sun microsystems

Programmer's Tutorial *to* SunWindows

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

sun microsystems

Programmer's Tutorial *to* SunWindows

Sun Microsystems, Inc. • 2550 Garcia Avenue • Mountain View, CA 94043 • 415-960-1300

Part No: 800-1182-01
Revision B of 15 May, 1985

Revision History

Version	Date	Comments
A	7 January 1984	First edition of this manual.
B-α	19 November 1984	First release of total rewrite.
B	15 April 1985	Second release of total rewrite.

Preface

This document teaches the reader how to write SunWindows tools and canvas programs.

It is intended for C programmers who want a tutorial introduction to writing programs that run in the SunWindows environment. The prerequisites: proficiency in C, and knowledge of UNIX signals and I/O primitives.

Chapter 1 introduces the windowing system and its use.

Chapter 2 teaches how to write a simple tool.

Chapter 3 introduces the panel subwindow package.

Chapter 4 discusses cursors and icons.

Chapter 5 illustrates drawing in windows, display locking, and retained subwindows.

Chapter 6 covers input handling.

Chapter 7 shows how to create pop-up menus.

Chapter 8 discusses how to handle changes in window shape and size.

Chapter 9 describes how to write a custom subwindow package.

Chapters 10 and 11 discuss how to write single-window applications called canvas programs.

Appendix A is a glossary of terms.

Appendix B is a detailed overview of the SunWindows system.

Companion Documents

Programmer's Reference Manual for SunWindows

Contents

Contents

Tables

Figures

Chapter 1

Introduction

SunWindows is an operating environment for application programs. In particular, SunWindows is a ***windowing*** environment, that is, an environment where programs run inside ***windows***. A window is an arbitrary-size rectangular portion of a display.

In SunWindows, individual windows may overlap, change size, and change position on the display. SunWindows provides a method for managing screen access and changes to windows.

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

Tool Most application programs in SunWindows are tools. A tool is a SunWindows application program that creates, owns, and manages a window and one or more subwindows. It has a mechanism for dealing with multiple windows within a single user process.

You interact with tools in order to perform various tasks, such as playing chess or editing fonts. An example of a tool you probably use every day is `shelltool`, which emulates a terminal and usually runs a UNIX shell.

Canvas program
A ***canvas*** program is characterized by its creating and managing only one window. Since a canvas program owns only one window, the mechanism required for dealing with it is simpler than for a tool.

Noninteractive utility
A noninteractive utility is typified by not creating any windows. Instead it queries and manipulates the state of one or more existing windows. Since this type of program does not own any windows, it never has to deal with window input or SIGWINCH signals. `toolplaces` is a noninteractive utility that prints a list of current window positions to standard output.

The process that is responsible for displaying a window's image and reading its input is said to ***own*** that window.

This tutorial teaches you how to build SunWindows tools and canvas programs. Chapters 2 through 9 cover tools, and chapters 10 and 11 discuss canvas programs.

Running Suntools

If you're running SunWindows and feel comfortable with it, then skip to the next chapter. If you're not running SunWindows and don't yet have a `~/.suntools` file, then use your favorite editor to create a file named `.suntools` in your home directory, containing the following text:

```
clocktool -r -S
gfxtool -Wi -C
shelltool
```

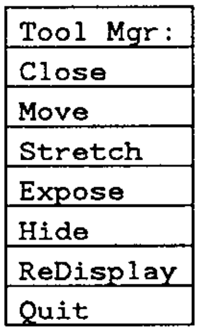
Now start up the windowing environment by typing:

```
% suntools
```

This gives you a clock in the lower left corner of the display, an open shell window in the middle of the screen and an iconic console graphics tool next to the clock. For more on `suntools`, `gfxtool`, `shelltool` and `clocktool`, please see the ***User's Manual for the Sun Workstation***.

A window can be open, or it can be closed (iconic). The normal state is open. A closed window is still running, but is quiescent. A graphical image, or ***icon***, represents a closed window on the display. A tool's icon is usually a picture showing the function performed by the tool. For example, an iconic shell tool appears graphically on the display as a picture of a video display terminal.

Now try the following. Point the mouse arrow anywhere inside the black border of the shell tool window, and press and hold the right-hand mouse button to pop up the following menu:



This is the Tool Manager menu, which represents a set of functions common to all tools.

Experiment with the options. Still holding down the right mouse button, move the arrow down the list of options, releasing the right button when you get to the choice you want.

The `Close` option turns an open window into an icon. The default icon is a black-bordered square with the tool's name inside it. Once you have an icon, press the left mouse button to get back to the open tool. The `Move` option changes the tool location, and the `Stretch` option allows you to change tool window size. Ordinarily, the window system gives you directions whenever you need to click another mouse button. Choosing `Expose` shifts the tool window in front of all other windows, whereas `Hide` places it behind all other windows. The `Redisplay` option refreshes the window in case it gets scrambled. Use `Quit` to exit the program.

To make the menu vanish without selecting any of the options, move the arrow completely out of the menu and release the button.

Now turn the shell tool window into an icon by popping up the Tool Manager menu and selecting the `Close` option. The open window should collapse into a shell tool icon on your screen. To open the shell tool, point at the terminal icon, and press the left mouse button. Your screen now displays an open shell tool, which executes C shell commands for you.

Tools also have ***accelerators*** so that users can perform commonly used actions quickly, without using the Tool Manager menu. One example of an accelerator is the quick method we just used to open the shell icon — we did not have to pop up the Tool Manager menu; we just pressed the left button. Accelerators are as follows:

- To expose an overlapped window, point at the tool's namestripe or borders, then press the left mouse button.
- To move a window, press the middle button and hold it down; an outline of the window will follow the mouse as long as you hold down the middle button. When the outline is in the desired spot, release the button to place the window.

Chapter 2

Tool Subwindows

This chapter teaches you how to write a simple tool. A tool always consists of an outer window that acts as a frame, and one or more disjoint subwindows. ***Subwindows*** are rectangular portions of a tool window, no larger than the tool itself.

A tool can read input from the keyboard and the mouse. It knows how to change window size and origin whenever you tell it to stretch or move. It can cope with overlapping windows, and can repaint portions that were once hidden behind another window, and later exposed.

First you will type in a program, and compile it using the proper libraries. While running the program, you will learn about different parts of a tool. You will edit, recompile, and run the program several times to learn about subwindow size, subwindow tiling, and the iconic form of a tool.

If you've never seen a tool up close, try running `icontool`. Nearly all of its subwindows are panels (which will be discussed in the next chapter); the exceptions are the drawing and proofing areas, which are subwindows built specifically for `icontool`. You'll certainly want to learn `dbxtool`, which facilitates program debugging. It has a panel in the middle, and user-defined subwindows above and below.

2.1. Sample Program: hello.c

In order to make things as simple as possible, we present only the bare minimum — a tool that prints "Hello world!" in a message subwindow. This program can be used as a template for more complicated tools.

Source files for the programs in this manual reside in the directory `/usr/src/sun/suntool/tutorial`. However, we encourage you to type in the programs to gain familiarity with the code being presented.

Use your favorite editor to place the following program into a file called *hello.c*:

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

struct tool *tool;
int sigwinched();

main(argc, argv)          /* print "Hello world!" in message subwindow */
        int argc;
        char *argv[];
{
        struct toolsw *msg_sw;

        if ((tool = tool_make(WIN_LABEL, argv[0], 0)) == NULL) {
                fputs("Can't make tool\n", stderr);
                exit(1);
        }
        if ((msg_sw = msgsw_createtoolsubwindow(tool, "",
            TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE,
            "Hello world!", NULL)) == NULL) {
                fputs("Can't create msgsw\n", stderr);
                exit(1);
        }
        signal(SIGWINCH, sigwinched);   /* trap window change signal */
        tool_install(tool);             /* install tool in window tree */
        tool_select(tool, 0);           /* main loop to read input */
        tool_destroy(tool);             /* after user exits, clean up */
}

sigwinched()    /* note window size change and damage repair signal */
{
        tool_sigwinch(tool);
}
```

The following header files are needed for compilation:

`<stdio.h>`
provides a standard library of input and output routines.

`<suntool/tool_hs.h>`
includes a collection of header files for dealing with tools and subwindows.

`<suntool/msgsw.h>`
declares procedures, structures, and defined constants needed for message subwindows.

Let's look at the program on a line-by-line basis. The `tool` structure needs to be global so the `sigwinched()` function, which traps the SIGWINCH (window change) signal, can pass the current tool to `tool_sigwinch()`. By contrast, the tool subwindow structure `toolsw`, used for the message subwindow, can be local to the `main()` routine.

The `tool_make()` routine creates the tool we need. Many of the tool library functions take one or more attribute/value pairs as parameters. In this case, WIN_LABEL is paired with `argv[0]`, meaning that the window label, which appears in the stripe at the top of the window, is set to the name of the program, The tool does not actually appear on the screen until the call to `tool_select()` — `tool_install()` merely places the tool window in a window data base, which keeps track of details such as position on the screen, and whether the window is hidden or exposed.

A message subwindow is created with the call to `msgsw_createtoolsubwindow()`, which must be passed the tool, the subwindow name (in this case a NULL string), the width and height of the subwindow, the string to print, and the font to print it in. TOOL_SWEXTENDTOEDGE is used for both width and height, indicating that the subwindow covers the entire tool window. Rather than supplying a special font, this program uses NULL to indicate the default font.

The call to `signal()` traps the window changing signal, SIGWINCH, and `tool_install()` installs the tool in the window tree. The window tree arbitrates the partitioning of the screen between all the windows displayed at one time. The routine `tool_select()` contains a loop where the tool spends most of its time. This loop reads keyboard and mouse events, and responds to whatever you decide it should consider.

The remainder of the code cleans up and destroys the tool when you quit. There are two ways to exit this program: by using the tool manager menu to quit, or by typing control-C from the parent window, which is the `shelltool` from where it was called.

To compile the above program, invoke `cc` directly, rather than going through `make`, so that you can see which libraries need to be loaded. SunWindows tools call routines from the following libraries:

`libsuntool.a`
provides user interface utilities, tool support, canvas program support and subwindow support.

`libsunwindow.a`
provides window device support, display access routines, and input control.

`libpixrect.a`
provides a device-independent interface to pixel operations.

Use the `-l` flag of `cc` to specify which library is to be loaded. Loading is order-dependent: high-level libraries must be given before low-level libraries, because they make use of lower-level routines:

```
% cc hello.c -o hello -lsuntool -lsunwindow -lpixrect
```

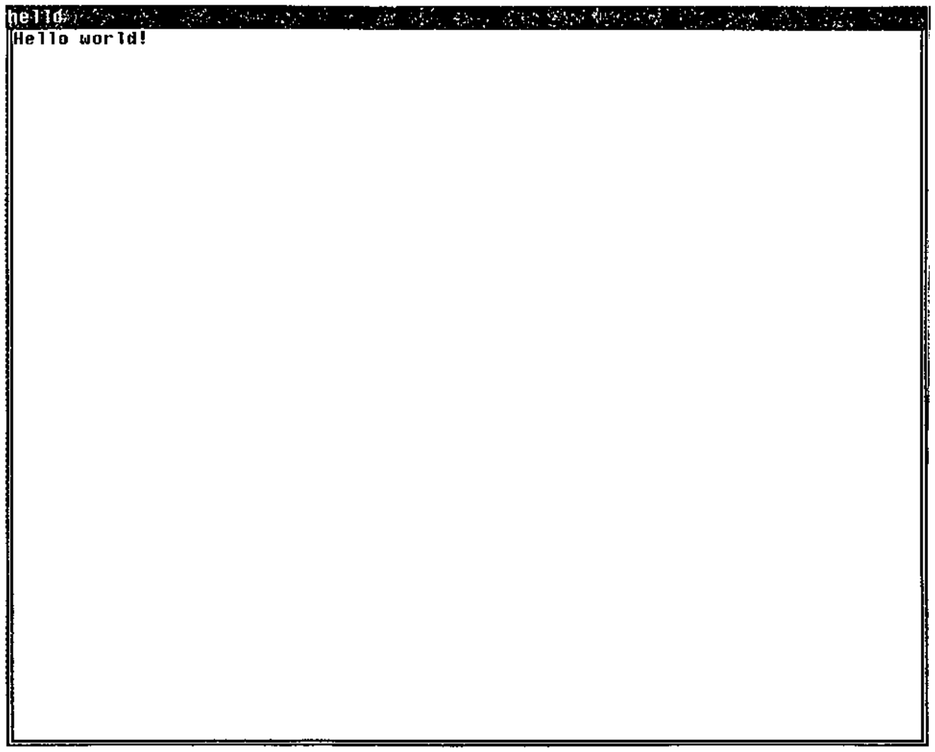
Go ahead, compile the program. These libraries must always be given in this order, or program loading won't work. If you want to debug using `dbxtool`, run `cc` with the `-g` option.

Before you run your program, make sure your workstation is running `suntools`, the window system program. This was discussed in the first chapter. Your tool program is dependent on the window system for underlying support. To run the sample program, type:

```
% hello
```

Let's examine the parts of the tool running on your screen. The black band running across the top of the tool, containing some text, is called the ***namestripe***. In the sample program, and in most programs, the namestripe contains the name of the tool. The double line going around the remainder of the tool window is the ***border***. Borders help the eye separate one tool from another. You can also use the border to bring up a menu. The entire tool window (except the namestripe and border) is hidden by the message subwindow, which contains the words "Hello world!". If the tool window changes size, the subwindow changes to remain within the tool window's borders.

Figure 2-1: Running the hello program



2.2. Message Subwindows

The above program constitutes the mininum boiler plate needed for a tool. The call to `tool_make()` creates the outer tool window, including the border and namestripe. The call to `msgsw_createtoolsubwindow()` creates the message subwindow with the text, "Hello world!". Its last argument, NULL, specifies the default font, which is used for the letters in the tool namestripe as well as the text of the message subwindow.

Try changing the sample program so that it has no subwindow. Edit the file `hello.c` again and comment out these lines with an `#ifdef` as follows:

