfd* and its descendants and stores it in *rl*. This is done by any window management routine that is going to try to preserve bits across window changes. For example, *wmgr_completechangerect* calls *wmgr_winandchildrenexposed* before and after changing the window size/position. The intersection of the two rectlists from the two calls are those bits that could possibly be saved.

```
wmgr_changelevel(windowfd, parentfd, top)
    int         windowfd, parentfd;
    bool        top;
```

moves a window to the top or bottom of the heap of windows that are descendants of its parent. *Windowfd* identifies the window to be moved; *parentfd* is the file descriptor of that window's parent, and *top* controls whether the window goes to the top (TRUE) or bottom (FALSE). Unlike *wmgr_top* and *wmgr_bottom*, no optimization is performed to reduce the amount of repainting. *Wmgr_changelevel* is used in conjunction with other window rearrangements, which make repainting unlikely. For example, *wmgr_close* puts the window at the bottom of the window stack after changing its state.

```
#define WMGR_ICONIC WUF_WMGR1

wmgr_iswindowopen(windowfd)
    int         windowfd;
```

The user data of *windowfd* reflects the state of the window via the WMGR_ICONIC flag. WUF_WMGR1 is defined in */usr/include/sunwindow/win_ioctl.h* and WMGR_ICONIC is defined in */usr/include/suntool/wmgr.h*. *Wmgr_iswindowopen* tests the WMGR_ICONIC flag (see above) and returns TRUE or FALSE as the window is open or closed.

The *rootfd* window maintains a "next slot" position for both normal tool windows and icon windows in its unused iconic rect data. *Wmgr_setrectalloc* stores the next slot data and *wmgr_getrectalloc* retrieves it:

```
wmgr_setrectalloc(rootfd, tool_left, tool_top, icon_left, icon_top)
    int         rootfd;
    short       tool_left, tool_top, icon_left, icon_top;

wmgr_getrectalloc(rootfd, tool_left, tool_top, icon_left, icon_top)
    int         rootfd;
    short       *tool_left, *tool_top, *icon_left, *icon_top;
```

If you do a *wmgr_setrectalloc*, make sure that all the values you are not changing were retrieved with *wmgr_getrectalloc*. In other words, both procedures affect all the values.

# Appendix A

# Rects and Rectlists

This appendix describes the geometric structures used with the *sunwindow* layer and a full description of the operations on these structures. Throughout *sunwindow*, images are dealt with in rectangular chunks. Where complex shapes are required, they are built up out of groups of rectangles. A *rect* is a structure that defines a rectangle. A *rectlist* is a structure that defines a list of rects.

The header files *rect.h* and *rectlist.h* are found in */usr/include/sunwindow/*. The library that provides the implementation of the functions of these data types are part of */usr/lib/libsunwindow.a*.

Although these structures are presented in terms of *sunwindow* usage with pixel units, they are really separate and can be thought of as a rectangle algebra package. Any application that needs such a facility should consider using rects and rectlists.

## A.1. Rects

The rect is the basic description of a rectangle, and there are macros and procedures to perform common manipulations on a rect.

```
#define coord  short

struct rect {
    coord       r_left;
    coord       r_top;
    short       r_width;
    short       r_height;
};
```

The rectangle lies in a coordinate system whose origin is in the upper left-hand corner and whose dimensions are given in pixels.

### A.1.1. Macros on Rects

The same header file defines some interesting macros on rectangles. To determine an edge not given explicitly in the rect:

```
#define rect_right(rp)
#define rect_bottom(rp)
struct rect *rp;
```

returns the coordinate of the last pixel within the rectangle on the right or bottom, respectively.
Useful predicates returning TRUE or FALSE are:

```
#define bool      unsigned
#define TRUE                    1
#define FALSE                   0

rect_isnull(r)                  r's width or height is 0
rect_includespoint(r,x,y)       (x,y) lies in r
rect_equal(r1, r2)              r1 and r2 coincide exactly
rect_includesrect(r1, r2)       every point in r2 lies in r1
rect_intersectsrect(r1, r2)     at least one point lies in both r1 and r2
   struct     rect *r, *r1, *r2;
   coord      x, y;
```

Macros which manipulate dimensions of rectangles are:

```
rect_construct(r, x, y, w, h)
   struct     rect *r;
   int        x, y, w, h;
```

This fills in *r* with the indicated origin and dimensions.

```
rect_marginadjust(r, m)
   struct     rect *r;
   int        m;
```

adds a margin of *m* pixels on each side of *r*; that is, *r* becomes 2**m* larger in each dimension.

```
rect_passtoparent(x, y, r)
rect_passtochild(x, y, r)
   coord      x, y;
   struct     rect *r;
```

sets the origin of the indicated rect to transform it to the coordinate system of a parent or child rectangle, so that its points are now located relative to the parent or child's origin. *X* and *y* are the origin of the parent or child rectangle within *its* parent; these values are added to, or respectively subtracted from, the origin of the rectangle pointed to by *r*, thus transforming the rectangle to the new coordinate system.

## A.1.2. Procedures and External Data for Rects

A null rectangle, that is one whose origin and dimensions are all 0, is defined for convenience:

```
extern struct rect rect_null;
```

The following procedures are also defined in *rect.h*:

```
struct rect rect_bounding(r1, r2)
   struct     rect *r1, *r2;
```

This returns the minimal rect that encloses the union of *r1* and *r2*. The returned value is a

struct, not a pointer.

```
rect_intersection(r1, r2, rd)
    struct      rect *r1, *r2, *rd;
```

computes the intersection of *r1* and *r2*, and stores that rect into *rd*.

```
bool rect_clipvector(r, x0, y0, x1, y1)
    struct      rect *r;
    coord       *x0, *y0, *x1, *y1;
```

modifies the vector endpoints so they lie entirely within the rect, and returns FALSE if that excludes the whole vector, otherwise it returns TRUE.

*Note:* This procedure should not be used to clip a vector to multiple abutting rectangles. It may not cross the boundaries smoothly.

```
bool rect_order(r1, r2, sortorder)
    struct      rect *r1, *r2;
    int         sortorder;
```

returns TRUE if *r1* precedes or equals *r2* in the indicated ordering:

```
#define    RECTS_TOPTOBOTTOM   0
#define    RECTS_BOTTOMTOTOP   1
#define    RECTS_LEFTTORIGHT   2
#define    RECTS_RIGHTTOLEFT   3
```

Two related defined constants are:

```
#define    RECTS_UNSORTED      4
```

indicating a "don't-care" order, and

```
#define    RECTS_SORTS         4
```

giving the number of sort orders available, for use in allocating arrays and so on.

## A.2. Rectlists

A *rectlist* is a structure that defines a list of rects. A number of rectangles may be collected into a list that defines an interesting portion of a larger rectangle. An equivalent way of looking at it is that a large rectangle may be fragmented into a number of smaller rectangles, which together comprise all the larger rectangle's interesting portions. A typical application of such a list is to define the portions of one rectangle remaining visible when it is partially obscured by others.

```
struct rectlist {
    coord       rl_x, rl_y;
    struct      rectnode *rl_head;
    struct      rectnode *rl_tail,
    struct      rect rl_bound;
};

struct rectnode {
    struct      rectnode *rn_next;
    struct      rect rn_rect;
};
```

Each node in the rectlist contains a rectangle which covers one part of the visible whole, along with a pointer to the next node. *Rl_bound* is the minimal bounding rectangle of the union of all the rectangles in the node list. All rectangles in the rectlist are described in the same coordinate system, which may be translated efficiently by modifying *rl_x* and *rl_y*.

The routines that manipulate rectlists do their own memory management on rectnodes, creating and freeing them as necessary to adjust the area described by the rectlist.

## A.2.1. Macros and Constants Defined on Rectlists

Macros to perform common coordinate transformations are provided:

```
rl_rectoffset(rl, rs, rd)
    struct      rectlist *rl;
    struct      rect *rs, *rd;
```

copies *rs* into *rd*, and then adjusts *rd*'s origin by adding the offsets from *rl*.

```
rl_coordoffset(rl, x, y)
    struct      rectlist *rl;
    coord       x, y;
```

offsets *x* and *y* by the offsets in *rl*. For instance, it converts a point in one of the rects in the rectnode list of a rectlist to the coordinate system of the rectlist's parent.

Parallel to the macros on rect's, we have:

```
rl_passtoparent(x, y, rl)
rl_passtochild(x, y, rl)
    coord       x, y;
    struct      rectlist *rl;
```

which add or subtract the given coordinates from the rectlist's *rl_x* and *rl_y* to convert the *rl* into its parent's or child's coordinate system.

## A.2.2. Procedures and External Data for Rectlists

An empty rectlist is defined, which should be used to initialize any rectlist before it is operated on:

```
extern struct rectlist rl_null;
```

Procedures are provided for useful predicates and manipulations. The following declarations apply uniformly in the descriptions below:

```
struct    rectlist *rl, *rl1, *rl2, *rld;
struct    rect *r;
coord     x, y;
```

Predicates return TRUE or FALSE. Refer to the following table for specifics.

Table A-1: Rectlist Predicates

| Macro | Returns TRUE if |
|---|---|
| rl_empty(rl) | Contains only null rects |
| rl_equal(rl1, rl2) | The two rectlists describe the same space identically — same fragments in the same order |
| rl_includespoint(rl,x,y) | (*x*, *y*) lies within some rect of *rl* |
| rl_equalrect(r, rl) | *rl* has exactly one rect, which is the same as *r* |
| rl_boundintersectsrect(r, rl) | Some point lies both in *r* and in *rl*'s bounding rect |

Manipulation procedures operate through side-effects, rather than returning a value. Note that it is legitimate to use a rectlist as both a source and destination in one of these procedures. The source node list will be freed and reallocated appropriately for the result.

