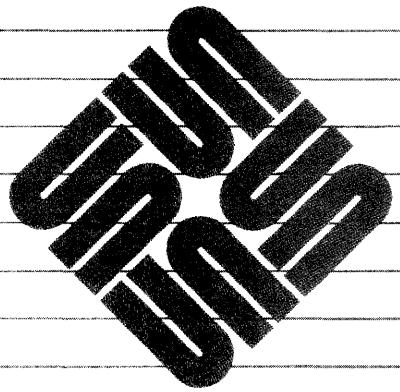
sun®
microsystems

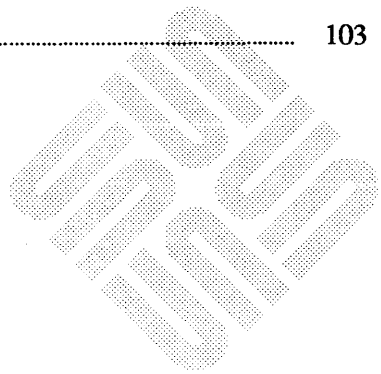
Pixrect Reference Manual



Part Number: 800-1785-10
Revision A, of 9 May 1988

Contents

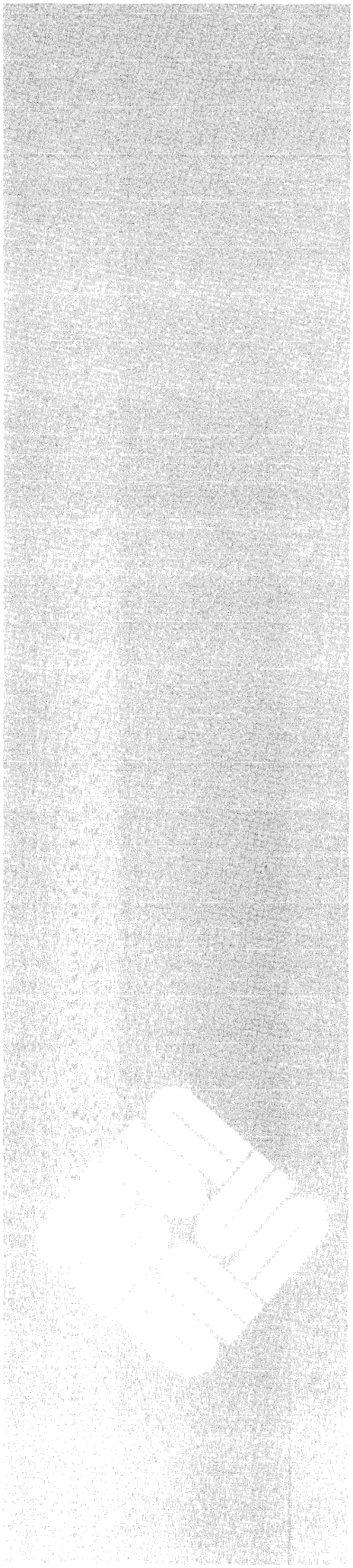
Tables



Figures

1

Introduction



1

Introduction

This document describes the *Pixrect graphics library*, a set of routines that manipulate rectangular arrays of pixel values, on screen or in memory. These routines, called *RasterOps*, are common to all Sun workstations. With these routines, application programs can manipulate the bit-mapped display on any Sun Workstation.

From a software perspective, the Pixrect graphics library is a low-level graphics package, sitting on top of the display device drivers. For most applications, the higher-level abstractions available in *SunView* and the Sun graphic standards libraries are more appropriate. For more information on these other packages, see the preface of this manual for references.

Limitations

The *Pixrect* library is intended only for accessing and manipulating two-dimensional, rectangular regions of a display device in a device-independent fashion.

Windows
: The *Pixrect* library does not support overlapping windows. These can be implemented with memory pixrects by the application, but the *SunView* package already offers a sophisticated, easy-to-use programming interface for this purpose.

Input Devices
: The *Pixrect* library does not have input functions. An application can use the input functions available in *SunView*, or make system calls directly to the raw input devices (see `mouse`(4) and `kbd`(4)).