```
#ifdef notdef
        if ((msg_sw = msgsw_createtoolsubwindow(tool, "",
            TOOL_SWEXTENDTOEDGE,
            TOOL_SWEXTENDTOEDGE,
            "Hello world!", NULL)) == NULL) {
                fputs("Can't create msgsw\n", stderr);
                exit(1);
        }
#endif
```

Now recompile (using `!cc` if you have history) and run the sample tool. You should see a blank white area inside the tool border. This is the tool window that was underneath the message subwindow in the first example, before recompilation. Be sure to edit `hello.c` again and remove the lines you just put in there.

2.3. Subwindow Size

A good way to get a feel for how to extend a tool is by code modification. Begin by looking at the function that creates the message subwindow. The argument TOOL_SWEXTENDTOEDGE is given for the width and the height of the subwindow. This extends the subwindow to the edge of the tool window. Try using 200 in place of both TOOL_SWEXTENDTOEDGE arguments. Recompile the new tool code and run it, as before. The tool window should now be visible, below and to the right of the subwindow. You may want to experiment with other values for the subwindow width and height.

2.4. Tiling Mechanism

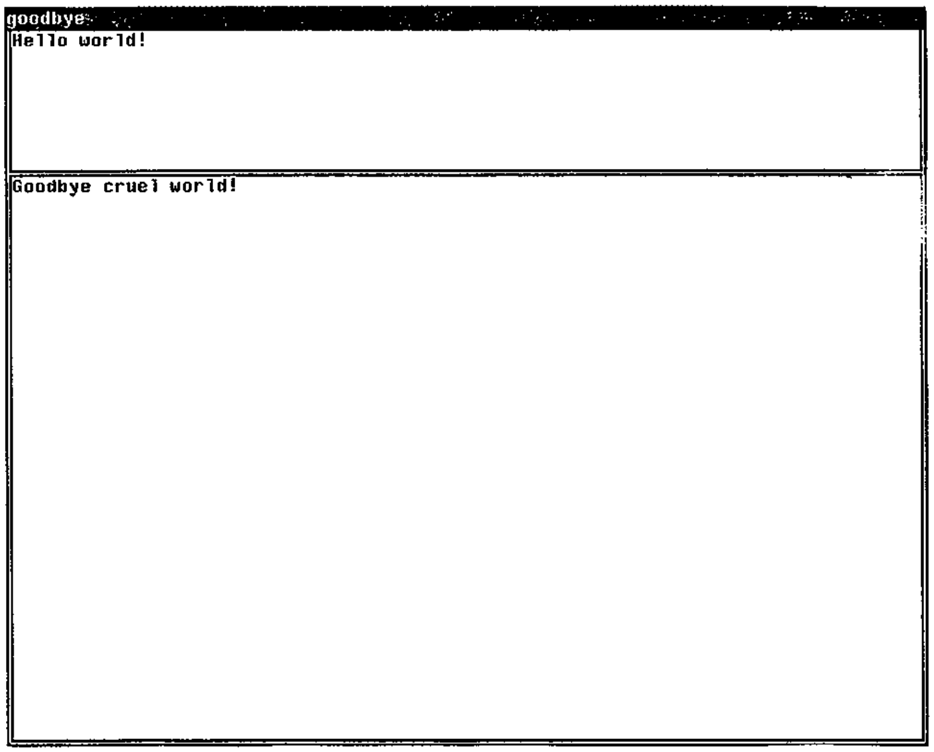
Tools provide a tiling mechanism, which enables them to display multiple subwindows without overlapping. Tool windows may overlap, but subwindows do not — they are laid down like tiles on a floor. You can observe this by creating a second subwindow, as follows:

Edit `hello.c` and copy the five lines of **`msgsw_createtoolsubwindow()`** code to a spot directly after the original code, as shown in the listing below. The first subwindow should have a width of TOOL_SWEXTENDTOEDGE
and a height of 100. Now give the second subwindow a width and height of TOOL_SWEXTENDTOEDGE. Also change the text argument in the second subwindow from "Hello World" to "Goodbye cruel world!". This helps you tell the two windows apart. Here is a listing of the revised code:

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

struct tool *tool;
int sigwinched();

main(argc, argv)        /* create two message subwindows for tiling */
        int argc;
        char *argv[];
{
        struct toolsw  *msg_sw;

        if ((tool = tool_make(WIN_LABEL, argv[0], 0)) == NULL) {
                fputs("Can't make tool\n", stderr);
                exit(1);
        }
        if ((msg_sw = msgsw_createtoolsubwindow(tool, "",
            TOOL_SWEXTENDTOEDGE, 100, "Hello world!", NULL)) == NULL) {
                fputs("Can't create msgsw\n", stderr);
                exit(1);
        }
        if ((msg_sw = msgsw_createtoolsubwindow(tool, "",
            TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE,
            "Goodbye cruel world!", NULL)) == NULL) {
                fputs("Can't create msgsw\n", stderr);
                exit(1);
        }
        signal(SIGWINCH, sigwinched);
        tool_install(tool);             /* install tool in tree of windows */
        tool_select(tool, 0);           /* main loop to read input */
        tool_destroy(tool);             /* clean up */
}

sigwinched()    /* note window size change and damage repair signal */
{
        tool_sigwinch(tool);
}
```

Now recompile the tool (using `!cc` if you have history) and rerun it.

Figure 2-2: Running the goodbye program



```
Hello world!
```

```
Goodbye cruel world!
```

Notice how there are two subwindows, one right above the other. The tiling mechanism is useful if you want to place a number of subwindows on the screen, without having to worry about the location and size of each one. Now use the mouse to shrink the entire tool window. Subwindows are resized as necessary. It is possible to modify the placement of subwindows, using advanced features discussed in the *Programmer's Reference Manual for SunWindows*.

2.5. Tools in Iconic Form

Sometimes you may prefer tools to appear initially in iconic form. Edit `hello.c`, and add the attribute value pair `WIN_ICONIC,TRUE` so that line 15 looks like this:

```
        if ((tool = tool_make(WIN_LABEL, argv[0], WIN_ICONIC, TRUE, 0)) == NULL)
```

Now recompile the program and run it. You should have a small square somewhere on your screen containing the word "hello". The default tool icon is a black-bordered square containing the name of the tool. The string paired with the WIN_LABEL attribute for `tool_make()` specifies the name of the tool. You could also specify a bitmap image for the icon, by pairing a pointer to an icon structure with the WIN_ICON attribute. We will do this in a later chapter. Consult the ***Programmer's Reference Manual for SunWindows*** for a list of all possible arguments to `tool_make()`.

Now move the mouse until the arrow points at the icon. Press the left button on the mouse. This results in an open tool exactly like the one you had before adding WIN_ICONIC to the `tool_make()` function call. Some tools, such as the clock, normally come up as icons, because they are easier to read that way. Try opening your clock by moving the mouse arrow to it, and clicking the left button.

Chapter 3

Panels

This chapter introduces ***panels***. The panel subwindow package is most commonly used to build control panels, which allow the user to graphically issue commands to a tool and to set the tools' options. The user moves the mouse so that the arrow points at an option, and presses the left mouse button to select that option. For an example of a tool with a complex control panel, try running `icontool`.[1]

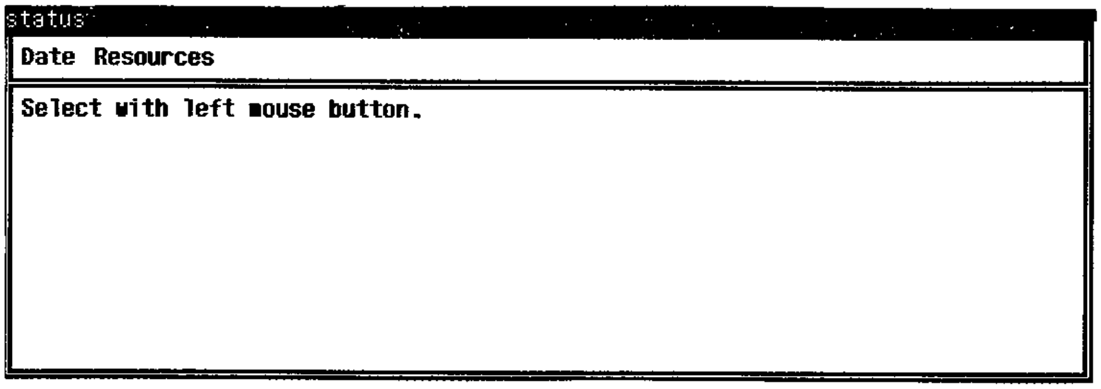
Panels, like the message subwindows introduced in the previous chapter, are a special predefined type of subwindow that an application program can use to perform useful functions. You can define your own type of subwindow if the standard subwindow packages don't provide what you need (see Chapter 9).

First you will key in a sample program, and compile it using the **make** facility. Then you will be asked to add another option to the program.

3.1. Sample Program: status.c

This simple program displays two panels, as shown in the picture below:

Figure 3-1: Running status.c



```
status
Date  Resources
Select with left mouse button.
```

[1] Also see Figure 8-1 in the *Sunwindows Reference Manual*.

The upper panel is the control panel; the lower panel is for output. The control panel contains two buttons. Selecting the "Date" button with the mouse causes the current date and time to be displayed in the output panel. Selecting the "Resources" button causes the tool's CPU resource utilization information to be displayed in the output panel.

Before proceeding with the program, let's set up the **make** definitions. The source file is called ***status.c*** and we need to include the same libraries as before. Use your favorite editor to place the following lines in a file named ***makefile***. This may seem like a lot of work, but it saves time in the long run:

```
LIBS = -lsuntool -lsunwindow -lpixrect

status: status.o
        cc status.o -o status $(LIBS)
```

Now it's time to enter the C code. Carefully remaining in the same directory, place the following program into a file called ***status.c*** (it might be helpful to recycle some code from the last program):

```
#include <stdio.h>
#include <suntool/tool_hs.h>
#include <suntool/panel.h>
#include <sys/resource.h>

struct tool     *tool;
struct toolsw   *control_panel_sw, *output_panel_sw;
Panel           control_panel, output_panel;
Panel_item      output_item, date_item, rusage_item;
int             sigwinched(), date_proc(), rusage_proc();

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

        if ((tool = tool_make(WIN_LABEL, argv[0], 0)) == NULL) {
                fputs("Can't make tool\n", stderr);
                exit(1);
        }

        /* setup control panel */
        if ((control_panel_sw = panel_create(tool, 0)) == NULL) {
                fputs("Can't create control_panel\n", stderr);
                exit(1);
        }
        control_panel = control_panel_sw->ts_data;
        date_item =   panel_create_item(control_panel, PANEL_BUTTON,
            PANEL_LABEL_STRING, "Date",
            PANEL_NOTIFY_PROC,  date_proc,
            0);
        rusage_item = panel_create_item(control_panel, PANEL_BUTTON,
            PANEL_LABEL_STRING, "Resources",
            PANEL_NOTIFY_PROC,  rusage_proc,
            0);
        panel_fit_height(control_panel);
        /* setup output panel */
        if ((output_panel_sw = panel_create(tool, 0)) == NULL) {
                fputs("Can't create output_panel\n", stderr);
                exit(1);
        }
        output_panel = output_panel_sw->ts_data;
        output_item = panel_create_item(output_panel, PANEL_MESSAGE,
            PANEL_LABEL_STRING, "Select with left mouse button.", 0);

        signal(SIGWINCH, sigwinched);
        tool_install(tool);     /* install tool in window tree */
        tool_select(tool, 0);   /* main loop to read input */
        tool_destroy(tool);     /* clean up tool */
}

sigwinched()    /* note window size change and damage repair signal */
{
        tool_sigwinch(tool);
}

date_proc(item, event)  /* get and display date and time */
        Panel_item item;
        struct inputevent *event;
{
        long clock;
        char *ctime();

        time(&clock);
        panel_set(output_item, PANEL_LABEL_STRING, ctime(&clock), 0);
}

rusage_proc(item, event)        /* get and display resource usage */
        Panel_item item;
        struct inputevent *event;
{
        struct rusage rusage;
        static char buf[80];

        getrusage(RUSAGE_SELF, &rusage);
        sprintf(buf, "User %D secs %D millisecs; System %D secs %D millisecs",
            rusage.ru_utime.tv_sec, rusage.ru_utime.tv_usec/1000,
            rusage.ru_stime.tv_sec, rusage.ru_stime.tv_usec/1000);
        panel_set(output_item, PANEL_LABEL_STRING, buf, 0);
}
```

First we create the tool by calling `tool_make()`. Next we set up the two panels. Setting up a panel involves several steps. In order to manipulate a panel you must have its ***handle***, which is a variable of type `Panel`. To obtain a `Panel`, first create the panel subwindow with `panel_create()`, then use the `ts_data` field of that subwindow. Once you have the panel's handle in hand, you can populate the panel with items by calling `panel_create_item()`.

In `status.c`, the steps outlined above are followed first for the control panel, then for the output panel. Note that after creating the control panel and its items, there is a call to `panel_fit_height()`. This sets the panel's height to fit the items which have been created so far within the panel. There is no such call following the creation of the output panel, so the output panel extends all the way to the bottom and right edges of the tool.

`panel_create_item()` (and most other panel routines) take as arguments variable-length, null-terminated ***attribute lists***. For example, the call to create the control panel's `date_item` specifies that the label appearing on the screen will be the string "Date", and the procedure to be called when the user selects the item with the mouse is `date_proc()`. (For details on attributes or other aspects of panels, consult Chapter 8 of the ***Sunwindows Reference Manual***).

`date_proc()` simply gets the current time, and then calls `panel_set()` to change the label of `output_item` to that time formatted as a string.