Refer to the following table for specifics.

| Procedure | Effect |
|---|---|
| rl_intersection(rl1, rl2, rld) | Stores into *rld* a rectlist which covers the intersection of *rl1* and *rl2*. |
| rl_union(rl1, rl2, rld) | Stores into *rld* a rectlist which covers the union of *rl1* and *rl2*. |
| rl_difference(rl1, rl2, rld) | Stores into *rld* a rectlist which covers the area of *rl1* not covered by *rl2* |
| rl_coalesce(rl) | An attempt is made to shorten *rl* by coalescing some of its fragments. An *rl* whose bounding rect is completely covered by the union of its node rects will be collapsed to a single node; other simple reductions will be found; but the general solution to the problem is not attempted. |
| rl_sort(rl, rld, sort)<br>int sort; | *rl* is copied into *rld*, with the node rects arranged in *sort* order. |
| rl_rectintersection(r, rl, rld) | *rld* is filled with a rectlist that covers the intersection of *r* and *rl*. |
| rl_rectunion(r, rl, rld) | *rld* is filled with a rectlist that covers the union of *r* and *rl*. |
| rl_rectdifference(r, rl, rld) | *rld* is filled with a rectlist that covers the portion of *rl* which is not in *r*. |
| rl_initwithrect(r, rl) | Fills in *rl* so that it covers the rect *r* |
| rl_copy(rl, rld) | Fills in *rld* with a copy of *rl*. |
| rl_free(rl) | Frees the storage allocated to *rl*. |
| rl_normalize(rl) | Resets *rl*'s offsets (*rl_x*, *rl_y*) to be 0 after adjusting the origins of all rects in *rl* accordingly. |

# Appendix B

# Sample Tools

This appendix contains sample tool code for writing your own tools. Code is provided for the graphics window (*gfxtool.c*), which produces a shell subwindow and an empty subwindow in which graphics programs can run, the pane tool (*panetool.c*), which produces multiple subwindows, the option tool (*optiontool.c*), which tests the option subwindow library, and the icon tool (*icontool.c*), which is a bitmap editor for painting icons and cursors. The source files for these and other tools are found in */usr/suntool/src/*tool.c*.

## B.1. gfxtool.c Code

Code for *gfxtool.c* follows.

```
#ifndef lint
static  char sccsid[] = "@(#)gfxtool.c 1.6 83/10/18 Sun Micro";
#endif
/*
 * Sun Microsystems, Inc.
 */

/*
 *      Overview:       Graphics Window: A shell subwindow and an empty
 *                      subwindow in which graphics programs can run.
 */
#include <sys/types.h>
#include <signal.h>
#include "pixrect/pixrect.h"
#include "pixrect/pixfont.h"
#include "pixrect/pr_util.h"
#include "pixrect/memvar.h"
#include "sunwindow/rect.h"
#include "sunwindow/rectlist.h"
#include "sunwindow/pixwin.h"
#include "sunwindow/win_struct.h"
#include "sunwindow/win_environ.h"
#include "suntool/icon.h"
#include "suntool/tool.h"
#include "suntool/emptysw.h"
#include "suntool/ttysw.h"

static short ic_image[256]={
#include "gfxtool.icon"
};
mpr_static(gfxic_mpr, 64, 64, 1, ic_image);
```

```
static  struct icon icon = {64, 64, (struct pixrect *)0, 0, 0, 64, 64,
            &gfxic_mpr, 0, 0, 0, 0, (char *)0, (struct pixfont *)0,
            ICON_BKGRDGRY};

static  int sigwinchcatcher(), sigchldcatcher();

static  struct tool *tool;

gfxtool_main(argc, argv)
        int argc;
        char **argv;
{
        char    *toolname = "Graphics Tool 1.0";
        struct toolsw *ttysw, *emptysw;
        char    name[WIN_NAMESIZE];

        /*
         * Create tool window
         */
        tool = tool_create(toolname, TOOL_NAMESTRIPE|TOOL_BOUNDARYMGR,
            (struct rect *)0, &icon);
        /*
         * Create subwindows
         */
        ttysw = ttysw_createtoolsubwindow(tool, "ttysw",
            TOOL_SWEXTENDTOEDGE, 200);
        emptysw = esw_createtoolsubwindow(tool, "emptysw",
            TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE);
        /*
         * Setup gfx window environment value.
         */
        win_fdtoname(emptysw->ts_windowfd, name);
        we_setgfxwindow(name);
        /*
         * Install tool in tree of windows
         */
        signal(SIGWINCH, sigwinchcatcher);
        signal(SIGCHLD, sigchldcatcher);
        tool_install(tool);
        /*
         * Start tty process
         */
        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);
        }
        /*
         * Handle input
         */
        tool_select(tool, 1 /* means wait for child process to die*/);
        /*
         * Cleanup
         */
        tool_destroy(tool);
        exit(0);
}

static
```

```
sigchldcatcher()
{
        tool_sigchld(tool);
}

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

## B.2. panetool.c Code

Code for the *panetool.c* follows.

```
#ifndef lint
static  char sccsid[] = "@(#)panetool.c 1.8 83/10/18 Sun Micro";
#endif

/*
 * Sun Microsystems, Inc.
 */

/*
 *      Overview:       Pane Tool: Sample program to illustrate multiple
 *                      subwindows.
 */

#include <sys/types.h>
#include <sys/time.h>
#include <signal.h>
#include "pixrect/pixrect.h"
#include "pixrect/pixfont.h"
#include "sunwindow/rect.h"
#include "sunwindow/rectlist.h"
#include "sunwindow/pixwin.h"
#include "sunwindow/win_input.h"
#include "sunwindow/win_struct.h"
#include "suntool/icon.h"
#include "suntool/tool.h"
#include "suntool/msgsw.h"
#include "suntool/menu.h"

static  int sigwinchcatcher();

static  struct tool *tool;

static  char charbuf[4];

struct  menuitem m3_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct  menu m3_menubody = {
          MENU_IMAGESTRING, "M3", sizeof(m3_items) / sizeof(struct menuitem), m3_items, 0, 0 };
struct  menuitem m2_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct  menu m2_menubody = {
          MENU_IMAGESTRING, "M2", sizeof(m2_items) / sizeof(struct menuitem),
          m2_items, &m3_menubody, 0};
struct  menuitem m1_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct  menu m1_menubody = {
```

```
        MENU_IMAGESTRING, "M1", sizeof(m1_items) / sizeof(struct menuitem),
        m1_items, &m2_menubody, 0};
struct menu *stack1menutop = &m1_menubody;

struct menuitem m4_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct menu m4_menubody = {
        MENU_IMAGESTRING, "M4", sizeof(m4_items) / sizeof(struct menuitem),
        m4_items, 0, 0 };
struct menuitem m5_items[] = { MENU_IMAGESTRING, "Menu Item", 0};
struct menu m5_menubody = {
        MENU_IMAGESTRING, "M5", sizeof(m5_items) / sizeof(struct menuitem),
        m5_items, &m4_menubody, 0};
struct menuitem m6_items[] = { MENU_IMAGESTRING, "Menu Item",   0};
struct menu m6_menubody = {
        MENU_IMAGESTRING, "M6", sizeof(m6_items) / sizeof(struct menuitem),
        m6_items, &m5_menubody, 0};
struct menu *stack2menutop = &m6_menubody;
int     menutoggle;

main(argc, argv)
        int argc;
        char **argv;
{
        char   *toolname = "Pane Tool 1.0 (A sample tool)";
        struct toolsw *paneNW, *paneNE, *paneSW, *paneSE;
        extern struct pixfont *pf_sys;

        /*
         * Create tool window
         */
        tool = tool_create(toolname, TOOL_NAMESTRIPE|TOOL_BOUNDARYMGR,
           (struct rect *) 0, (struct icon *) 0);
        /*
         * Create msg subwindows
         */
        paneNW = msgsw_createtoolsubwindow(tool, "paneNW",
           100, 100, "Raw keyboard input", pf_sys);
        paneNE = msgsw_createtoolsubwindow(tool, "paneNE",
           TOOL_SWEXTENDTOEDGE, 100,
           "Key input here redirected to NW subwindow", pf_sys);
        paneSW = msgsw_createtoolsubwindow(tool, "paneSW",
           100, TOOL_SWEXTENDTOEDGE, "Display alternating menu stacks",pf_sys);
        paneSE = msgsw_createtoolsubwindow(tool, "paneSE",
           TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE,
           "Try moving subwindow boundaries", pf_sys);
        /*
         * Raw input and flushing
         */
        {
        struct inputmask im;
        int     paneNW_selected();

        input_imnull(&im);
        im.im_flags |= IM_UNENCODED;
        win_setinputmask(paneNW->ts_windowfd, &im,  &im, WIN_NULLLINK);
        paneNW->ts_io.tio_selected = paneNW_selected;
        }
        /*
         * Input redirection
```