Now that you have a `makefile` and a source file, you simply type **make** to compile and link the program:

```
% make
cc -c status.c
cc status.o -o status -lsuntool -lsunwindow -lpixrect
% status
```

The system responds with the names of the file being compiled, and the modules and libraries being loaded. When program building is completed, run the program by typing its name, `status`. When the tool window comes up, move the mouse arrow and click the left button for each of the two choices. When you're done, press the right button over the tool's namestripe, bringing up the tool manager menu, and `Quit` the tool.

3.2. Adding Panel Items

As an exercise, add a panel button that allows the user to quit. Hint: you need to have a `quit_proc()` function defined, which looks something like this:

```
quit_proc(ip, new_value)
        Panel_item          ip;
        int                 new_value;
        struct inputevent *event;
{
        /* announce to user that we're quitting */
        panel_set(output_item, PANEL_LABEL_STRING, "Bye!", 0);
        /* terminate tool's execution */
        tool_done(tool);
}
```

You also need to declare the global variable `quit_item` (of type `Panel_item`) and the integer function `quit_proc()`. Then you need a few lines to create a panel item for quitting, which would call the quit procedure above.

A hint on debugging programs using panels: if your program compiles correctly but core dumps when you execute it, check to see if all of the attribute lists in panel routine calls are properly formed. In particular, the list must be terminated with a zero. Also, remember that `panel_create()` takes one mandatory argument (the tool) followed by an attribute list, while `panel_create_item()` takes two mandatory arguments (the panel and the type of the item) followed by an attribute list.

Chapter 4

Pixrects

This chapter explains how to define and use *pixrects*, which are rectangles composed of pixels. Cursors and icons are two objects that use pixrects. A cursor is a 16 by 16 bit pattern that tracks the position of the mouse. An icon is a larger bit pattern (often 64 by 64) that represents a closed window. Both cursors and icons are manipulated using routines from the `pixrect` library, which provides a device-independent interface to pixel operations on Sun workstations. Among other things, the library provides routines for displaying text, drawing vectors, and for painting and transferring pixel rectangles. To the programmer using this library, all pixrects are described in the same way, and manipulated by the same operations.

A particular pixrect may refer to an entire display, or to small images such as a character in a font, or a cursor image. The array containing the pixels may be visible on the display, or it may be stored in memory or on disk. Peculiarities of specific devices are hidden below the pixrect interface. Some devices provide hardware support for common operations, while others require all operations to be performed in software.

First you will run a program that changes the default cursor. Then you will modify the program so it saves the old cursor, makes a new cursor, and changes back again to the original cursor. Finally, you will learn to use icons, and modify the `status` tool from the last chapter so it has a fancy icon when closed.

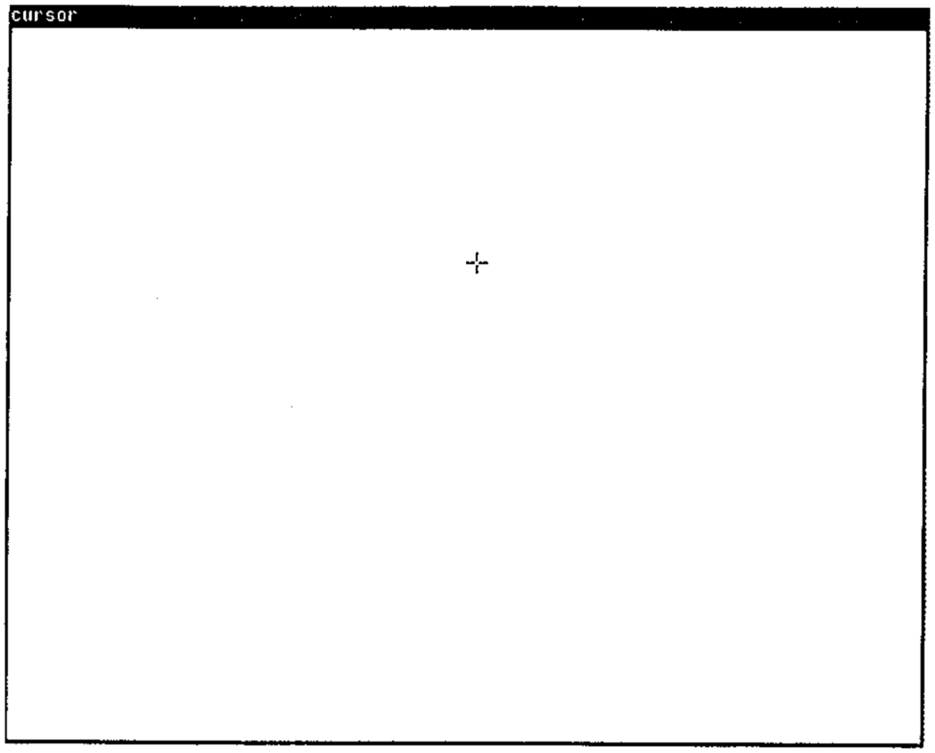
4.1. Changing the Cursor

The following program creates a tool window, then changes the mouse arrow into a crosshair. The crosshair was created with `icontool`, one of the utilities on a standard Sun system. The `icontool` program wrote out a bitmap in hex codes, which gets included into the source code below. What looks like an illegal external function call to `DEFINE_CURSOR()` is actually a macro that creates a cursor structure named `crosshair`. The `DEFINE_CURSOR()` macro takes care of declaring some substructures of the cursor, one of which is a pixrect to hold the cursor image.

Enter the following program into a file called *cursor.c*:

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

struct tool *tool;
int sigwinched();

DEFINE_CURSOR(crosshair, 7, 7, PIX_SRC | PIX_DST,
        0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0000, 0xFC7E,
        0x0000, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0000);

main(argc, argv)        /* create window with crosshair cursor */
        int argc;
        char *argv[];
{
        if ((tool = tool_make(WIN_LABEL, argv[0], 0)) == NULL) {
                fputs("Can't make tool\n", stderr);
                exit(1);
        }
        signal(SIGWINCH, sigwinched);
        tool_install(tool);                              /* in window tree */
        win_setcursor(tool->tl_windowfd, &crosshair);    /* change cursor */
        tool_select(tool, 0);                            /* loop for input */
        tool_destroy(tool);                              /* tool clean up */
}

sigwinched()    /* note window size change and damage repair signal */
{
        tool_sigwinch(tool);
}
```

Figure 4-1: Running the cursor program



Most everything here is familiar, except the call to **win_setcursor()**, which changes the mouse's arrow cursor into the crosshair. You need to supply the tool window file descriptor as the first argument to **win_setcursor()**, and the address of the crosshair cursor structure as the second argument. Think of window file descriptors as a low-level way of referring to a window. Note that no attempt is made to save the old cursor; the next example shows how to save and restore the old cursor.

Try moving the mouse crosshair onto the namestripe. Notice how the crosshair disappears — the cursor is the same color as the namestripe. A ***raster-op*** is a method for combining one bitmap with another. The raster-op specified in the `DEFINE_CURSOR()` macro above is a logical OR of the cursor bitmap and the destination window. Change the raster-op so that it is an exclusive OR of the cursor and destination:

```
            PIX_SRC^PIX_DST,
```

Now recompile the program. Run it and note how the cursor appears when it moves over the namestripe (it's now the opposite of its background). Here is a table of some useful raster-ops:

Table 4-1: Some useful raster-ops

Raster-op	**Effect**
`PIX_SRC`	overwrite destination with source
`PIX_SRC | PIX_DST`	paint destination with source bitmap
`PIX_SRC ^ PIX_DST`	inverted paint of destination with source
`PIX_SRC & PIX_DST`	mask destination with source bitmap
`PIX_NOT(PIX_SRC) & PIX_DST`	mask destination with inverted source
`PIX_NOT(PIX_DST)`	invert destination area

Painting and masking have special meanings here: painting is ORing while masking is ANDing. The shape of the bitmap is more likely to be preserved in painting than in masking. Since the cursor and namestripe are both black, and ORing one with one gives one, the cursor disappears when it's over the namestripe. By contrast, exclusive ORing one with one gives zero, so the cursor still appears when on the namestripe. If you don't understand raster-ops, experiment by modifying the program to use other raster-ops in the table.

For black and white backgrounds, `PIX_SRC^PIX_DST` is preferable, but for white and grey backgrounds, `PIX_SRC|PIX_DST` is the best choice.

In addition to these raster-ops, there are two operations to set and clear pixels. The first is `PIX_SET`, which turns a pixel on (black), and the second is `PIX_CLR`, which turns a pixel off (white). By definition, black is one and white is zero.

4.2. Restoring the Old Cursor

Often you want to change back to the old cursor after something happens. The following program sets an alarm to go off in five seconds. The resulting SIGALRM calls the `changecursor()` routine to restore the old cursor. This alarm mechanism might be used by software that asks you to confirm an operation that would irreversibly affect a file. If no confirmation arrives within five seconds, the operation would not be performed, and the software would return to its normal state. One program that dynamically changes the cursor is `chesstool`, which shows an hourglass when you have to wait for the computer to make its next move.

Cursors are defined on a subwindow-by-subwindow basis. The `win_setcursor()` routine should be passed the file descriptor of the window where you want the cursor changed.

Copy the file *cursor.c* into a file named *restore_cursor.c*, which you can edit to look like the following program. Most of the lines in this program are the same as ones in the last program:

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

struct tool *tool;
int sigwinched(), changecursor();

DEFINE_CURSOR(crosshair, 7, 7, PIX_SRC | PIX_DST,
        0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0000, 0xFC7E,
        0x0000, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0100, 0x0000);
DEFINE_CURSOR(oldcursor, 0, 0, 0,
        0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0);

main(argc, argv)        /* create window with 5 second crosshair cursor */
        int argc;
        char *argv[];
{
        if ((tool = tool_make(WIN_LABEL, argv[0], 0)) == NULL) {
                fputs("Can't make tool\n", stderr);
                exit(1);
        }
        signal(SIGWINCH, sigwinched);
        tool_install(tool);                             /* in window tree */
        win_getcursor(tool->tl_windowfd, &oldcursor);   /* save cursor */
        win_setcursor(tool->tl_windowfd, &crosshair);   /* change cursor */
        signal(SIGALRM, changecursor);                  /* signal routine */
        alarm(5);                                       /* alarm in 5 seconds */
        tool_select(tool, 0);                           /* loop for input */
        tool_destroy(tool);                             /* tool clean up */
}

changecursor()          /* change cursor back to original image */
{
        win_setcursor(tool->tl_windowfd, &oldcursor);
}

sigwinched()    /* note window size change and damage repair signal */
{
        tool_sigwinch(tool);
}
```

This program shows the necessary steps for saving the old cursor. Setting the cursor is simple — all you need is one call to `win_setcursor()`. But saving the old cursor is more prone to error. Calling `win_getcursor()` is not enough — one must allocate space for the cursor structure and its substructures. The macro DEFINE_CURSOR does this for you. Finally, after the five second alarm goes off, the UNIX signal mechanism calls the `changecursor()` function, which sets the cursor back to what it was originally.

4.3. Changing the Tool Icon

When the `status` program in the last chapter goes iconic, its icon isn't very fancy — just the program's name in an outline square. A sophisticated tool should have its own icon, so that looking at the icon tells you what the tool does. This isn't hard to accomplish: first you must use the `icontool` program to create an icon and store it in some file (see *icontool*(1)); then, in your program, declare an array that contains the data in the `icontool` output file (see below), and call `DEFINE_ICON_FROM_IMAGE()`, supplying the desired name of the `icon` structure and the name of the array.

```
unsigned short  icon_image[256] = {
#include <images/status.icon>
};
DEFINE_ICON_FROM_IMAGE(icon, icon_image);
```

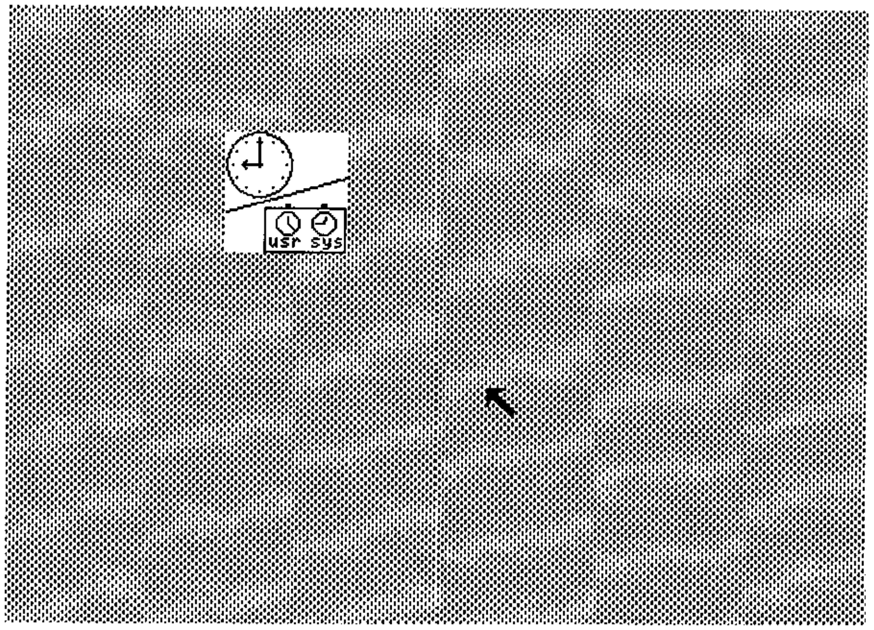
When you create the tool with `tool_make()`, you must specify the address of the `icon` structure as a value paired with the `WIN_ICON` keyword.

```
if ((tool = tool_make(WIN_LABEL, argv[0], WIN_ICON, &icon, 0)) == NULL) {
        fputs("Can't make tool\n", stderr);
        exit(1);
}
```

Make these changes to the `status` program. Now compile the program by typing **`make status`** to the shell. Only the ***status.o*** module compiles, and then the program will be loaded from the module and the three libraries.