```
        */
        {
        struct inputmask im;

        win_getinputmask(paneNE->ts_windowfd, &im, 0);
        win_setinputmask(paneNE->ts_windowfd, &im,  (struct inputmask *) 0,
            win_fdtonumber(paneNW->ts_windowfd));
        }
        /*
         * Multi menu stacks
         */
        {
        struct inputmask im;
        int     paneSW_selected();

        input_imnull(&im);
        win_setinputcodebit(&im, MENU_BUT);
        win_setinputmask(paneSW->ts_windowfd, &im,  &im, WIN_NULLLINK);
        paneSW->ts_io.tio_selected = paneSW_selected;
        }
        /*
         * Install tool in tree of windows
         */
        signal(SIGWINCH, sigwinchcatcher);
        tool_install(tool);
        /*
         * Handle input
         */
        tool_select(tool, 0);
        /*
         * Cleanup
         */
        tool_destroy(tool);
        exit(0);
}

paneNW_selected(msgsw, ibits, obits, ebits, timer)
        struct msgsubwindow *msgsw;
        int     *ibits, *obits, *ebits;
        struct timeval **timer;
{
        struct inputevent event;
        int     error;

        error = input_readevent(msgsw->msg_windowfd, &event);
        if (error < 0) {
                perror("panetool");
                return;
        }
        charbuf[0] = 'c';
        charbuf[1] = ':';
        charbuf[2] = (char) event.ie_code&0X7f;
        charbuf[3] = ' ';
        msgsw_setstring(msgsw, charbuf);
        *ibits = *obits + *ebits + 0;
}

paneSW_selected(msgsw, ibits, obits, ebits, timer)
        struct msgsubwindow *msgsw;
```

```
        int     *ibits, *obits, *ebits;
        struct timeval **timer;
{
        struct inputevent event;
        int     error;
        extern struct menuitem *menu_display();

        error = input_readevent(msgsw->msg_windowfd, &event);
        if (error < 0) {
                perror("panetool");
                return;
        }
        (void) menu_display((menutoggle)? &stack1menutop: &stack2menutop,
           &event, msgsw->msg_windowfd);
        menutoggle = !menutoggle;
        *ibits = *obits + *ebits + 0;
}

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

## B.3. optiontool.c Code

Code for the *optiontool.c* follows.

```
#ifndef lint
static char  sccsid[] = "@(#)optiontool.c 1.10 84/01/17  Sun Micro";
#endif

/*
 *   Sun Microsystems Inc.
 */

/*
 *      optiontool:    test optionsw library
 */

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

static struct tool      *tool;
static char             *name = "Option Tool 1.1";
static struct toolsw    *tsw;
static caddr_t          osw;
static struct  pixwin opt_pixwin;

static caddr_t          items [32];
static struct pixfont *font;
static struct rect      r;


static unsigned         dump_glyph[16] = {
                        0x00002000, 0x00007000, 0x0000D800, 0x00018C00,
                        0x00030600, 0x00060C1F, 0x300C1819, 0x0C183019,
```

```
                0x0330701F, 0x00E0C87F, 0x0031847F, 0x029B0261,
                0x07EF6FED, 0x1FE49012, 0x7FF89012, 0xFFF8600C
        };
static mpr_static(dump_pr, 32, 16, 1, dump_glyph);

static struct typed_pair title = { IM_TEXT, "Option Subwindow Demo" };
static struct typed_pair confirm_label = { IM_TEXT, "Quittable" };
static struct typed_pair quit_label = { IM_TEXT, "Quit" };
static struct typed_pair verbose_label = { IM_TEXT, "Verbose" };
static struct typed_pair dump_label = { IM_GRAPHIC, (caddr_t)&dump_pr};
static struct typed_pair x_label = { IM_TEXT, "Flag X" };
static struct typed_pair y_label = { IM_TEXT, "Flag Y" };
static struct typed_pair z_label = { IM_TEXT, "Flag Z" };
static struct typed_pair t1_label = { IM_TEXT, "Type here" };
static struct typed_pair t2_label = { IM_TEXT, "Secret" };
static struct typed_pair t3_label = { IM_TEXT, "line 1"};
static struct typed_pair t4_label = { IM_TEXT, "line 2"};
static struct typed_pair txtcmd_label = { IM_TEXT, "Report Text" };

static struct typed_pair enum_label = { IM_TEXT, "Choose" };
static char  *choice_values[] = { "Zero", "One", "Many", 0 };
static struct typed_pair enum_choices = { IM_TEXTVEC, (caddr_t)choice_values };

static int      n,
                confirmed = FALSE,
                remove_which,
                chooser(),
                confirmer(),
                dumper(),
                quitter(),
                reporter(),
                sigwinched(),
                verbose,
                verboser(),
                texter();

FILE  *sysout = stderr;

int     removed_items;

main()
{
        struct item_place  p;
/*
 * Create tool window
 */
        tool = tool_create(name, TOOL_NAMESTRIPE, NULL, NULL);

/*
 * Create subwindow and fill it out
 */
        tsw = optsw_createtoolsubwindow(tool, "optsw",
                        TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE);
        osw = tsw->ts_data;
        n = 0;
        items[n++] = optsw_label(osw, &title);
        items[n++] = optsw_bool(osw, &confirm_label, FALSE, confirmer);
        items[n++] = optsw_command(osw, &quit_label, quitter);
        items[n++] = optsw_bool(osw, &verbose_label, FALSE, verboser);
```

```
        items[n++] = optsw_command(osw, &dump_label, dumper);
        items[n++] = optsw_enum(osw, &enum_label, &enum_choices, 0, 0, chooser);
        items[n++] = optsw_bool(osw, &x_label, TRUE, reporter);
        items[n++] = optsw_bool(osw, &y_label, FALSE, reporter);
        items[n++] = optsw_bool(osw, &z_label, TRUE, reporter);
        items[n++] = optsw_text(osw, &t1_label, "A text parameter", 0, NULL);
        items[n++] = optsw_text(osw, &t2_label, "Shhhhh...", OPT_TEXTMASKED, NULL);
        remove_which = n;
        items[n++] = optsw_text(osw, &t3_label, "", 0, NULL);
        items[n++] = optsw_text(osw, &t4_label, "", 0, NULL);
        items[n++] = optsw_command(osw, &txtcmd_label, texter);

        rect_construct(&p.rect, 128, 12, -1, -1);   /* pixel positioning */
        p.fixed.x = TRUE; p.fixed.y = TRUE;
        optsw_setplace(osw, items[0], &p, FALSE);

        p.rect.r_left = optsw_coltox(osw, 0);       /* character positioning */
        p.rect.r_top = -1;
        p.fixed.y = FALSE;
        optsw_setplace(osw, items[1], &p, FALSE);

        removed_items = optsw_removeitems(osw, items[remove_which], 2, FALSE);

/*
 * Install tool in tree of windows
 */
        signal(SIGWINCH, sigwinched);
        win_insert(tool->tl_windowfd);

/*
 *      main loop
 */
        tool_select(tool, 0);

/*
 * Cleanup
 */
        tool_destroy(tool);
        exit(0);
}

static
sigwinched()
{
        tool_sigwinch(tool);
}

static
confirmer(sw, ip, value)
caddr_t         sw;
caddr_t         ip;
int       value;
{
        int     result;

        confirmed = value;
        if (verbose) {
                printf("Confirmation set to %d\n", confirmed);
        }
```

```
}

static
reporter(sw, ip, value)
caddr_t         sw;
caddr_t         ip;
int       value;
{
        int     result;

        int     count;
        count = (int)optsw_getvalue(items[6], &result) +
                (int)optsw_getvalue(items[7], &result) +
                (int)optsw_getvalue(items[8], &result);
        if (count == 3) {
                count = 2;
        }
        optsw_setvalue(sw, items[5], count);
}

static
chooser(sw, ip, value)
caddr_t         sw;
caddr_t         ip;
int       value;
{
        int     result;

        if (verbose) {
                printf("Choice set to %d\n", value);
        }
}

static
dumper(sw, ip)
caddr_t         sw;
caddr_t         ip;
{
        int     result;

        if (verbose) {
                optsw_dumpsw(stdout, sw, TRUE);
        }
}

static
quitter(sw,ip)
caddr_t         sw;
caddr_t         ip;
{
        int     result;

        if (verbose) {
                printf("Quit invoked\n");
        }
        if (!confirmed) {
                if (verbose) {
                        printf("but not confirmed.\n");
                }
```

```
            return;
        }
        tool_done(tool);
}

static char             buf1[1024];
static char             buf2[1024];
static struct string_buf  str1 = { 1024, buf1 };
static struct string_buf  str2 = { 1024, buf2 };

static
texter(sw, ip)
caddr_t  sw;
caddr_t         ip;
{
        int     result;

        if (verbose) {
                result = optsw_getvalue(items[9], &str1);
                result = optsw_getvalue(items[10], &str2);
                switch (optsw_getvalue(items[5], &result)) {
                  case 0:       printf("Mum's the word.\n");
                                break;
                  case 2:       printf("First field: %s\n", buf1);
                  case 1:       printf("Second field: %s\n", buf2);
                }
        }
        if (removed_items != 0) {
                optsw_restoreitems(osw, items[remove_which],
                                removed_items, TRUE);
                removed_items = 0;
        } else {
                removed_items = optsw_removeitems(osw, items[remove_which],
                                                2, TRUE);
        }
}

static
verboser(sw, ip, val)
caddr_t         sw, val;
caddr_t         ip;
{
        verbose = (int) val;
}
```