Now run the program by typing `status`. The tool comes up in open form. Move the mouse arrow to the namestripe, click the right button down, and release it inside the `Close` option. The tool goes iconic and appears at the edge of your screen. Now move the mouse arrow to the icon, and click the left button to open the tool. The program works just as it did before, but now it has an icon of its own.

Figure 4-2: The status program's icon



Chapter 5

Pixwins

A pixrect is a low-level object that defines a rectangle composed of pixels. A ***pixwin***, on the other hand, is a higher-level object that encapsulates locking and clipping information needed to support multiple overlapping windows. Pixwin routines are contained in the `sunwindow` library.

In order to learn about pixwin library calls, you will modify the `status` program from a previous chapter, employing pixwin functions to do fancy graphics. You will move other windows over top of your tool, thus demonstrating clipping and locking facilities. The output panel will be replaced by a graphics subwindow in which a clock image and some inverted text will be displayed.

5.1. Sample Program: status_image.c

Let's return to the `status` program presented in the previous chapter. We will modify that program so it uses pixwin routines to place pixrects on the screen. In particular, it puts up a clock, rather than merely printing out the time. In order to accomplish this, it's best to use a graphics subwindow, which provides a single window (a canvas) on which to write graphical output. Before going any further, however, you need to change your `makefile` to look like this; all you need to add are the two lines to specify how to make `status_image`:

```
LIBS = -lsuntool -lsunwindow -lpixrect

status_image: status_image.o
        cc status_image.o -o status_image $(LIBS)
status: status.o
        cc status.o -o status $(LIBS)
```

Now, you have to modify the program so it includes definitions for a clock image. You need the clock outline, available on-line as a suntools icon image, and the clock hands, available on-line in the suntools source directory. The clock outline is declared as a static memory pixrect using the `mpr_static` macro call.

Copy the file ***status.c*** to a file named ***status_image.c***, and edit the new file to look like this:

```
#include <stdio.h>
#include <suntool/tool_hs.h>
#include <suntool/panel.h>
#include <suntool/gfxsw.h>
#include <sys/resource.h>

struct tool       *tool;
struct toolsw     *control_panel_sw, *gfx_sw;
struct gfxsubwindow *gfx;
Panel             control_panel;
Panel_item        date_item, rusage_item;
```

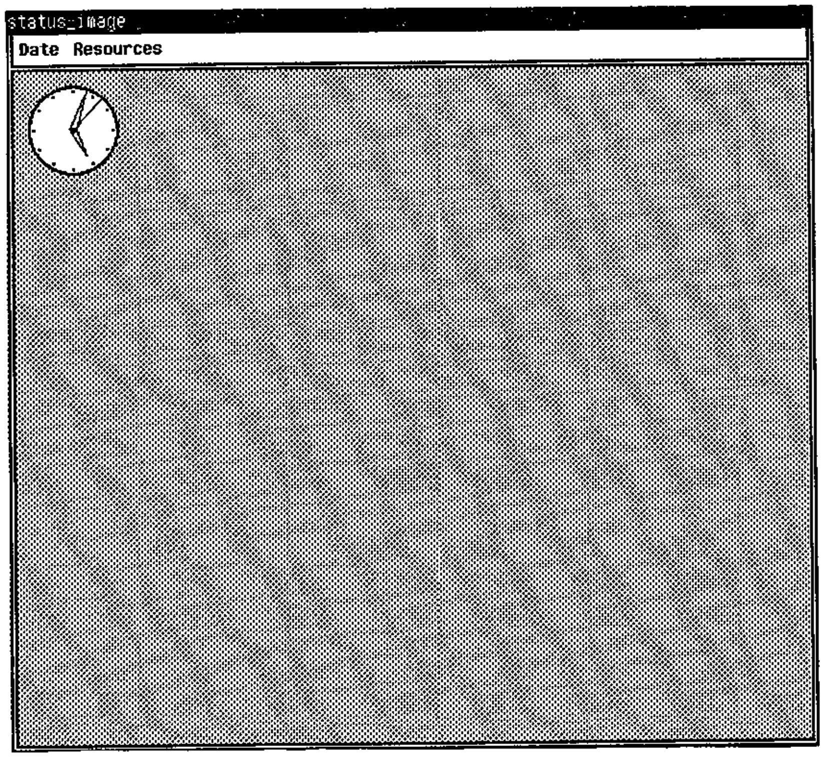
```
int             sigwinched(), date_proc(), rusage_proc();

unsigned short  icon_image[256] = {
#include <images/status.icon>
};
DEFINE_ICON_FROM_IMAGE(icon, icon_image);

unsigned short clock_image[256] = {     /* clock outline */
#include <images/clocktool.icon>
};
mpr_static(clock_pr, 64, 64, 1, clock_image);
#include "/usr/src/sun/suntool/clockhands.h"    /* Define struct hand */
#include "/usr/src/sun/suntool/clockhands.c"    /* Table of hand positions */

main(argc, argv)        /* panel subwindow for date or resource usage */
        int argc;
        char *argv[];
{

        if ((tool = tool_make(WIN_LABEL,argv[0], WIN_ICON,&icon, 0)) == NULL) {
                fputs("Can't make tool\n", stderr);
                exit(1);
        }

        /* setup control panel */
        if ((control_panel_sw = panel_create(tool, 0)) == NULL) {
                fputs("Can't create control_panel\n", stderr);
                exit(1);
        }
        control_panel = control_panel_sw->ts_data;
        date_item =   panel_create_item(control_panel, PANEL_BUTTON,
            PANEL_LABEL_STRING, "Date",
            PANEL_NOTIFY_PROC,  date_proc,
            0);
        rusage_item = panel_create_item(control_panel, PANEL_BUTTON,
            PANEL_LABEL_STRING, "Resources",
            PANEL_NOTIFY_PROC,  rusage_proc,
            0);
        panel_fit_height(control_panel);

        /* setup graphics subwindow */
        if ((gfx_sw = gfxsw_createtoolsubwindow(tool, "",
            TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE, NULL)) == NULL) {
                fputs("Can't create graphics subwindow\n", stderr);
                exit(1);
        }
        gfx = (struct gfxsubwindow *) gfx_sw->ts_data;
        gfxsw_getretained(gfx);

        signal(SIGWINCH, sigwinched);
        tool_install(tool);     /* install tool in window tree */
        tool_select(tool, 0);   /* main loop to read input */
        tool_destroy(tool);     /* clean up tool */
}
```

```
sigwinched()    /* note window size change and damage repair signal */
{
        tool_sigwinch(tool);
}

date_proc(item, event)  /* put clock in graphics subwindow */
        Panel_item item;
        struct inputevent *event;
{
#define DATE_X_OFFSET 10
#define DATE_Y_OFFSET 10
        long clock;
        struct tm *local;
        struct hands *hand;

        time(&clock);
        local = localtime(&clock);              /* get time of day */
        /* Initialize the graphics subwindow to grey */
        pw_replrop(gfx->gfx_pixwin, 0, 0, gfx->gfx_rect.r_width,
            gfx->gfx_rect.r_height, PIX_SRC, tool_bkgrd, 0, 0);
        /*  write clock outline */
        pw_write(gfx->gfx_pixwin, DATE_X_OFFSET, DATE_Y_OFFSET,
            clock_pr.pr_width, clock_pr.pr_height, PIX_SRC, &clock_pr, 0, 0);
        /* write hour hand */
        hand = &hand_points[(local->tm_hour*5 + (local->tm_min + 6)/12) % 60];
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->x1, DATE_Y_OFFSET + hand->y1,
            DATE_X_OFFSET + hand->hour_x, DATE_Y_OFFSET + hand->hour_y,
            PIX_SET, 0);
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->x2, DATE_Y_OFFSET + hand->y2,
            DATE_X_OFFSET + hand->hour_x, DATE_Y_OFFSET + hand->hour_y,
            PIX_SET, 0);
        /* write minute hand */
        hand = &hand_points[local->tm_min];
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->x1, DATE_Y_OFFSET + hand->y1,
            DATE_X_OFFSET + hand->min_x, DATE_Y_OFFSET + hand->min_y,
            PIX_SET, 0);
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->x2, DATE_Y_OFFSET + hand->y2,
            DATE_X_OFFSET + hand->min_x, DATE_Y_OFFSET + hand->min_y,
            PIX_SET, 0);
        /* write second hand */
        hand = &hand_points[local->tm_sec];
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->sec_x, DATE_Y_OFFSET + hand->sec_y,
            DATE_X_OFFSET + hand->min_x, DATE_Y_OFFSET + hand->min_y,
            PIX_SET, 0);
}

rusage_proc(item, event) /* put resource usage in graphics subwindow */
        Panel_item item;
        struct inputevent *event;
```

```
{
#define RUSAGE_X_OFFSET 10
#define RUSAGE_Y_OFFSET 20
        struct rusage rusage;
        static char buf[80];

        getrusage(RUSAGE_SELF, &rusage);
        sprintf(buf, "User %D secs %D millisecs; System %D secs %D millisecs",
            rusage.ru_utime.tv_sec, rusage.ru_utime.tv_usec/1000,
            rusage.ru_stime.tv_sec, rusage.ru_stime.tv_usec/1000);
        /* clear screen */
        pw_writebackground(gfx->gfx_pixwin, 0, 0,
            gfx->gfx_rect.r_width, gfx->gfx_rect.r_height, PIX_CLR);
        /* write out time resource usage string in reverse video */
        pw_text(gfx->gfx_pixwin, RUSAGE_X_OFFSET, RUSAGE_Y_OFFSET,
            PIX_NOT(PIX_SRC), NULL, buf);
}
```

Much of this program should be familiar, since much of it is recycled from the first **status** program. The **date_proc()** and **rusage_proc()** functions, however, are quite different.

Figure 5-1: Running the status_image program



First look at **rusage_proc**. The field **gfx_rect** contains the dimensions of the graphics subwindow, which is then cleared by passing the origins and dimensions of the subwindow, along with the operation PIX_CLR, to **pw_writebackground()**.

Now look at `date_proc()`. After getting the time, it floods the screen with a grey pattern using `pw_replrop()`. The clock outline is placed in the subwindow with `pw_write()`, which must be given the origins, dimensions, operation PIX_SRC, and address of the clock pixrect. The final two arguments specify the origin in the source pixrect, usually zero. The hands are placed on the clock with `pw_vector()`. Both the hour and the minute hands are elongated triangles, so they have two origins near the center of the clock: `x1,y1` and `x2,y2`. The hour hand is shorter then the minute hand, so its destination `hour_x,hour_y` is closer to the center of the clock than the minute hand's destination `min_x,min_y`. The second hand is much simpler, since it has only the origin `sec_x,sec_y`, and uses the same destination as the minute hand does.

5.2. Explicit Locking

Every time a pixwin routines accesses the screen, it needs to get the exclusive right to do so. It acquires the ***display lock***, accesses the screen and then releases the display lock. Display lock access is a relatively expensive operation. So, if you can do it only once for a series of pixwin calls then you can save time by reducing graphics system overhead.

You are now going to optimize the part of your program that displays the clock image. Add the lines:

```
/* Do explicit display locking (for efficiency) */
pw_lock(gfx->gfx_pixwin, &gfx->gfx_rect);
```

right after the call to `localtime` in `date_proc`. Also, add the lines:

```
/* Release display lock */
pw_unlock(gfx->gfx_pixwin);
```

at the end of the routine `date_proc`. Now, each of the pixwin calls between `pw_lock` and `pw_unlock` wouldn't have to gain display access rights and should run quicker. Unfortunately, you may not notice any difference in the speed because there are not enough pixwin operations to show the improvement. However, other applications can be sped up significantly by using explicit display locking.

5.3. Retained Subwindows

We are using a retained graphics subwindow in `status_image`. With a ***retained subwindow***, the system keeps an image of the window in memory. If the window is damaged (for example, covered and then exposed), the image in memory can be copied back in order to repair the window. Only a single call is necessary to retain a graphics subwindow:

```
gfxsw_getretained(gfxsw);
```

The graphics subwindow is the only pre-packaged type that can be retained.

There is more detail on damage and how to repair it in a later chapter.

Chapter 6

Input Handlers

The 4.2 BSD system call `select()` is used for synchronous I/O multiplexing. It waits for something specific to happen without consuming any resources. You tell it what to wait for. Traditionally, UNIX I/O has been synchronous — the system blocks during a read or write. By multiplexing synchronous I/O, it is possible to have multiple entities waiting for events simultaneously. Originally, `select()` was invented for networking, but it is also useful for interactive graphics. In a window system, for example, each subwindow is a separate I/O entity.

In this chapter you will learn how to read input from the mouse by writing your own `select` handler. Each subwindow gets notified of input events, such as mouse or keyboard input. The program explicitly states which events it is interested in receiving. Events ignored by a subwindow go to the tool window. If you want certain mouse buttons to have a special effect in a certain subwindow, you have to enable the events you want to consider for that subwindow.

We will work on a program that allows you to sketch with the mouse. As rectangles appear in a large sketchpad window, pixels appear in a small proofing window. For this program, we use retained graphics subwindows because we want repainting to be done for us automatically.