## B.4. icontool.c Code

Code for the *icontool.c* follows.

```
#ifndef lint
static char  sccsid[] = "@(#)icontool.c 1.6 84/01/18 Sun Micro";
#endif
/*
 *      Sun Microsystems Inc.
 *
 *      icontool:       bitmap editor for icons & cursors
 *
 */
```

```
#include <suntool/tool_hs.h>
#include <sys/stat.h>
#include <stdio.h>
#include <errno.h>
#include "patches.h"
#include <suntool/msgsw.h>
#include <suntool/optionsw.h>

extern char   *sys_errlist[];
extern int     errno;

#define ICONIC          1
#define ICON_SIZE 8
#define CURSOR          0
#define CURSOR_SIZE    (ICON_SIZE * 4)

#define      MSG_HEIGHT       24
#define      PROOF_SIDE       96
#define      PROOF_MARGIN     16
#define      OPTIONS_HEIGHT PROOF_SIDE
#define      CANVAS_DISPLAY (CURSOR_SIZE * 16)
#define      CANVAS_MARGIN 16
#define      CANVAS_SIDE     (CANVAS_DISPLAY +  2 * CANVAS_MARGIN)
#define BIG          2048

static u_int   icon_array[128];
mpr_static(icon_pr, 64, 64, 1, icon_array);
static u_int   new_cursor_array[8];
mpr_static(new_cursor_pr, 16, 16, 1, new_cursor_array);
static struct cursor  new_cursor = {
                        0, 0,
                        PIX_SRC ^ PIX_DST,
                        &new_cursor_pr
};

static u_int   main_cursor_array[8] = {
                    0xC000E000, 0xF000F800,
                    0xFC00F000, 0x90001800,
                    0x18000C00, 0x0C000600,
                    0x06000300, 0x03000100
               };
mpr_static(main_cursor_pr, 16, 16, 1, main_cursor_array);
static struct cursor  main_cursor = {
                        0, 0,
                        PIX_SRC | PIX_DST,
                        &main_cursor_pr
};


/*                      general tool area                       */

#include "icontool.icon"
mpr_static(my_icon_pr, 64, 64, 1, icon_data);
static struct icon      my_icon = {
                        TOOL_ICONWIDTH, TOOL_ICONHEIGHT, NULL,
                        {0, 0, TOOL_ICONWIDTH, TOOL_ICONHEIGHT},
                        &my_icon_pr, {0, 0, 0, 0}, NULL, NULL, 0
                    };
```

```
static char             tool_name[] = "Icon Tool 1.0";
static struct rect      tool_rect;
static struct tool      *tool;

static                  sigwinched();


/*                          error message area                  */

static struct toolsw  *msg_sw;
struct msgsubwindow         *msw;


/*                          painting area                       */

static struct toolsw  *canvas_sw;
static struct pixwin  *canvas_pixwin;
static struct pixrect *canvas_pr;
static struct pixrect *fill_pr;
static int            (*canvas_reader)();
static                  canvas_sighandler();
static                  canvas_selected();
static                  canvas_basereader();
static                  canvas_tracker();
static                  set_canvas_tracker();
static                  reset_canvas_reader();
static                  canvas_feedback();
static                  wait_legal_mouse();


/*                          result-display area                 */

static struct toolsw  *proof_sw;
static struct pixwin  *proof_pixwin;
static struct pixrect *proof_pr;
static  proof_sighandler();


/*                          commands and options area           */

static struct toolsw  *options_sw;
caddr_t                     osw;

/* labels for items in the order they occur; enum values appear below */

caddr_t                 mode_item;
struct typed_pair  mode_label = {IM_TEXT, "Draw a" };
void            mode_proc();

caddr_t                 label_item;
struct typed_pair  name_label =
                    {IM_TEXT, "Left paints, Middle erases " };

caddr_t                 quit_item;
struct typed_pair  quit_label = {IM_TEXT, "Quit" };
void            quit_proc();

caddr_t                 load_item;
struct typed_pair  load_label = {IM_TEXT, "Load" };
```

```
void            load_proc();

caddr_t             store_item;
struct typed_pair  store_label = {IM_TEXT, "Store" };
void            store_proc();

caddr_t             fname_item;
struct typed_pair  file_label = {IM_TEXT, "File" };

caddr_t             fill_item;
struct typed_pair  fill_label = {IM_TEXT, "Fill" };
void            fill_proc();

caddr_t             fill_value_item;
struct typed_pair  fill_value_label = {IM_TEXT, "with" };
void            fill_value_proc();

caddr_t             invert_item;
struct typed_pair  invert_label = {IM_TEXT, "Invert" };
void            invert_proc();

caddr_t             fill_op_item;
struct typed_pair  fill_op_label = {IM_TEXT, "Load / Fill should" };
void            fill_op_proc();

caddr_t             paint_op_item;
struct typed_pair  paint_op_label = {IM_TEXT, "Cursor op" };
void            paint_op_proc();
int             paint_op_removed = FALSE;

caddr_t             bkgrnd_value_item;
struct typed_pair  bkgrnd_value_label = {IM_TEXT, "Proof background" };
void            bkgrnd_proc();

/*    Values for enums above                                    */

#define OP_OR     0                 /* paint ops        */
#define OP_XOR    1

#define OP_REPLACE    0             /* load / fill ops    */
#define OP_MERGE      1

#define GR_WHITE      0             /* gray codes             */
#define GR_GRAY25     1
#define GR_ROOT_GRAY      2
#define GR_GRAY50     3
#define GR_GRAY75     4
#define GR_BLACK      5

void
mode_proc(optsw, ip, val)
caddr_t         optsw;
caddr_t         ip;
u_int    val;
{
        set_state(val);
}

#define IC_MODECOUNT 2
```

```
char            *mode_values[IC_MODECOUNT+1] = { "Cursor", "Icon" };
struct typed_pair  mode_choices = {IM_TEXTVEC, (caddr_t)mode_values };

#define IC_GRAYCOUNT 6
char            *gray_values[IC_GRAYCOUNT+1] = {
      "White", "25%", "Root Gray", "50%", "75%", "Black" };
struct typed_pair  gray_choices = {IM_TEXTVEC, (caddr_t)gray_values };

#define IC_FOPCOUNT   2
char            *fill_op_values[IC_FOPCOUNT+1] = { "Replace", "Merge" };
struct typed_pair  fill_op_choices = {IM_TEXTVEC, (caddr_t)fill_op_values };

#define IC_POPCOUNT   2
char            *paint_op_values[IC_POPCOUNT+1] = { "OR", "XOR" };
struct typed_pair  paint_op_choices = {IM_TEXTVEC, (caddr_t)paint_op_values };


/*                    general globals                                    */

int     errno;

static u_int   cur_x, cur_y,
               cur_op,
               cell_count,
               cell_size,
               state =     -1;     /* so first set_state really does */

char           file_default[] = "test.icon";
char           file_name[1024];

struct pixfont *font;
FILE        *sysout = stderr;

main(argc,argv)
int      argc;
char   **argv;
{
       tool = tool_create(tool_name, TOOL_NAMESTRIPE, NULL, &my_icon);

       font = pf_default();
       msg_sw = msgsw_createtoolsubwindow(tool, "", -1, MSG_HEIGHT, "", font);
       msw = (struct msgsubwindow *)msg_sw->ts_data;

       proof_sw = tool_createsubwindow(tool, "", PROOF_SIDE, PROOF_SIDE);
       init_proof();

       options_sw = optsw_createtoolsubwindow(tool, "", -1, OPTIONS_HEIGHT);
       init_options();

       canvas_sw = tool_createsubwindow(tool, "", -1, CANVAS_SIDE);
       init_canvas();

       fix_tool_rect();
       set_state(CURSOR);
       fill_value_proc(NULL, NULL, GR_ROOT_GRAY);
       bkgrnd_proc(NULL, NULL, GR_ROOT_GRAY);
       signal(SIGWINCH, sigwinched);
       tool_install(tool);
```

```
        tool_select(tool, 0);

        tool_destroy(tool);
        exit(0);
}

fix_tool_rect()
{
        if (wmgr_iswindowopen(tool->tl_windowfd) ) {
                win_getrect(tool->tl_windowfd, &tool_rect);
        } else {
                win_getsavedrect(tool->tl_windowfd, &tool_rect);
        }
        tool_rect.r_width = 2*tool_borderwidth(tool) +
                        max(PROOF_SIDE + optsw_coltox(osw, 64) +
                            tool_subwindowspacing(tool),
                            CANVAS_SIDE);
        tool_rect.r_height = MSG_HEIGHT
                        + CANVAS_SIDE + PROOF_SIDE
                        + tool_stripeheight(tool)
                        + tool_borderwidth(tool)
                        + 2*tool_subwindowspacing(tool);
        if (rect_bottom(&tool_rect) >= 800) {
                tool_rect.r_top -= rect_bottom(&tool_rect) - 799;
        }
        if (wmgr_iswindowopen(tool->tl_windowfd) ) {
                win_setrect(tool->tl_windowfd, &tool_rect);
        } else {
                win_setsavedrect(tool->tl_windowfd, &tool_rect);
        }
}


set_state(which)
{
        if (state == which) {
                return;
        }
        if ( (state = which) == CURSOR) {
                canvas_pr = &new_cursor_pr;
                cell_size = CURSOR_SIZE;
                if (paint_op_removed) {
                        optsw_restoreitems(osw, paint_op_item, 1, TRUE);
                        paint_op_removed = FALSE;
                }
        } else {
                canvas_pr = &icon_pr;
                cell_size = ICON_SIZE;
                if (!paint_op_removed) {
                        optsw_removeitems(osw, paint_op_item, 1, TRUE);
                        paint_op_removed = TRUE;
                }

        }
        optsw_setvalue(osw, mode_item, which);
        set_cursor();
        cell_count = CANVAS_DISPLAY / cell_size;
        paint_proof();
        paint_canvas();
```

```
}

set_cursor()
{
        if (state == ICONIC) {
                win_setcursor(proof_sw->ts_windowfd, &main_cursor);
        } else {
                win_setcursor(proof_sw->ts_windowfd, &new_cursor);
        }
}

static
sigwinched()
{
        tool_sigwinch(tool);
}

nullproc()
{
        return;
}


int   (*saved_handler)();
int     saved_mask;
int     clear_message();

bitch(format, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8)
char   *format, *arg1, *arg2, *arg3, *arg4, *arg5, *arg6, *arg7, *arg8;
{
        char    buf[256];

        sprintf(buf,format, arg1, arg2, arg3, arg4, arg5, arg6, arg7, arg8);
        msgsw_setstring(msw, buf);
        saved_mask = tool->tl_io.tio_inputmask;
        tool->tl_io.tio_inputmask = (1 << tool->tl_windowfd) +
                                    (1 << canvas_sw->ts_windowfd) +
                                    (1 << options_sw->ts_windowfd);
        if (tool->tl_io.tio_selected != clear_message) {
                saved_handler = tool->tl_io.tio_selected;
        }
        tool->tl_io.tio_selected = clear_message;
}

clear_message(datum, ibits, obits, ebits, timer)
caddr_t                 datum;
u_int           *ibits, *obits, *ebits;
struct timeval **timer;
{
        msgsw_setstring(msw, "");
        tool->tl_io.tio_selected = saved_handler;
        tool->tl_io.tio_inputmask = saved_mask;
}

/*
 *      Proof Section
 */

init_proof()
```

```
{
        struct inputmask  mask;

        input_imnull(&mask);
        win_setinputmask(proof_sw->ts_windowfd, &mask, NULL, WIN_NULLLINK);
        proof_sw->ts_io.tio_handlesigwinch = proof_sighandler;
        proof_sw->ts_destroy = nullproc;
        proof_pixwin = pw_open(proof_sw->ts_windowfd);
}

static
proof_sighandler(sw_data)
caddr_t         sw_data;
{
        pw_damaged(proof_pixwin);
        paint_proof();
        pw_donedamaged(proof_pixwin);
        proof_sw->ts_width = win_getwidth(proof_sw->ts_windowfd);
        proof_sw->ts_height = win_getheight(proof_sw->ts_windowfd);
}

paint_proof()
{
        int     x, y;

        for (y = 0; y < proof_sw->ts_height; y += 64) {
                for (x = 0; x < proof_sw->ts_width; x += 64) {
                        pw_write(proof_pixwin, x, y, 64, 64,
                                PIX_SRC, proof_pr, 0, 0);
                }
        }
        if (state == ICONIC) {
                paint_proof_icon();
        }
}

paint_proof_icon()
{
        pw_write(proof_pixwin, PROOF_MARGIN, PROOF_MARGIN, 64, 64,
                PIX_SRC, &icon_pr, 0, 0);
}

/*
 *      Options subwindow section
 */
init_options()
{
        win_setcursor(options_sw->ts_windowfd, &main_cursor);

        osw = options_sw->ts_data;

        mode_item = optsw_enum(osw, &mode_label, &mode_choices,
                                0, CURSOR, mode_proc);
        start_new_line(mode_item);

        label_item = optsw_label(osw, &name_label);

        quit_item = optsw_command(osw, &quit_label, quit_proc);
```

```
        load_item = optsw_command(osw, &load_label, load_proc);
        start_new_line(load_item);

        store_item = optsw_command(osw, &store_label, store_proc);

        fname_item = optsw_text(osw, &file_label, file_default, 0, NULL);

        fill_item = optsw_command(osw, &fill_label, fill_proc);
        start_new_line(fill_item);

        fill_value_item = optsw_enum(osw, &fill_value_label,
                                     &gray_choices,
                                     0, 2, fill_value_proc);

        invert_item = optsw_command(osw, &invert_label, invert_proc);

        fill_op_item = optsw_enum(osw, &fill_op_label,
                                  &fill_op_choices,
                                  0, 0, NULL);
        start_new_line(fill_op_item);

        paint_op_item = optsw_enum(osw, &paint_op_label,
                                   &paint_op_choices,
                                   0, 1, paint_op_proc);

        bkgrnd_value_item = optsw_enum(osw, &bkgrnd_value_label,
                                       &gray_choices,
                                       0, 2, bkgrnd_proc);
        start_new_line(bkgrnd_value_item);
}

start_new_line(item)        /* special-case routine for legibility */
caddr_t     item;
{
        struct item_place  p;

        rect_construct(&p.rect, 0, -1, -1, -1);
        p.fixed.x = TRUE;
        p.fixed.y = p.fixed.w = p.fixed.h = FALSE;
        optsw_setplace(osw, item, &p, 0);
}

/*      handlers for the various option items, in their creation order */

void
quit_proc(optsw, ip)
caddr_t         optsw;
caddr_t         ip;
{
        msgsw_setstring(msw,
        "Please confirm with the left mouse button, or cancel with right or middle.");
        if (cursor_confirm(canvas_sw->ts_windowfd)) {
                tool_done(tool);
        } else {
                bitch("Quit cancelled.");
        }
}

void
```

```
load_proc(optsw, ip)
caddr_t         optsw;
caddr_t         ip;
{
        int                     c;
        u_int                   count, data[256], *dp, result;
        u_int                   op, mode, chunks;
#define SHORT_CHUNKS 2
#define LONG_CHUNKS  1
        struct string_buf       file_name_buf;
        FILE                    *fd;

        file_name_buf.limit = 1024;
        file_name_buf.data = file_name;
        result = optsw_getvalue(fname_item, &file_name_buf);
        if (result < 0) {
                bitch("Trouble: icontool couldn't read the filename");
                sleep(10);
                exit(-1);
        }
        fd = fopen(file_name, "r");
        if (fd == NULL) {
                bitch("Sorry, couldn't open %s", file_name);
                return;
        }
        while ( (c= getc(fd)) != '{') {             /* matching } */
                if (c==EOF) {
                        bitch("Sorry, I need an array of shorts or unsigneds");
                        return;
                }
        }
        dp = data;
        count =0;
        do {    result = fscanf(fd, " 0x%X,", dp++);
                count++;
        } while (result == 1);
        fclose(fd);

        switch (--count) {
          case  8:      mode = CURSOR;
                        chunks = LONG_CHUNKS;
                        break;
          case 16:      mode = CURSOR;
                        chunks = SHORT_CHUNKS;
                        break;
          case 128:     mode = ICONIC;
                        chunks = LONG_CHUNKS;
                        break;
          case 256:     mode = ICONIC;
                        chunks = SHORT_CHUNKS;
                        break;
          default:      bitch("Sorry, I don't understand that array.");
                        return;
        }
        op = optsw_getvalue(fill_op_item, &result);
        if (mode == CURSOR) {
                dp = new_cursor_array;
        } else {
                dp = icon_array;
```

```
        }
        if (chunks == LONG_CHUNKS) {
                if (op == OP_REPLACE) {
                        replace_longs(dp, data, count);
                } else {
                        merge_longs(dp, data, count);
                }
        } else {
                if (op == OP_REPLACE) {
                        replace_shorts(dp, data, count);
                } else {
                        merge_shorts(dp, data, count);
                }
        }
        state = -1;
        set_state(mode);
}

static
replace_longs(target, source, count)
int     *target, *source, count;
{
        while (count-- > 0) {
                target[count] = source[count];
        }
}

static
merge_longs(target, source, count)
int   *target, *source, count;
{
        while (count-- > 0) {
                target[count] |= source[count];
        }
}

static
replace_shorts(target, source, count)
int     *target, *source, count;
{
        while (count-- > 0) {
                target[count/2] = source[count];
                target[count/2] |= (source[count-1] << 16);
                count -= 1;
        }
}

static
merge_shorts(target, source, count)
int     *target, *source, count;
{
        while (count-- > 0) {
                target[count/2] |= source[count];
                target[count/2] |= (source[count-1] << 16);
                count -= 1;
        }
}

void
```

```
store_proc(optsw, ip)
caddr_t         optsw;
caddr_t         ip;
{
        int                     i, limit, result, size;
        u_int                   *data;
        char                    *token;
        struct string_buf       file_name_buf;
        FILE                    *fd;
        struct stat             stat_buf;

        file_name_buf.limit = 1024;
        file_name_buf.data = file_name;
        if (state == CURSOR) {
                size = 16;
                data = new_cursor_array;
                token = "cursor";
        } else {
                size = 64;
                data = icon_array;
                token = "icon";
        }

        result = optsw_getvalue(fname_item, &file_name_buf);
        if (stat(file_name, &stat_buf) == -1)  {
                if (errno != ENOENT)  {
                        bitch("Sorry, %s", sys_errlist[errno]);
                        return;
                }
        } else {                        /* stat succeeded; file exists        */
                bitch("%s exists; please confirm overwrite.", file_name);
                if (!cursor_confirm(canvas_sw->ts_windowfd))  {
                        return;
                }
        }
        fd = fopen(file_name, "w");
        if (fd == NULL)  {
                bitch("Sorry, can't write to %s", file_name);
                return;
        }
        fprintf(fd, "\nstatic unsigned%s_data[%d] = {\n",
                token, size * size / 32);

        limit = size * size / 128;