6.1. Sample Program: sketch.c

Before you proceed, modify your `makefile` so it contains these two lines above the other dependencies:

```
sketch: sketch.o
        cc sketch.o -o sketch $(LIBS)
```

The idea behind this tool is to have two subwindows: a large window to act as the sketchpad, and a small window beside it containing a proofing image of what's being sketched. Since the sketchpad and proof will be 40 x 40, we need to make sure the tool window is the right size. The call to `tool_make()` has to look something like the following:

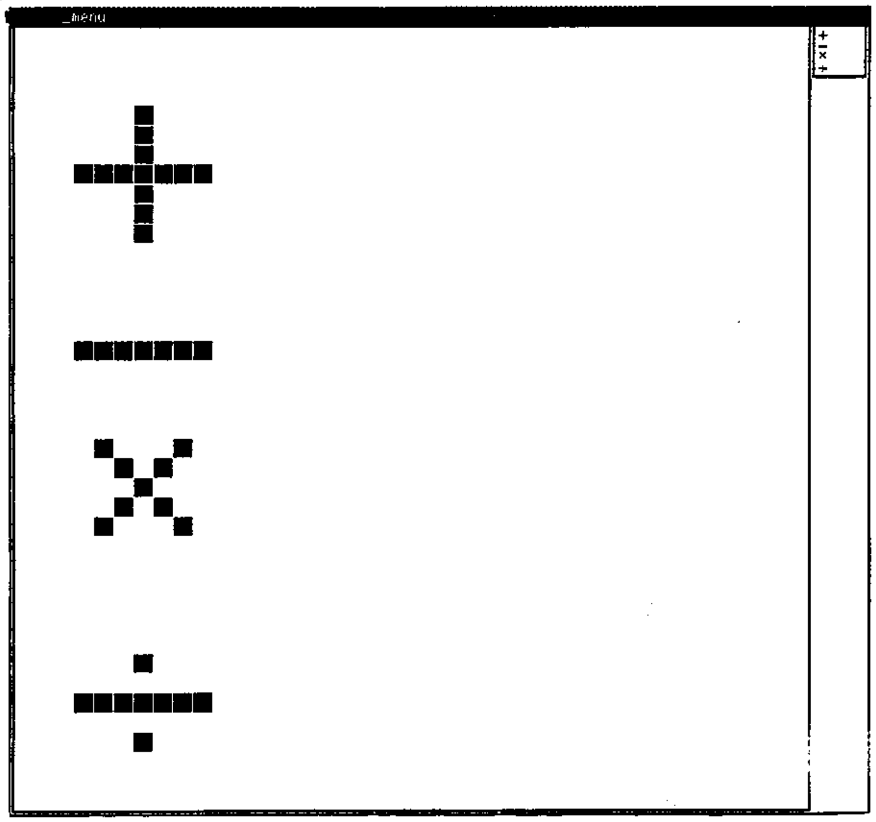
```
tool = tool_make(
        WIN_LABEL,      argv[0],
        WIN_TOP,        100,
        WIN_LEFT,       100,
        WIN_WIDTH,      wsiz + psiz + (TOOL_BORDERWIDTH * 3),
        WIN_HEIGHT,     wsiz + (font->pf_defaultsize.y +2) + TOOL_BORDERWIDTH,
        0);
```

The window height is (40 * 16) pixels, plus the height of the namestripe (in this case, this is the height of the font plus 2), plus the size of the bottom border. The window width is (40 * 16) pixels, plus 40 for the proof window, plus 15 pixels for three borders. The sketchpad subwindows is (40 * 16) pixels square, while the proof subwindow is 40 pixels square. In order to make sure the entire tool window fits on the screen, we position it 100 pixels down from the top, and 100 pixels from the left edge of the display.

Now enter this program into a file called *sketch.c*:

```
#include <stdio.h>
#include <suntool/tool_hs.h>
#include <suntool/gfxsw.h>

struct tool             *tool;
struct toolsw           *sgsw, *pgsw;
struct gfxsubwindow     *sketch, *proof;
int                     sigwinched(), do_select();

int sqsiz = 16;
int psiz = 40;

main(argc, argv)        /* tool with sketchpad and pixel image */
        int argc;
        char *argv[];
{
        struct inputmask mask;
        struct pixfont *font;
        int wsiz;

        wsiz = sqsiz * psiz;
        font = pw_pfsysopen();
        if ((tool = tool_make(
            WIN_LABEL,  argv[0],
            WIN_TOP,    100,
            WIN_LEFT,   100,
            WIN_WIDTH,  wsiz + psiz + (TOOL_BORDERWIDTH * 3),
            WIN_HEIGHT, wsiz + (font->pf_defaultsize.y + 2) + TOOL_BORDERWIDTH,
            0)) == NULL) {
                fputs("Can't make tool\n", stderr);
                exit(1);
        }

        /* setup sketch subwindow (including select routine) */
        if ((sgsw = gfxsw_createtoolsubwindow(tool,"",wsiz,wsiz,0)) == NULL) {
                fputs("Can't create sketch graphics subwindow\n", stderr);
                exit(1);
        }
        sketch = (struct gfxsubwindow *)sgsw->ts_data;
        gfxsw_getretained(sketch);
        sgsw->ts_io.tio_selected = do_select;

        /* setup mouse buttons for sketch subwindow (can't OR them together) */
        input_imnull(&mask);
        win_setinputcodebit(&mask, MS_LEFT);
        win_setinputcodebit(&mask, MS_MIDDLE);
        win_setinputcodebit(&mask, LOC_MOVEWHILEBUTDOWN);
        win_setinputmask(sketch->gfx_windowfd, &mask,
            (struct inputmask *)NULL, WIN_NULLLINK);

        /* setup proof subwindow */
        if ((pgsw = gfxsw_createtoolsubwindow(tool, "", psiz,psiz,0)) == NULL)
                fputs("Can't create proof graphics subwindow\n", stderr);
                exit(1);
        }
        proof = (struct gfxsubwindow *)pgsw->ts_data;
        gfxsw_getretained(proof);

        /* normal boilerplate */
        signal(SIGWINCH, sigwinched);
        tool_install(tool);                             /* in window tree */
        tool_select(tool, 0);                           /* loop for input */
        tool_destroy(tool);                             /* tool clean up */
}

sigwinched()    /* note window size change and damage repair signal */
{
        tool_sigwinch(tool);
}

do_select(sw, ibits, obits, ebits, timer)       /* respond to user input */
        caddr_t sw;
        int *ibits, *obits, *ebits;
        struct timeval **timer;
{
        struct inputevent ie;
        static drawval;
        int x, y;

        input_readevent(sketch->gfx_windowfd, &ie);
        /* if button up, else button down */
        if (win_inputnegevent(&ie))
                goto done;
        else if (ie.ie_code == MS_LEFT)
                drawval = PIX_SET;
        else if (ie.ie_code == MS_MIDDLE)
                drawval = PIX_CLR;
        /* paint rectangle and dot */
        x = ie.ie_locx - (ie.ie_locx % sqsiz);
        y = ie.ie_locy - (ie.ie_locy % sqsiz);
        pw_writebackground(sketch->gfx_pixwin, x, y, sqsiz-1, sqsiz-1, drawval);
        pw_put(proof->gfx_pixwin, x/sqsiz, y/sqsiz, drawval==PIX_SET ? 1 : 0);
done:
        *ibits = *obits = *ebits = 0;
}
```

Figure 6-1: Running the sketch program



In the first setup section, the graphics subwindow pointers for each of the tool subwindows are set to point at `ts_data`, and are coerced into pointers to objects of the proper type. Both graphics subwindows are retained so they can repair themselves if they become damaged (damage will be discussed in a later chapter).

The proper mouse buttons have to be enabled. First, `input_imnull()` is called to initialize the input mask so that all input codes are disabled. Then, the left, middle, and button-down-move code bits are turned on in the input code mask. Finally, the input mask is set so that these input events, and no others, are recognized.

Nothing happens in the sketchpad subwindow until the first mouse button is pushed. At that time, the `tool_select()` routine calls `do_select()`, which reads the input event and deals with it appropriately. This routine is called once for each input event.

Input events are read with the `input_readevent()` routine. Pressing the left button yields a black square (PIX_SET), while pressing the middle button yields a white square (PIX_CLR). The input event location `ie_locx,ie_locy` is rounded down to the nearest multiple of `sqsiz`. A call to `pw_writebackground()` draws the 16 x 16 square in the sketchpad window, whereas a call to `pw_put()` draws the single pixel in the proof window.

Chapter 7

Pop-Up Menus

In this chapter you will learn how to create menus which allow users to select one of several options listed. The SunWindows package provides library routines that make it easy to create a ***pop-up menu***, which appears on the screen when you push the right-hand mouse button, and disappears when you release the mouse button. Pop-up menus have the advantage that they take up screen space only when invoked. In contrast, for example, a panel remains on the display the entire time the tool runs.

We will continue working on the `sketch` program, adding an option to clear the window, and another to blacken the entire window. These options will be choices on a pop-up menu. Because the pop-up menu will apply only inside the first graphics subwindow, users will still be able to access the tool manager menu from the namestripe.

7.1. Sample Program: sketch.c

Edit *sketch.c* and add these lines to the global declarations at the top of the program:

```
#include <suntool/menu.h>
#define CLEAR_BUTTON     (caddr_t)'c'
#define BLACKEN_BUTTON   (caddr_t)'b'

struct menuitem menu_items[] = {
        { MENU_IMAGESTRING, "clear",   CLEAR_BUTTON },
        { MENU_IMAGESTRING, "blacken", BLACKEN_BUTTON }
};
struct menu menu_body = {
        MENU_IMAGESTRING, "Commands",
        sizeof(menu_items) / sizeof(struct menuitem),
        menu_items, NULL, NULL
};
struct menu *menu_ptr = &menu_body;
```

Two strings, `clear` and `blacken`, appear as options in the pop-up menu, which is labeled `Commands`. When the `clear` or `blacken` options are chosen, the `menu_display()` library routine returns a pointer to the menu item containing `c` or `b` respectively.

The right mouse button is normally used for popping up the menu. In order to make use of it, we need to enable it inside the tool subwindow. This line has to be added beneath the other calls to `win_setinputcodebit()`:

```
        win_setinputcodebit(&mask, MS_RIGHT);
```

This routine must be called once for every event being enabled (the event flags cannot be ORed together). The `do_select()` routine has to be modified so it does something when the right mouse button is pushed. All that's needed is another `else if` statement and a `goto` label to circumvent the normal drawing of rectangles and pixels. Here's what the routine looks like after these modifications:

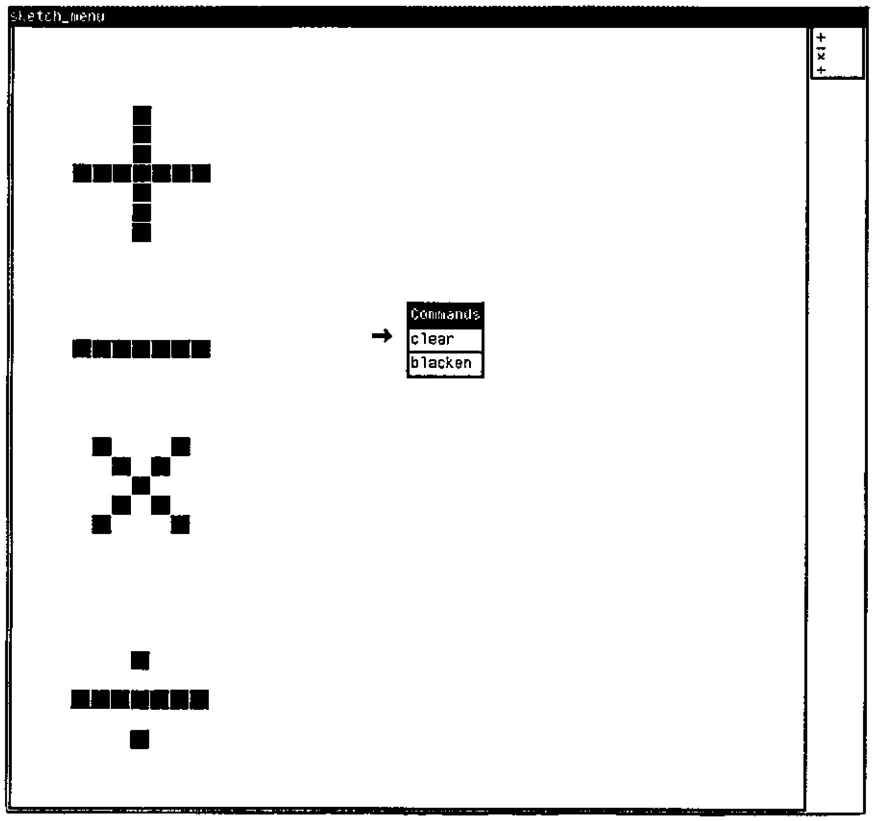
```
do_select(sw, ibits, obits, ebits, timer)      /* respond to user input */
        caddr_t sw;
        int *ibits, *obits, *ebits;
        struct timeval **timer;
{
        struct inputevent ie;
        static drawval;
        int x, y;

        input_readevent(sketch->gfx_windowfd, &ie);
        /* if button up, else button down */
        if (win_inputnegevent(&ie))
                goto done;
        else if (ie.ie_code == MS_LEFT)
                drawval = PIX_SET;
        else if (ie.ie_code == MS_MIDDLE)
                drawval = PIX_CLR;
        else if (ie.ie_code == MS_RIGHT) {
                do_menu(&ie);
                goto done;
        }
        /* paint rectangle and dot */
        x = ie.ie_locx - (ie.ie_locx % sqsiz);
        y = ie.ie_locy - (ie.ie_locy % sqsiz);
        pw_writebackground(sketch->gfx_pixwin, x, y, sqsiz-1, sqsiz-1, drawval);
        pw_put(proof->gfx_pixwin, x/sqsiz, y/sqsiz, drawval==PIX_SET ? 1 : 0);
done:
        *ibits = *obits = *ebits = 0;
}
```

The only thing left is to add the `do_menu()` routine, which performs actions requested from the pop-up options menu. Add this routine at the end of the program:

```
do_menu(ie)             /* perform requests issued from pop-up menu */
        struct inputevent *ie;
{
        struct menuitem *mi;
        struct rect r;

        if (mi = menu_display(&menu_ptr, ie, sketch->gfx_windowfd)) {
                win_getsize(sketch->gfx_windowfd, &r);
                if (mi->mi_data == CLEAR_BUTTON) {
                        pw_writebackground(sketch->gfx_pixwin,
                           0, 0, r.r_width, r.r_height, PIX_CLR);
                        pw_writebackground(proof->gfx_pixwin,
                           0, 0, psiz, psiz, PIX_CLR);
                } else if (mi->mi_data == BLACKEN_BUTTON) {
                        pw_writebackground(sketch->gfx_pixwin,
                           0, 0, r.r_width, r.r_height, PIX_SET);
                        pw_writebackground(proof->gfx_pixwin,
                           0, 0, psiz, psiz, PIX_SET);
                }
        }
}
```

Now compile the program by typing **make**, and try out the options on the new pop-up menu.