        for (i=0; i<limit;) {
                fprintf(fd, "0x%-08X, 0x%-08X, 0x%-08X, 0x%-08X",
                        data[0], data[1], data[2], data[3]);
                data += 4;
                if (++i < limit)  {
                        fputs(",\n", fd);
                }
        }
        fputs("\n};\n", fd);
        fclose(fd);
}

void
fill_proc(optsw, ip, val)
{
```

```
        int     op, x, y, result;

        switch (optsw_getvalue(fill_op_item, &result) ) {
          case OP_REPLACE:      op = PIX_SRC; break;
          case OP_MERGE: op = PIX_SRC | PIX_DST; break;
          default:      bitch("Trouble: fill doesn't know what to do.");
        }
        for (y = 0; y < cell_count; y += 64) {
                for (x = 0; x < cell_count; x += 64) {
                        pr_rop(canvas_pr, x, y, cell_count, cell_count,
                                        op, fill_pr, 0, 0);
                }
        }
        paint_canvas();
        if (state == ICONIC) {
                paint_proof_icon();
        } else {
                set_cursor();
        }
}

void
fill_value_proc(optsw, ip, val)
{
        switch (val) {
           case GR_WHITE:       fill_pr = &white_patch;
                                break;
           case GR_GRAY25:      fill_pr = &gray25_patch;
                                break;
           case GR_ROOT_GRAY:       fill_pr = &root_gray_patch;
                                break;
           case GR_GRAY50:      fill_pr = &gray50_patch;
                                break;
           case GR_GRAY75:      fill_pr = &gray75_patch;
                                break;
           case GR_BLACK:       fill_pr = &black_patch;
                                break;
           default:             fill_pr = &root_gray_patch;
        }
}

void
invert_proc()
{
        pr_rop(canvas_pr, 0, 0, cell_count, cell_count,
                PIX_NOT(PIX_DST), 0, 0, 0);
        paint_canvas();
        if (state == ICONIC) {
                paint_proof_icon();
        } else {
                set_cursor();
        }
}

void
paint_op_proc(optsw, ip, val)
caddr_t         optsw;
caddr_t         ip;
u_int   val;
```

```
{
        u_int  op;

        if (val == OP_XOR) {
                new_cursor.cur_function = PIX_SRC ^ PIX_DST;
        } else {
                new_cursor.cur_function = PIX_SRC | PIX_DST;
        }
        set_cursor();
}

void
bkgrnd_proc(optsw, ip, val)
{
        switch (val) {
           case GR_WHITE:       proof_pr = &white_patch;
                                break;
           case GR_GRAY25:      proof_pr = &gray25_patch;
                                break;
           case GR_ROOT_GRAY:       proof_pr = &root_gray_patch;
                                break;
           case GR_GRAY50:      proof_pr = &gray50_patch;
                                break;
           case GR_GRAY75:      proof_pr = &gray75_patch;
                                break;
           case GR_BLACK:       proof_pr = &black_patch;
                                break;
           default:             proof_pr = &root_gray_patch;
        }
        paint_proof();
}

/*
 *      Canvas Section
 */

init_canvas()
{
        struct inputmask mask;

        canvas_reader = canvas_basereader;
        canvas_pixwin = pw_open(canvas_sw->ts_windowfd);
        canvas_sw->ts_io.tio_selected = canvas_selected;
        canvas_sw->ts_io.tio_handlesigwinch = canvas_sighandler;
        canvas_sw->ts_destroy = nullproc;

        input_imnull(&mask);
        win_setinputcodebit(&mask, MS_LEFT );
        win_setinputcodebit(&mask, MS_MIDDLE);
        win_setinputcodebit(&mask, LOC_MOVEWHILEBUTDOWN);
        win_setinputcodebit(&mask, LOC_STILL);
        win_setinputcodebit(&mask, LOC_WINEXIT);
        mask.im_flags |= IM_NEGEVENT;
        win_setinputmask(canvas_sw->ts_windowfd, &mask, NULL, WIN_NULLLINK);
        win_setcursor(canvas_sw->ts_windowfd, &main_cursor);
        cur_x = cur_y = -1;
}

static
```

```
canvas_selected(nullsw, ibits, obits, ebits, timer)
caddr_t             *nullsw;
int           *ibits, *obits, *ebits;
struct timeval      *timer;
{
        struct inputevent  ie;

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

static
canvas_basereader(ie)
struct  inputevent  *ie;
{
        if (win_inputnegevent(ie) ) {
                return;
        }
        switch (ie->ie_code) {
          case MS_LEFT:       cur_op = 1;
                              break;
          case MS_MIDDLE:   cur_op = 0;
                              break;
          default:          return;           /* ignore all other input  */
        }
        set_canvas_tracker();
        canvas_feedback(ie);
}

static
canvas_tracker(ie)
struct inputevent  *ie;
{
        if (win_inputnegevent(ie))  {
                switch (ie->ie_code) {
                  case MS_LEFT:             /*  mouse button up        */
                  case MS_MIDDLE:
                        reset_canvas_reader();
                        if (state == ICONIC)  {
                                paint_proof_icon();
                        }
                }
                return;
        }
        switch (ie->ie_code) {
          case LOC_WINEXIT:     reset_canvas_reader();
                                if (state == ICONIC)  {
                                        paint_proof_icon();
                                }
                                return;
          case MS_LEFT:                     /*  two buttons down!          */
          case MS_MIDDLE: cur_op = -1;
                                canvas_reader = wait_legal_mouse;
                                return;
          case LOC_STILL:
```

```
        case LOC_MOVEWHILEBUTDOWN:
                        canvas_feedback(ie);
                        return;
        }                       /* ignore all other input */
}

static
wait_legal_mouse(ie)
struct inputevent *ie;
{
        if (ie->ie_code == LOC_WINEXIT) {
                reset_canvas_reader();
                return;
        }
        if (win_inputnegevent(ie)) {
                switch (ie->ie_code) {
                  case MS_LEFT:         cur_op = 0;
                                        break;
                  case MS_MIDDLE:       cur_op = 1;
                                        break;
                  default:              return;
                }
        set_canvas_tracker();
        canvas_feedback(ie);
        }
}

static
set_canvas_tracker()
{
        cur_x = cur_y = -1;
        canvas_reader = canvas_tracker;
}

static
reset_canvas_reader()
{
        canvas_reader = canvas_basereader;
        cur_op = -1;
}

static
canvas_feedback(ie)
struct inputevent *ie;
{
        register int    new_x, new_y, color;

        if (ie->ie_code == LOC_STILL && state == ICONIC) {
                paint_proof_icon();
                return;
        }
        if (ie->ie_locx < CANVAS_MARGIN || ie->ie_locy < CANVAS_MARGIN) {
                return;
        }
        new_x = (ie->ie_locx - CANVAS_MARGIN) / cell_size;
        new_y = (ie->ie_locy - CANVAS_MARGIN) / cell_size;
        if (new_x >= cell_count || new_y >= cell_count) {
                return;
        }
```

```
        color = pr_get(canvas_pr, new_x, new_y);
        if (new_x == cur_x && new_y == cur_y && cur_op == color)
                return;
        cur_x = new_x;
        cur_y = new_y;
        paint_cell(new_x, new_y, cur_op);
        pr_put(canvas_pr, new_x, new_y, cur_op);
        if (state == CURSOR) {
                set_cursor();
        }
}
static
canvas_sighandler()
{
        pw_damaged(canvas_pixwin);
        paint_canvas();
        pw_donedamaged(canvas_pixwin);
}

paint_canvas()
{
        register int    x, y;
        struct rect     r;

        pw_writebackground(canvas_pixwin, 0, 0, BIG, BIG, PIX_CLR);
        r.r_left = CANVAS_MARGIN;
        r.r_width = cell_count * cell_size;
        r.r_height = cell_size;
        pw_vector(canvas_pixwin, CANVAS_MARGIN, CANVAS_MARGIN,
                                rect_right(&r), CANVAS_MARGIN,
                                PIX_SET, 1);
        pw_vector(canvas_pixwin, CANVAS_MARGIN, CANVAS_MARGIN,
                                CANVAS_MARGIN, rect_right(&r),
                                PIX_SET, 1);
        for (y = 0; y < cell_count; y++) {
                r.r_top = CANVAS_MARGIN + cell_size * y;
                pw_lock(canvas_pixwin, &r);
                for (x = 0; x < cell_count; x++) {
                        if (pr_get(canvas_pr, x,y)) {
                                paint_cell(x, y, 1);
                        }
                }
                pw_unlock(canvas_pixwin);
        }
        r.r_top = CANVAS_MARGIN;
        r.r_width += 1;
        r.r_height = cell_count * cell_size + 1;
        pw_lock(canvas_pixwin, &r);
        pw_vector(canvas_pixwin, rect_right(&r), CANVAS_MARGIN,
                                rect_right(&r), rect_bottom(&r),
                                PIX_SET, 1);
        pw_vector(canvas_pixwin, CANVAS_MARGIN, rect_bottom(&r),
                                rect_right(&r), rect_bottom(&r),
                                PIX_SET, 1);
        pw_unlock(canvas_pixwin);
}
paint_cell(x, y, color)
int     x, y, color;
{
```

```
        register int    dx, dy, dim;

        dx = CANVAS_MARGIN + cell_size*x + 1;
        dy = CANVAS_MARGIN + cell_size*y + 1;
        dim = cell_size - 1;
        pw_write(canvas_pixwin, dx, dy, dim, dim, PIX_SRC,
                (color ? &gray50_patch : &white_patch), 1, 1);
}
```