Figure 7-1: Running the sketch_menu program



7.2. Adding Options to the Menu

As an exercise, add an option to `invert` the sketchpad and bitmap window. This should be easy enough if you modify the **menuitem** structure. Hint: you can use `PIX_NOT(PIX_DST)` as an argument to `pw_writebackground()` to invert a pixwin.

Chapter 8

Window Reshaping

In this chapter you will learn how to handle changes in window shape and size. You have to watch out for the SIGWINCH signal, which informs you that something has changed. Panels and message subwindows will be updated automatically whenever necessary, but a graphics subwindow needs attention when its size changes.

Windows may be stretched or shrunk. Execute the `status_image` program again. Now move the mouse arrow to the namestripe, and press the right button to get the pop-up menu. Pick the `Stretch` option, and use the left button to extend the program's window out to the right. When it redraws, note how the newly exposed part of the graphics subwindow is garbled. In such a case, the subwindow has been ***damaged*** and lacks the proper repair.

While you're at it, use the `Stretch` option to shrink the window until it's smaller than the clock or the process times. Notice how these things are clipped if their image is too big for the window. (This is not damage, so we don't need to worry about it.) Stretch the tool window back to its original size, and notice how the clock or process times reappears.

8.1. SIGWINCH Signal

A SIGWINCH is an asynchronous signal that notifies the window owner process that its image is incorrect, either due to a size change or for some other reason. This signal is transmitted in four different situations:

(1) When window size changes after stretching or shrinking.

(2) When part of a window that was covered up becomes uncovered.

(3) When a tool changes from iconic to open. (this is actually an example of (2)).

(4) When a window is first created (this is actually an example of (1)).

A SIGWINCH signal is ***not*** transmitted when a window is moved or dragged (unless it becomes uncovered in the process), or when a window becomes partially obscured. In those cases, tools don't have to take special actions.

There are three ways to handle damage: with retained subwindows, redrawing through a clipping list, or selective redrawing from a clipping list. We have already demonstrated the first method in the `sketch` example. The second method will be covered by an example in the next chapter. The third method, used internally by the `ttysubwindow`, is too complicated for this tutorial.

8.2. SIGWINCH Handlers

The proper place to repair damage is in a SIGWINCH handler routine. In the example below, you supply a SIGWINCH handler for the graphics subwindow:

```
     gfx_sw->ts_io.tio_handlesigwinch = gfx_sigwinch;
```

Note that a SIGWINCH handler is different from the SIGWINCH signal catching routine `sigwinched`. The signal catcher is set up via the call to `signal`. `tool_select()` calls the SIGWINCH handler some time after `sigwinched` is called by the system.

There are two steps a SIGWINCH handler must take to handle a size change: rescale the contents of the window, and then repaint the image. Some applications assume a window of fixed size and so don't need to rescale the window contents. To determine whether a window's size has changed, compare the window size with the size the last time a SIGWINCH arrived.

8.3. Retained Subwindows

Although a retained subwindow relieves you of worrying about simple damage, it does not relieve you of concerns regarding size changes. Here we show you how to deal with size changes in a retained graphics subwindow.

Copy the program `status_image.c` into a file named `status_size.c`. Update your make file to be able to build `status_size`. Edit your program to look like the following:

```
     #include <stdio.h>
     #include <suntool/tool_hs.h>
     #include <suntool/panel.h>
     #include <suntool/gfxsw.h>
     #include <sys/resource.h>

     struct tool      *tool;
     struct toolsw    *control_panel_sw, *gfx_sw;
     struct gfxsubwindow *gfx;
     Panel            control_panel;
     Panel_item       date_item, rusage_item, latest_command;
     int              sigwinched(), date_proc(), rusage_proc(), gfx_sigwinch();

     unsigned short  icon_image[256] = {
     #include <images/status.icon>
     };
     DEFINE_ICON_FROM_IMAGE(icon, icon_image);

     unsigned short clock_image[256] = {     /* clock outline */
     #include <images/clocktool.icon>
     };
     mpr_static(clock_pr, 64, 64, 1, clock_image);
     #include "/usr/src/sun/suntool/clockhands.h"    /* Define struct hand */
     #include "/usr/src/sun/suntool/clockhands.c"    /* Table of hand positions */

     main(argc, argv)        /* panel subwindow for date or resource usage */
             int argc;
             char *argv[];
     {

             if ((tool = tool_make(WIN_LABEL,argv[0], WIN_ICON,&icon, 0)) == NULL) {
                     fputs("Can't make tool\n", stderr);
                     exit(1);
```

```
        }

        /* setup control panel */
        if ((control_panel_sw = panel_create(tool, 0)) == NULL) {
                fputs("Can't create control_panel\n", stderr);
                exit(1);
        }
        control_panel = control_panel_sw->ts_data;
        date_item =   panel_create_item(control_panel, PANEL_BUTTON,
            PANEL_LABEL_STRING, "Date",
            PANEL_NOTIFY_PROC,  date_proc,
            0);
        rusage_item = panel_create_item(control_panel, PANEL_BUTTON,
            PANEL_LABEL_STRING, "Resources",
            PANEL_NOTIFY_PROC,  rusage_proc,
            0);
        panel_fit_height(control_panel);

        /* setup graphics subwindow */
        if ((gfx_sw = gfxsw_createtoolsubwindow(tool, "",
            TOOL_SWEXTENDTOEDGE, TOOL_SWEXTENDTOEDGE, NULL)) == NULL) {
                fputs("Can't create graphics subwindow\n", stderr);
                exit(1);
        }
        gfx = (struct gfxsubwindow *) gfx_sw->ts_data;
        gfxsw_getretained(gfx);
        gfx_sw->ts_io.tio_handlesigwinch = gfx_sigwinch;

        signal(SIGWINCH, sigwinched);
        tool_install(tool);     /* install tool in window tree */
        tool_select(tool, 0);   /* main loop to read input */
        tool_destroy(tool);     /* clean up tool */
}

sigwinched()    /* note window size change and damage repair signal */
{
        tool_sigwinch(tool);
}

date_proc(item, event)  /* put clock in graphics subwindow */
        Panel_item item;
        struct inputevent *event;
{
#define DATE_X_OFFSET 10
#define DATE_Y_OFFSET 10
        long clock;
        struct tm *local;
        struct hands *hand;

        time(&clock);
        local = localtime(&clock);              /* get time of day */
        /* Initialize the graphics subwindow to grey */
        pw_replrop(gfx->gfx_pixwin, 0, 0, gfx->gfx_rect.r_width,
            gfx->gfx_rect.r_height, PIX_SRC, tool_bkgrd, 0, 0);
        /*  write clock outline */
        pw_write(gfx->gfx_pixwin, DATE_X_OFFSET, DATE_Y_OFFSET,
            clock_pr.pr_width, clock_pr.pr_height, PIX_SRC, &clock_pr, 0, 0);
        /* write hour hand */
        hand = &hand_points[(local->tm_hour*5 + (local->tm_min + 6)/12) % 60];
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->x1, DATE_Y_OFFSET + hand->y1,
            DATE_X_OFFSET + hand->hour_x, DATE_Y_OFFSET + hand->hour_y,
            PIX_SET, 0);
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->x2, DATE_Y_OFFSET + hand->y2,
            DATE_X_OFFSET + hand->hour_x, DATE_Y_OFFSET + hand->hour_y,
            PIX_SET, 0);
        /* write minute hand */
        hand = &hand_points[local->tm_min];
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->x1, DATE_Y_OFFSET + hand->y1,
            DATE_X_OFFSET + hand->min_x, DATE_Y_OFFSET + hand->min_y,
            PIX_SET, 0);
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->x2, DATE_Y_OFFSET + hand->y2,
            DATE_X_OFFSET + hand->min_x, DATE_Y_OFFSET + hand->min_y,
            PIX_SET, 0);
        /* write second hand */
        hand = &hand_points[local->tm_sec];
        pw_vector(gfx->gfx_pixwin,
            DATE_X_OFFSET + hand->sec_x, DATE_Y_OFFSET + hand->sec_y,
            DATE_X_OFFSET + hand->min_x, DATE_Y_OFFSET + hand->min_y,
            PIX_SET, 0);
        latest_command = item;
}

rusage_proc(item, event) /* put resource usage in graphics subwindow */
        Panel_item item;
        struct inputevent *event;
{
#define RUSAGE_X_OFFSET 10
#define RUSAGE_Y_OFFSET 20
        struct rusage rusage;
        static char buf[80];

        getrusage(RUSAGE_SELF, &rusage);
        sprintf(buf, "User %D secs %D millisecs; System %D secs %D millisecs",
            rusage.ru_utime.tv_sec, rusage.ru_utime.tv_usec/1000,
            rusage.ru_stime.tv_sec, rusage.ru_stime.tv_usec/1000);
        /* clear screen */
        pw_writebackground(gfx->gfx_pixwin, 0, 0,
            gfx->gfx_rect.r_width, gfx->gfx_rect.r_height, PIX_CLR);
        /* write out time resource usage string in reverse video */
        pw_text(gfx->gfx_pixwin, RUSAGE_X_OFFSET, RUSAGE_Y_OFFSET, PIX_NOT(PIX_SRC)
            NULL, buf);
        latest_command = item;
}
```

```
gfx_sigwinch(sw)
        caddr_t sw;
{
        /* Let graphics subwindow notice that sigwinched */
        gfxsw_interpretesigwinch(gfx);
        /* Let graphics subwindow update retained pixwin */
        gfxsw_handlesigwinch(gfx);
        /* See if need to redraw the window due to size change */
        if (gfx->gfx_flags & GFX_RESTART) {
                gfx->gfx_flags &= ~GFX_RESTART;
                if (latest_command == date_item)
                        date_proc(date_item, NULL);
                else if (latest_command == rusage_item)
                        rusage_proc(rusage_item, NULL);
                /* else already clear */
        }
}
```

Turn your attention to `gfx_sigwinch()`. We make a few housekeeping calls to `gfxsw_interpretesigwinch()` and `gfxsw_handlesigwinch()` so that the graphics subwindow package can notice that a SIGWINCH has arrived and also take care of simple damage situations with its retained image. However, if the size has changed, then the flag GFX_RESTART is set and you need to redraw the graphics subwindow based on the state information contained in `latest_command`.

Chapter 9

User-defined Subwindows

In this chapter you will learn how to create user-defined subwindows. There are times when a pre-packaged subwindow just doesn't do what you need. For example, a message subwindow doesn't provide any control over placement of text within the subwindow. If your application wanted to vertically center the text in the subwindow then the message subwindow package couldn't be used.

Whenever you employ a user-defined subwindow, you need to write your own SIGWINCH handler. Remember, a SIGWINCH signal is sent to a tool when the window becomes open, when window size changes, or when part of the window becomes uncovered. In the first two instances, the entire tool needs to be redrawn. In the third instance, only the uncovered portion needs redrawing.

You will write and run a modified version of the program ***hello.c*** that you wrote earlier. The new program will employ your own (user-defined) subwindow in which to draw text instead of using a message subwindow. The text will be vertically centered in the subwindow instead of at the top of the subwindow.

9.1. Sample Program: hello_own.c

Before you proceed, modify your `makefile` so it contains these two lines above the other dependencies:

```
hello_own: hello_own.o
        cc hello_own.o -o hello_own $(LIBS)
```

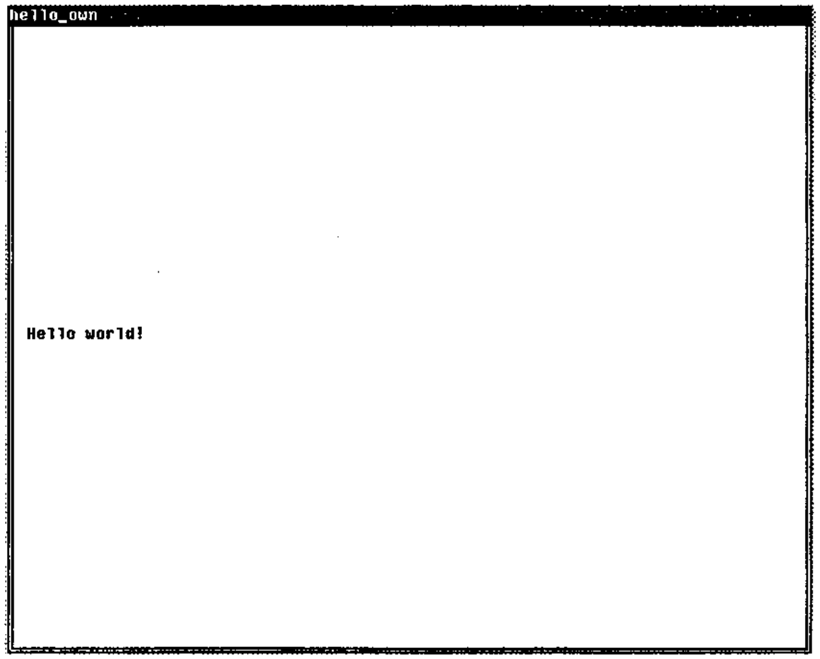
You are replacing the message subwindow in ***hello.c*** with a user-defined subwindow, which is created with `tool_createsubwindow()`. With these do-it-yourself subwindows, you have to explicitly open the pixwin using `pw_open`. In addition, you must write your own SIGWINCH signal handler.