# Appendix C

# Sample Graphics Programs

Use these sample programs as templates for your own graphics programs. Included is code for a bouncing ball demonstration (*bouncedemo.c*) and for a "movie camera" program (*framedemo.c*, which displays files sequentially like movie frames for producing a rotating globe for example. The source files for these and other graphics demos are found on */usr/suntool/src/*demo.c*.

## C.1. bouncedemo.c Code

Code for the *bouncedemo.c* follows.

```
#ifndef lint
static  char sccsid[] = "@(#)bouncedemo.c 1.5 83/08/26 Sun Micro";
#endif

/*
 * Sun Microsystems, Inc.
 */

/*
 *      Overview:     Bouncing ball demo in window
 */

#include <sys/types.h>
#include "pixrect/pixrect.h"
#include "sunwindow/rect.h"
#include "sunwindow/rectlist.h"
#include "sunwindow/pixwin.h"
#include "suntool/gfxsw.h"

main(argc, argv)
        int argc;
        char **argv;
{
        short  x, y, vx, vy, z, ylastcount, ylast;
        short  Xmax, Ymax, size;
        struct rect rect;
        struct gfxsubwindow *gfx = gfxsw_init(0, argv);

Restart:
        win_getsize(gfx->gfx_windowfd, &rect);
        Xmax = rect_right(&rect);
        Ymax = rect_bottom(&rect);
        if (Xmax < Ymax)
```

```
        size = Xmax/29+ 1;
    else
        size = Ymax/29+ 1;
    x=rect.r_left;
    y=rect.r_top;
    vx=4;
    vy=0;
    ylast=0;
    ylastcount=0;
    pw_writebackground(gfx->gfx_pixwin, 0, 0, rect.r_width, rect.r_height,
       PIX_SRC);
    while (gfx->gfx_reps) {
        if (gfx->gfx_flags&GFX_DAMAGED)
            gfxsw_handlesigwinch(gfx);
        if (gfx->gfx_flags&GFX_RESTART) {
            gfx->gfx_flags &= ~GFX_RESTART;
            goto Restart;
        }
        if (y==ylast) {
            if (ylastcount++ > 5)
                goto Reset;
        } else {
            ylast = y;
            ylastcount = 0;
        }
        pw_writebackground(gfx->gfx_pixwin, x, y, size, size,
           PIX_NOT(PIX_DST));
        x=x+vx;
        if (x>(Xmax-size)) {
            /*
             * Bounce off the right edge
             */
            x=2*(Xmax-size)-x;
            vx= -vx;
        } else if (x<rect.r_left) {
            /*
             * bounce off the left edge
             */
            x= -x;
            vx= -vx;
        }
        vy=vy+1;
        y=y+vy;
        if (y>=(Ymax-size)) {
            /*
             * bounce off the bottom edge
             */
            y=Ymax-size;
            if (vy<size)
                vy=1-vy;
            else
                vy=vy / size - vy;
            if (vy==0)
                goto Reset;
        }
        for (z=0; z<=1000; z++);
        continue;
Reset:
        if (--gfx->gfx_reps <= 0)
```

```
                break;
            x=rect.r_left;
            y=rect.r_top;
            vx=4;
            vy=0;
            ylast=0;
            ylastcount=0;
        }
        gfxsw_done(gfx);
}
```

## C.2. framedemo.c Code

Code for the *framedemo.c* follows.

```
#ifndef lint
static  char sccsid[] = "@(#)framedemo.c 1.10 84/01/11 SMI";
#endif

/*
 * Sun Microsystems, Inc.
 */

/*
 *      Overview:       Frame displayer in windows.  Reads in all the
 *                      files of form "frame.xxx" in working directory &
 *                      displays them like a movie.
 *                      See constants below for limits.
 */

#include <stdio.h>
#include <sys/types.h>
#include <sys/file.h>
#include <sys/time.h>
#include "pixrect/pixrect.h"
#include "pixrect/pr_util.h"
#include "pixrect/bw1var.h"
#include "pixrect/memvar.h"
#include "sunwindow/rect.h"
#include "sunwindow/rectlist.h"
#include "sunwindow/pixwin.h"
#include "sunwindow/win_input.h"
#include "sunwindow/win_struct.h"
#include "suntool/gfxsw.h"

#define     MAXFRAMES       1000
#define     FRAMEWIDTH      256
#define     FRAMEHEIGHT     256
#define     USEC_INC    50000
#define     SEC_INC         1

static  struct pixrect *mpr[MAXFRAMES];
static  struct timeval timeout = {SEC_INC,USEC_INC}, timeleft;
static  char s[] = "frame.xxx";
static  struct gfxsubwindow *gfx;
static  int frames, framenum, ximage, yimage;
static  struct rect rect;
```

```
main(argc, argv)
        int argc;
        char **argv;
{
        int     fd, framedemo_selected();
        struct inputmask im;

        for (frames = 0; frames < MAXFRAMES; frames++) {
                sprintf(&s[6], "%d", frames + 1);
                fd = open(s, O_RDONLY, 0);
                if (fd == -1) {
                        break;
                }
                mpr[frames] = mem_create(FRAMEWIDTH, FRAMEHEIGHT, 1);
                read(fd, mpr_d(mpr[frames])->md_image,
                   FRAMEWIDTH*FRAMEHEIGHT/8);
                close(fd);
                }
        if (frames == 0) {
           printf("Couldn't find any 'frame.xx' files in working directory\n");
           return;
        }
        /*
         * Initialize gfxsw ("take over" kind)
         */
        gfx = gfxsw_init(0, argv);
        /*
         * Set up input mask
         */
        input_imnull(&im);
        im.im_flags |= IM_ASCII;
        gfxsw_setinputmask(gfx, &im, &im, WIN_NULLLINK, 1, 0);
        /*
         * Main loop
         */
        framedemo_nextframe(1);
        timeleft = timeout;
        gfxsw_select(gfx, framedemo_selected, 0, 0, 0, &timeleft);
        /*
         * Cleanup
         */
        gfxsw_done(gfx);
}
framedemo_selected(gfx, ibits, obits, ebits, timer)
        struct gfxsubwindow *gfx;
        int     *ibits, *obits, *ebits;
        struct timeval **timer;
{
        if ((*timer && ((*timer)->tv_sec == 0) && ((*timer)->tv_usec == 0)) ||
           (gfx->gfx_flags & GFX_RESTART)) {
                /*
                 * Our timer expired or restart is true so show next frame
                 */
                if (gfx->gfx_reps)
                        framedemo_nextframe(0);
                else
                        gfxsw_selectdone(gfx);
        }
        if (*ibits & (1 << gfx->gfx_windowfd)) {
```

```
        struct inputevent event;

        /*
         * Read input from window
         */
        if (input_readevent(gfx->gfx_windowfd, &event)) {
                perror("framedemo");
                return;
        }
        switch (event.ie_code) {
        case 'f': /* faster usec timeout */
                if (timeout.tv_usec >= USEC_INC)
                        timeout.tv_usec -= USEC_INC;
                else {
                        if (timeout.tv_sec >= SEC_INC) {
                                timeout.tv_sec -= SEC_INC;
                                timeout.tv_usec = 1000000-USEC_INC;
                        }
                }
                break;
        case 's': /* slower usec timeout */
                if (timeout.tv_usec < 1000000-USEC_INC)
                        timeout.tv_usec += USEC_INC;
                else {
                        timeout.tv_usec = 0;
                        timeout.tv_sec += 1;
                }
                break;
        case 'F': /* faster sec timeout */
                if (timeout.tv_sec >= SEC_INC)
                        timeout.tv_sec -= SEC_INC;
                break;
        case 'S': /* slower sec timeout */
                timeout.tv_sec += SEC_INC;
                break;
        case '?': /* Help */
                printf("'s' slower usec timeout\n'f' faster usec timeout\n'S' slower sec timeout\n'F' faster sec timeout\n");
                /*
                 * Don't reset timeout
                 */
                return;
        default:
                gfxsw_inputinterrupts(gfx, &event);
        }
    }
    *ibits = *obits = *ebits = 0;
    timeleft = timeout;
    *timer = &timeleft;
}

framedemo_nextframe(firsttime)
    int     firsttime;
{
    int     restarting = gfx->gfx_flags&GFX_RESTART;

    if (firsttime || restarting) {
        gfx->gfx_flags &= ~GFX_RESTART;
        win_getsize(gfx->gfx_windowfd, &rect);
        ximage = rect.r_width/2-FRAMEWIDTH/2;
```

```
            yimage = rect.r_height/2-FRAMEHEIGHT/2;
            pw_writebackground(gfx->gfx_pixwin, 0, 0,
               rect.r_width, rect.r_height, PIX_CLR);
        }
        if (framenum >= frames) {
            framenum = 0;
            gfx->gfx_reps--;
        }
        pw_write(gfx->gfx_pixwin, ximage, yimage, FRAMEWIDTH, FRAMEHEIGHT,
           PIX_SRC, mpr[framenum], 0, 0);
        if (!restarting)
            framenum++;
}
```

# Appendix D

# Programming Notes

Here are useful hints for programmers who use any of the *pixrect*, *sunwindow* or *suntool* libraries.