Now enter these lines into a file called ***hello_own.c*** (you may want to start with ***hello.c***):

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

struct tool *tool;
struct toolsw *my_sw;
struct pixwin *my_pixwin;
struct rect my_rect;
int sigwinched(), my_sigwinch();

main(argc, argv)        /* print "Hello world!" in user defined subwindow */
        int argc;
        char *argv[];
```

```
{
        if ((tool = tool_make(WIN_LABEL, argv[0], 0)) == NULL) {
                fputs("Can't make tool\n", stderr);
                exit(1);
        }

        /* Create a vanilla subwindow */
        if ((my_sw = tool_createsubwindow(tool, "", TOOL_SWEXTENDTOEDGE,
            TOOL_SWEXTENDTOEDGE, "Hello world!", NULL)) == NULL) {
                fputs("Can't create subwindow\n", stderr);
                exit(1);
        }

        /* Open a pixwin with which to draw in the subwindow */
        if ((my_pixwin = pw_open(my_sw->ts_windowfd)) == NULL) {
                fputs("Can't open pixwin\n", stderr);
                exit(1);
        }

        /* Remember subwindow size so that we can notice size changes */
        win_getsize(my_sw->ts_windowfd, &my_rect);

        /* Register a SIGWINCH handler by setting up a function to call */
        my_sw->ts_io.tio_handlesigwinch = my_sigwinch;

        /* Normal boilerplate */
        signal(SIGWINCH, sigwinched);   /* trap window change signal */
        tool_install(tool);             /* install tool in window tree */
        tool_select(tool, 0);           /* main loop to read input */
        tool_destroy(tool);             /* after user exits, clean up */
}

sigwinched()    /* note window size change and damage repair signal */
{
        tool_sigwinch(tool);
}

my_sigwinch(sw) /* deal with subwindow size change and damage repair */
        caddr_t sw;
{
        struct rect nrect;

        /* Determine current size of subwindow */
        win_getsize(my_sw->ts_windowfd, &nrect);
        /* Prepare pixwin for damage repair */
        pw_damaged(my_pixwin);
        /* If the size has changed */
        if (my_rect.r_width != nrect.r_width ||
            my_rect.r_height != nrect.r_height)
                /* Set pixwin clipping to be all visible portion of subwindow */
                pw_donedamaged(my_pixwin);
        /* Clear pixwin */
        pw_writebackground(my_pixwin, 0, 0, nrect.r_width, nrect.r_height,
            PIX_CLR);
        /* Draw text, roughly vertically centered */
        pw_text(my_pixwin, 10, nrect.r_height/2, PIX_SRC, NULL, "Hello world!");
        /* Make sure call pw_donedamaged if haven't above */
        if (my_rect.r_width == nrect.r_width &&
            my_rect.r_height == nrect.r_height)
                pw_donedamaged(my_pixwin);
        else
                /* Remember new subwindow size */
                my_rect = nrect;
}
```

Figure 9-1: Running the hello_own program



9.2. Using the Clipping List

Much of the time, only part of a window is damaged, for example, when a corner of another window covers it up. As discussed above, the damaged window receives a SIGWINCH when it becomes uncovered. At this time, the program should call **pw_damaged()** to set the clipping list for its window. This routine determines which area has been damaged, and clips off the rest of the window. When the window gets refreshed, only the damaged part gets repaired. That is, **pw_write()**, which normally writes the whole window, will now write only the damaged part. Your program can then proceed to repair the entire window, without having to worry about what part was damaged. Finally, the program should call **pw_donedamaged()** to inform the system that you've repaired the damage.

The problem with the above method is that when the window size changes, you'll want to redraw the entire window. The solution is to get the size of the window with `win_getsize()`, and see if the window is the same size as it was before receiving the SIGWINCH. If the size is different after calling `pw_damaged()`, you must call `pw_donedamaged()` ***before*** redrawing the window.

Note that if you use this method to repair damage, `pw_damaged()` and `pw_donedamaged()` must be called ***every time*** a SIGWINCH arrives.

9.3. Signal Handlers

The `my_sigwinch()` routine is somewhat tricky. In order to repair damage correctly, we need to set a clipping list with `pw_damaged()`. If the window was reshaped, the whole thing needs redrawing so we do a `pw_donedamaged()` so that the pixwin refers to the entire subwindow. We record the width and height, so we will know if window size will have changed the next time a SIGWINCH comes.

As each pixwin routine is executed, it affects only the screen under the clipping list established by the latest call to either `pw_damaged()`or `pw_donedamaged()`. If only a portion of the window were damaged, only that part would be repaired. But if the whole window were damaged, the entire window would be redrawn.

Finally, `pw_donedamaged()` is called, if not called earlier, to restore the clipping list. Calls to further pixwin routines will now affect the visible portion of the window.

Although tempting, it is critical to not stray from the pattern of calls to `pw_damaged()`and `pw_donedamaged()` as given in `my_sigwinch()`. Annoying behavior can occur by trying to do things differently, for example, excessive repainting or show-through color (on a color monitor).

Signal handlers are provided automatically with pre-packaged subwindow types such as panels, message subwindows, and graphics subwindows. When you need a user-defined subwindow, however, you are responsible for writing a custom signal handler.

Chapter 10

Writing a Simple Canvas Program

This chapter describes a simple ***canvas program***, called `canvasflash`, that runs in SunWindows. A canvas program is one that owns a single window.

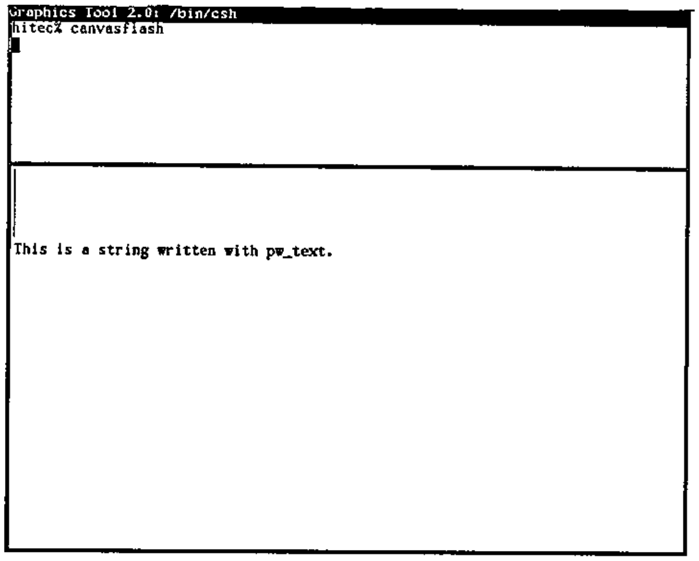
There are several canvas programs you can run from the graphics tool (`gfxtool`), or from the shell tool (`shelltool`):

- `/usr/demo/bouncedemo`
- `/usr/demo/framedemo`
- `/usr/demo/jumpdemo`
- `/usr/demo/spheresdemo`

The source for these programs resides in `/usr/src/sun/suntool`, so you can examine their source code as well.

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

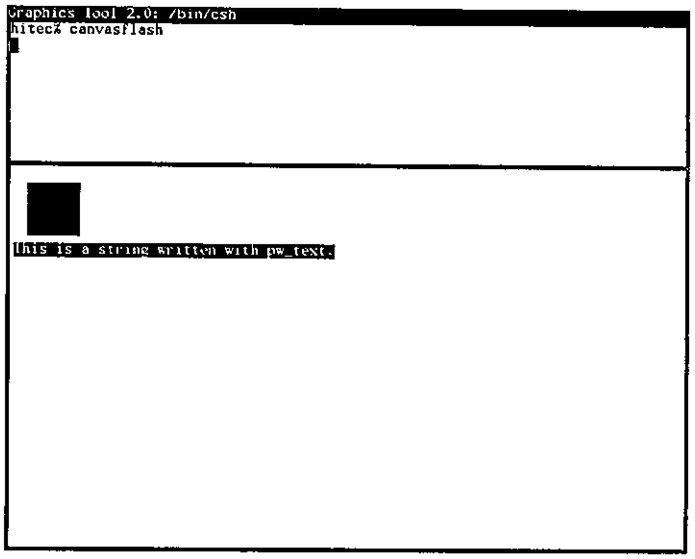
`canvasflash` creates a window and draws a vertical line, a white square and a string of text within the window. Every second it inverts the images within the window. Inverting the line causes it to appear white and essentially "disappear"; the text appears in reverse video; and the white square appears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



```
Graphics Tool 2.0: /bin/csh
hitec% canvasflash
```

```
This is a string written with pw_text.
```

Figure 10-1: `canvasflash` Output



```
Graphics Tool 2.0: /bin/csh
hitec% canvasflash
```

```
This is a string written with pw_text.
```

Figure 10-2: Inverted `canvasflash` Output

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

```
canvasflash -n number
```

The source code for this example is provided in the file `/usr/src/sun/suntool/tutorial/canvasflash.c`. To compile it, use:

```
cc -o canvasflash canvasflash.c -lsuntool -lsunwindow -lpixrect
```

10.1. The *canvasflash* Code

The flow of `canvasflash` is as follows:

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

Here is a listing of `/usr/src/sun/suntool/tutorial/canvasflash.c`. You may want to glance at it now; however, it is primarily for reference as you read the subsequent explanation. Extensive comments have been removed in favor of the accompanying text.

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

main(argc, argv)
        int argc;
        char **argv;
{
        int op;
        struct gfxsubwindow *gfx;


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

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

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

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

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

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

                /* sample pw_* calls */
```

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

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

10.2. External Declarations

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

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

It also allows the program to use the `printf` buffered output routines.

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

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

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

10.3. Initialization

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

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

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

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

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

`gfx_flags`
: is an **int** of window housekeeping flags. These flags, as well as `gfx_rect`, are updated

asynchronously by the graphics subwindow manager when something happens to affect the window's size or visibility.

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

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

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

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

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

10.4. Display Loop

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

`pw_writebackground()`
: is used to draw a solid square on the display.

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

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

```
sleep(1);
```

10.5. Cleanup

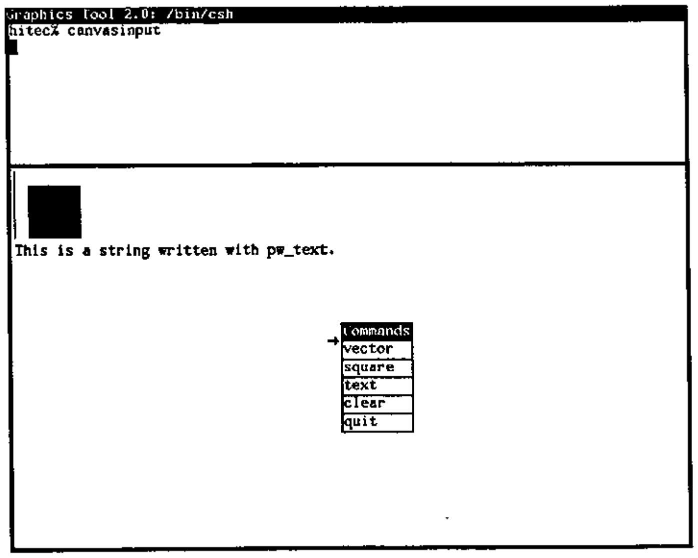
To cleanup before exiting the program, `gfxsw_done()` frees the resources allocated for the graphics subwindow.

Chapter 11

Writing a More Sophisticated Canvas Program

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

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



```
Graphics Tool 2.0: /bin/csh
hitec% canvasinput
```

```
This is a string written with pw_text.
```

Figure 11-1: `canvasinput` Output

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

```
cc -o canvasinput canvasinput.c -lsuntool -lsunwindow -lpixrect
```

11.1. The *canvasinput* Code

The flow of `canvasinput` is as follows:

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

Here is a listing of `/usr/src/sun/suntool/tutorial/canvasinput.c`. You may want to glance at it now; however, it is primarily for reference as you read the subsequent explanation. Extensive comments have been removed in favor of the accompanying text.

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

extern  struct menuitem *menu_display();

static  struct gfxsubwindow *gfx;

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

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

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

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

		/* Cleanup */
		gfxsw_done(gfx);
	}

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

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

```
                                gfxsw_inputinterrupts(gfx, &ie);
                        }
                }
                *ibits = *obits = *ebits = 0;
        }
```

11.2. External Declarations

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

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

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

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

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

11.3. Defining the Menu

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

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

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

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

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

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

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

11.4. Initialization

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

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

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

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

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

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

Note: Don't confuse the calling sequence of `gfxsw_setinputmask` with the lower level `win_setinputmask`. `win_setinputmask` is called for windows in general. `gfxsw_setinputmask` is called for graphics subwindows in canvas programs.

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

```
gfxsw_getretained(gfx);
```

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

11.5. Notification Manager

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

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

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

`gfxsw_select` loops indefinitely and can be terminated by a call to `gfxsw_selectdone`, which is described below.

11.6. Handling Notifications

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

The arms of the `switch` statement on the input event are similar to the code described in the chapter about the `canvasflash` program. The default arm of the `switch` statement is used to look for and act on some common interrupt character sequences (such as, ^C, DEL, ^D, ^Z, and control-shift-backslash) used with terminal-based programs:

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

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

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

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

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

11.7. Termination and Cleanup

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

Appendix A

Glossary

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

border
The thin double stripe along the left, bottom, and right side of a tool window.

canvas program
A SunWindows program that owns a single window, and often uses a graphics subwindow.

clocktool
A simple tool that continually updates a circular clock face with the time of day.

cursor
A small image that moves around the screen in response to mouse motions, indicating the position of the mouse.

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

gfxtool
A graphics tool that provides display space for programs that write to the current window.

handle
A pointer to an object.

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

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

manager
The software which creates and manipulates an object.

menu
A displayed list of related items for user choice.

namestripe
The wide stripe across the top of a tool window, generally containing the name of the tool.

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

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

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

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

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

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

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

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

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

rect
A structure that defines a rectangle.

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

repair
Regenerating the image for a part of a window that has just become visible (and thus is damaged).

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

shelltool
A tool window that acts as a 34-line 80-column terminal.

SIGWINCH
A signal to a process that a window has changed.

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

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

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

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

subwindow handle
A pointer to a subwindow object.

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

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

suntools
The package that takes over the workstation screen, enabling you to run shelltool, gfxtool, and clocktool.

SunWindows
The Sun window system.

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

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

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

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

user
A person using the system, as opposed to a programmer.

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

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

[2] † UNIX is a trademark of Bell Laboratories.

Appendix B

SunWindows Implementation Overview

This appendix investigates SunWindows from the programmer's point of view. In particular, it describes the major components of the SunWindows implementation and their interactions.

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

B.1. Architectural Principles

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

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

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

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

Information Hiding
: Many of the major packages of SunWindows are implemented with the concepts of ***data abstraction*** in mind. A data abstraction is a collection of subroutines and private data structures — only the subroutines access the data structures, and the interface to the data abstraction is defined entirely in terms of the interface that the subroutines provide. This practice is encouraged to provide flexibility of implementation for both SunWindows itself

and for programmers writing new tools using SunWindows facilities. For example, SunWindows' lowest level provides device-independent access to bit-mapped devices in a framework where new devices can be added with no impact on any existing code using this level. Additionally, by hiding the implementation information, SunWindows minimizes the impact on existing code due to reimplementation of the support for a particular bit-mapped device.

B.2. Layers of Implementation

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

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

sunwindow[3]
implements a manager for overlapping windows. This management includes creating and manipulating windows, maintaining the windows' images, a stream input format for keyboards and mice, and distribution of those user inputs. The associated library is ***libsunwindow.a***.

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

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

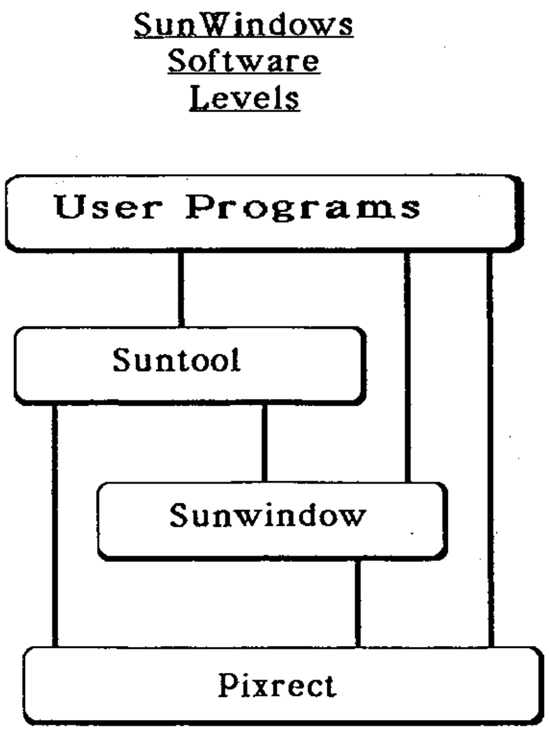


Figure B-1: SunWindows Layers

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

B.2.1. Suntool Layer

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

Several tools are provided with SunWindows. These include:

the shell tool
a terminal emulator,

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

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

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

Source for these programs, and other SunWindows tools, resides in the directory `/usr/src/sun/suntool/`.

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

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

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

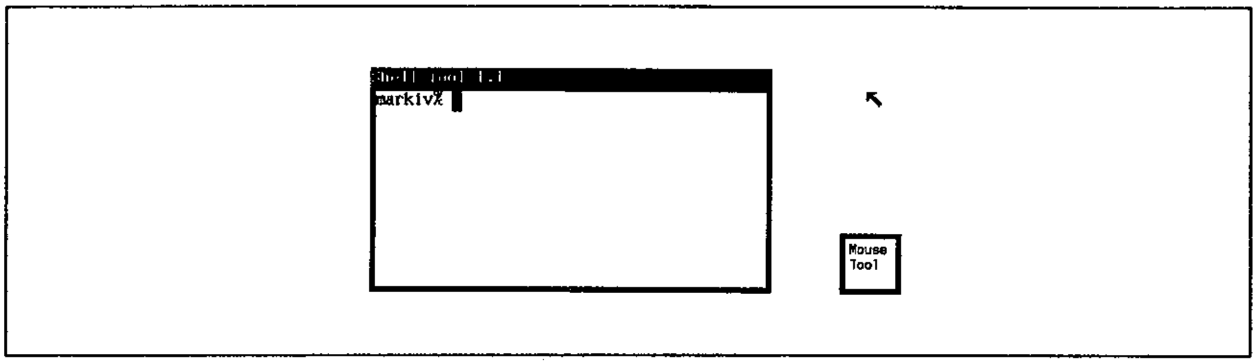


Figure B-2: Standard Tool Window and a Default Icon

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

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

a menu package
: that implements pop-up menus.

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

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

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

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

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

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

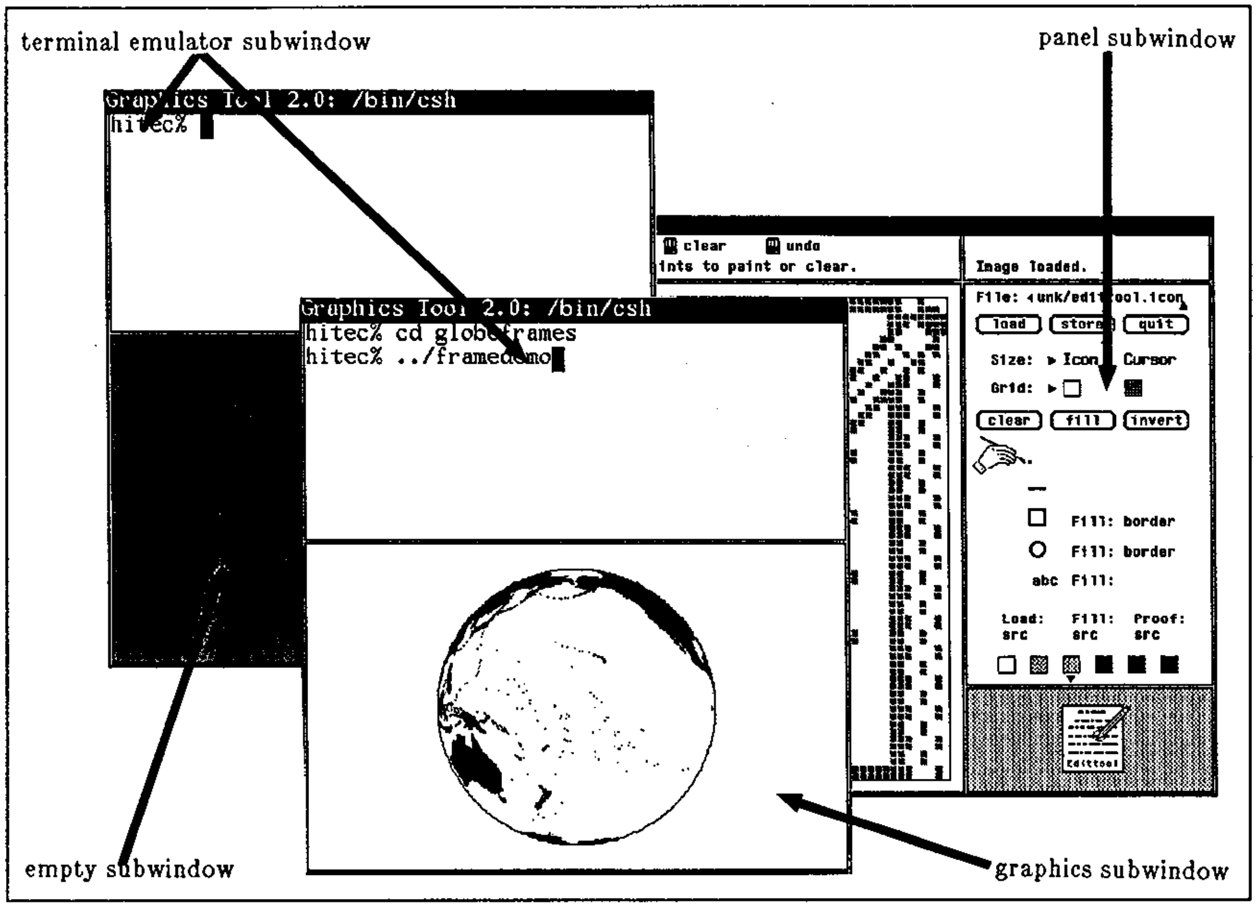
panel subwindow (panel)
: which provides sophisticated mouse and display entry of commands and parameters.

It is the window system analog to entering command-line arguments and typing mnemonic commands to an application. A option subwindow contains a number of items of various types, each item corresponding to one parameter.

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

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

The following figure shows each of the subwindow types:



```
hitec% 
```

```
hitec% cd globframes
hitec% ../framedemo
```

Figure B-3: System Subwindow Types

You can also make a custom subwindow from a skeletal version, as we describe in the chapter ***User-Defined Subwindows***.

B.2.2. Sunwindow Layer

The ***sunwindow*** layer of the system maintains a database of windows. This database is structured as a collection of trees, with one tree per display device. Each tree has a root at the top and descendants toward the bottom. We will use the metaphor of a family tree to describe this database, since it provides a convenient terminology, and the concept of ***age*** is useful for describing the locations of windows in the tree. Note, however, that the metaphor is not exact: it is possible to change a window's position in the tree, hence a window's age is subject to change.

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

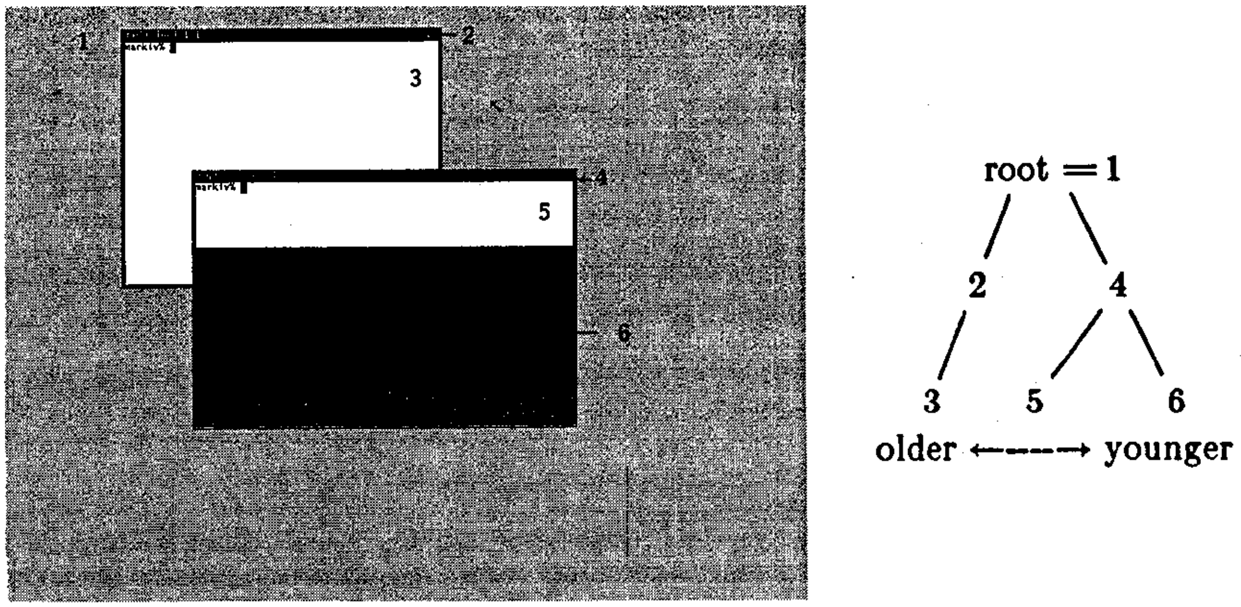


Figure B-4: Window Tree

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

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

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

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

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

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

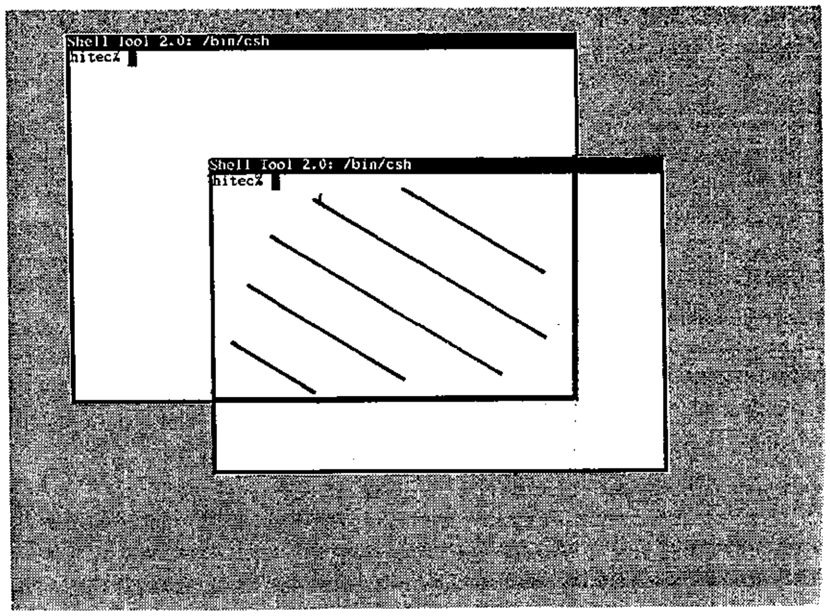


Figure B-5: Damage

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

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

B.2.3. Pixrect Layer

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

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

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

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

B.3. Choosing the Appropriate Layer

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

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

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

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

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

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

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

Index