## D.1. What Is Supported?

In each release, there may be some difference between the documentation and the actual product implementation. The documentation describes the supported implementation. In general, the documentation indicates where features are only partially implemented, and in which directions future extensions may be expected. Any necessary modifications to SunWindows are accompanied by a description of the nature of the changes and appropriate responses to them.

## D.2. Program By Example

We recommend that you try to program by example whenever possible. Take an existing program similar to what you need and modify it. Appendix B contains some sample tools and Appendix C contains some sample graphics programs. The source for these and other sample tools and graphics programs are available on */usr/suntool/src/*.c*.

## D.3. Header Files Needed

If you have problems finding the necessary header files for compiling your program, using the examples may help as many of the header files are already included. Moreover, there are certain header files that include most of the header files necessary for working at a certain level. The following table shows these header files:

Table D-1: Header Files Required

| Use | When Working at the Level of |
|---|---|
| */usr/include/suntool/tool_hs.h* | suntool tool-building facilities; includes headers needed to work at the more primitive layers as well |
| */usr/include/suntool/gfx_hs.h* | the suntool (standalone or "take over") graphics subwindow facilities; includes headers needed to work at the more primitive layers as well |
| */usr/include/sunwindow/window_hs.h* | sunwindow basic window facilities layer; includes headers needed to work at the pixrect layer as well |
| */usr/include/pixrect/pixrect_hs.h* | pixrect display primitives layer |

Include only one of the above header files plus whatever extra header files you need. In particular, you'll need to add the header file for each subwindow type that you use, the menu header file if you use menus, the selection header file if you are going to use selections, and so on. However, you'll probably only have to add a single header file for each additional increment of high-level functionality.

## D.4. Lint Libraries

SunWindows provides *lint libraries* to help you run *lint* over your program source. *Lint* catches argument mismatches and provides better type-checking than the C compiler. *Llib-lpixrect*, *llib-lsunwindow*, and *llib-lsuntool* are the source files to make the actual binary *lint*(1) libraries: *llib-lpixrect.ln*, *llib-lsunwindow.ln*, and *llib-lsuntool.ln*. These files are found on */usr/lib/lint/*.

## D.5. Library Loading Order

When loading programs, remember to load higher level libraries first, that is, *-lsuntool -lsunwindow -lpixrect*.

## D.6. Shared Text

The tools released with *suntools* rely on text sharing to reduce the memory working set. This is accomplished by placing the entire collection of tools in a single object file. This has the effect of letting each separate process share the same object code in memory. With many windows active at once this can achieve significant memory savings.

There are trade-offs using this approach. The main one is that the maximum number of per-process and non-sharable initial data pages tends to be larger. However, the paged virtual memory tends to reduce the effect of this by only having the working set paged in.

The upshot of this is that you may want to either add the tools that you create to the released shared object file or to bundle a few tools together into their own object file.

## D.7. Error Message Decoding

The default error reporting scheme described at the end of *Window Manipulation* displays a long hex number which is the *ioctl* number associated with the error. You can turn this number into a more meaningful operation name by:

- turning the two least significant digits into a decimal number;
- searching */usr/include/sunwindow/win_ioctl.h* for occurrences of this number; and
- noting the *ioctl* operation associated with this number.

This can provides a quick hint as to what is being complained about without resorting to a debugger.

## D.8. Debugging Hints

When debugging non-terminal oriented programs in the window system, there are some things that you should know to make things easier.

As discussed in the section entitled *Overlapped Windows: Imaging Facilities - Damage*, a process receives a SIGWINCH whenever one of its windows changes state. In particular, as soon as a tool issues a *tool_install*, the kernel sends it a SIGWINCH. When running as the child of a debugger, the SIGWINCH is sent to the parent debugger instead of to the tool. By default, *dbx* simply propagates the SIGWINCH to the tool, while *adb* traps, leaving the tool suspended until the user continues from *adb*. This behavior is not peculiar to SIGWINCH: *adb* traps all signals by default, while *dbx* has an initial list of signals (including SIGWINCH) that are passed on to the child process. You can instruct *adb* to pass SIGWINCH on to the child process by typing **1c:i** followed by RETURN. '1c' is the hex number for 28, which is SIGWINCH's number. Re-enable signal breaking by typing **1c:t** followed by RETURN. You can instruct *dbx* to trap on a signal by using the *catch* command.

For further details, see the entries for the individual debuggers in the *User's Manual for the Sun Workstation*. In addition, *ptrace*(2) describes the fine points of how kernel signal delivery is modified while a program is being debugged.

The two debuggers differ also in their abilities to interrupt programs built using tool windows. *Dbx* knows how to do interrupt these programs, but *adb* doesn't. See *Signals from the Control Terminal* below for an explanation.

Another situation specific to the window system is that various forms of locking are done that can get in the way of smooth debugging while working at low levels of the system. There are variables in the *sunwindow* library that disable the actual locking. These variables can be turned on from a debugger:

Table D-2: *sunwindow* Variables for Disabling Locking

| Variable | Action |
|---|---|
| *int pixwindebug* | When not zero this causes the immediate release of the display lock after locking so that the debugger is not continually getting hung by being blocked on writes to screen. Display garbage can result because of this action. |
| *int win_lockdatadebug* | When not zero, the data lock is never actually locked, preventing the debugger from being continually hung due to block writes to the screen. Unpredictable things may result because of this action that can't properly be described in this context. |
| *int win_grabiodebug* | When not zero will not actually acquire exclusive I/O access rights so that the debugger wouldn't get hung by being blocked on writes to the screen and not be able to receive input. The debugged process will only be able to do normal display locking and be able to get input only in the normal way. |
| *int fullscreendebug* | Like *win_grabiodebug* but applies to the fullscreen access package. |

Change these variables only during debugging. You can set them anytime after *main* has been called.

## D.9. Sufficient User Memory

To use the *suntool* environment comfortably requires adequate user memory for SunWindows and the Sun UNIX operating system. To achieve the best performance, reconfigure your own kernel, deleting unused device drivers. The procedure is documented in the *System Manager's Manual for the Sun Workstation*. For a workstation on the network with a single disk drive, you will be able to reclaim significant usable memory.

For the recommended amount of memory, see the *Sun Workstation Configuration Guide*.

## D.10. Coexisting with UNIX

This section discusses how a SunWindows tool interacts with traditional UNIX features in the areas of process groups, signal handling, job control and terminal emulation. If you are not familiar with these concepts, read the appropriate portions (*Process Groups, Signals*) of the *System Interface Overview* and the *signal*(3) and *tty*(4) entries in the *System Interface Manual for the Sun Workstation*.

This discussion explicitly notes those places where the shells and debuggers interact differently with a tool.

## D.10.1. Tool Initialization and Process Groups

System calls made by the library code in a tool affect the signals that will be sent to the tool. A tool acts like any program when first started: it inherits the process group and control terminal group from its parent process. However, when a tool calls *tool_create*, *tool_create* changes the tool's process group to its own process number. The following sections describe the effects of this change.

### D.10.1.1. Signals from the Control Terminal

When the C-Shell (see *csh*(1)) starts a program, it changes the process group of the child to the child's process number. In addition, if that program is started in the foreground, the C-Shell also modifies the process group of the control terminal to match the child's new process group. Thus, if the tool was started from the C-Shell, the process group modification done by *tool_create* has no effect.

The Bourne Shell (see *sh*(1)) and the standard debuggers do not modify their child's process and control terminal groups. Furthermore, both the Bourne Shell and *adb*(1) are ill-prepared for the child to perform such modification. They do not propagate signals such as SIGINT to the child because they assume that the child is in the same control terminal group as they are. The bottom-line is that when a tool is executed by such a parent, typing interrupt characters at the parent process does not affect the child, and vice versa. For example, if the user types an interrupt character at *adb* while it is debugging a tool, the tool is not interrupted. Although *dbx*(1) does not modify its child's process group, it is prepared for the child to do so.

### D.10.1.2. Job Control and the C-Shell

The terminal driver and C-Shell job control interact differently with tools. First, let us examine what happens to programs using the graphics subwindow library package. When the user types an interrupt character on the control terminal, a signal is sent to the executing program. Often the signal is a SIGTSTP. The *gfxsw* library code sees this signal and tidies up by releasing any SunWindows locks that it might have and by removing the graphics from the screen before it actually suspends the program. If the program is later continued, the graphics are restored to the screen.

However, when the user types the C-Shell's *stop* command to interrupt the executing program, the C-Shell sends a SIGSTOP to the program and the *gfxsw* library code has no chance to clean up. This causes problems when the code has acquired any of the SunWindows locks, as there is no opportunity to release them. Depending on the lock timeouts, the kernel will eventually break the locks, but until then, the entire screen is unavailable to other programs and the user. To avoid this problem, the user sends the C-Shell *kill* command with the -TSTP option instead of using *stop*.

The situation for tools parallels that of the *gfxsw* code. Thus a tool that wants to interact nicely with job control must receive the signals related to job control (SIGINT, SIGQUIT, and SIGTSTP) and release any locks it has acquired. If the tool is later continued, the tool must receive a SIGCONT so that it can reacquire the locks before resuming the window operations it was executing. The tool will still be susceptible to the same problems as the *gfxsw* code when it is sent a SIGSTOP.

A final note: the user often relies on job control without realizing it; the expectation is that typing interrupt characters will halt a program. Of course, even programs that do not use

SunWindows facilities, such as a program that opens the terminal in "raw" mode, have to provide a way to terminate the program. A program using the *gfxsw* package that reads any input can provide limited job control by calling *gfxsw_inputinterrupts*.

# Index

O O O