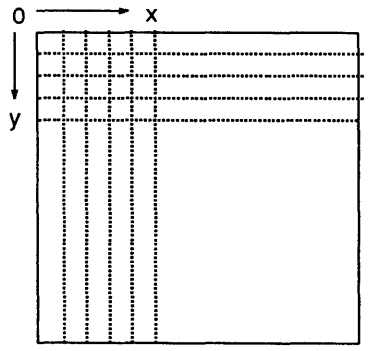
1.1. Overview

This manual is divided into chapters that describe the major features of the Pixrect library. This chapter provides an introduction to the Pixrect library, defining important terms and concepts, and describing the resources available to the programmer. Chapter 2 explains how to write Pixrect programs that can run on all Sun systems. Chapter 3 covers the operations for *opening and manipulating* pixrects. Chapter 4 describes the *text facilities* in the Pixrect library. Chapter 5 discusses *memory pixrects*, rectangular regions of virtual memory that are manipulated as pixrects. Chapter 6 explains the *file I/O* functions in the Pixrect library. These functions can be used to store and retrieve pixrects from disk files. Appendix A is a implementation guide for writing *pixrect device drivers.* Appendix B is a list of the *functions and macros* in the Pixrect library. Appendix

C is a list of *types and structures* in the Pixrect library. Appendix D describes the *curve facilities* in Pixrect .

1.2. Important Concepts

This section describes some of the important concepts behind the Pixrect library. It is not intended to be complete but rather to explain some features of the Pixrect library that make it unique among graphics packages.



Screen Coordinates

The *screen coordinate* system is two dimensional; the origin is in the upper left corner, with *x* and *y* increasing to the right and down. The coordinates describing pixel locations in a pixrect are integers ranging from 0 to the pixrect's width (for x) or height (for y) minus 1. The maximum value for x and y is 32767.

Pixels

A *pixel* is the smallest individual picture element that can be displayed on the screen. A pixel has an address (corresponding to an x and y coordinate) used to specify it, and a value, which controls the color displayed. The pixel address can be absolute (its screen coordinate) or relative to some rectangular sub-region of the screen. A pixel has a depth (the number of bits it contains) which determines the range of colors it can display. A single bit pixel can be only black or white, and are used in monochrome displays. Pixels with more bits can display grayscale values or color. The most common pixel depths are one, eight, sixteen, or twenty-four bits per pixel.

Bitmaps

A *bitmap* is a rectangular region of screen space. Each pixel on the screen corresponds to some number of bits in the screen memory. The value of these bits determines the color of the corresponding pixel. These groups are arranged in an array that can be accessed using the x and y coordinates of the corresponding pixel. A pixrect bitmap can be up to 32767 pixels wide, and up to 32767 pixels high.

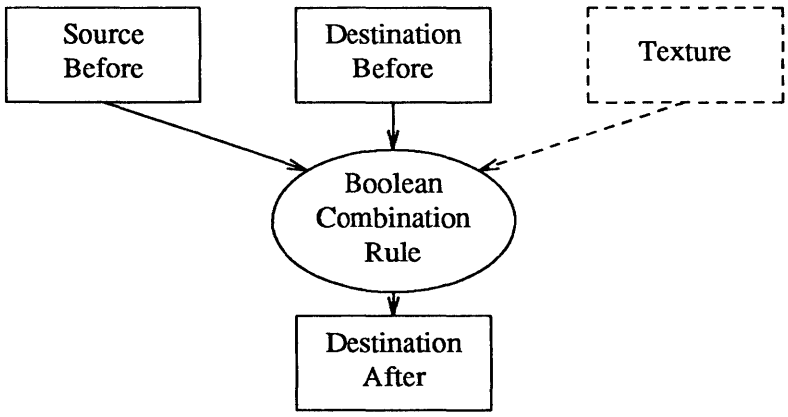
The word "bitmap" can describe the the type of display, indicating it uses raster (rather than vector) display technology, or more commonly, to the images stored in bitmap format. Examples of the second type of bitmap include the screen image, window images, the cursor, or icons.

RasterOps

RasterOps are the legal operations available for modifying pixrects. A rasterop is an operation which takes two bitmaps as arguments: a *source* bitmap, and the current state of the *destination* bitmap. The RasterOp then performs a boolean operation using these arguments, pixel by pixel, writing the final result to the destination bitmap. The source bitmap may be pattern, or defined as a region of some constant value.

The `pr_stencil()` function is the only RasterOp that breaks this rule. Along with the source and destination bitmaps, this function takes an additional argument, a *texture* bitmap, and combines the three in a boolean operation. See Chapter 3 for a more detailed explanation of the RasterOp functions available in the Pixrect graphics library.

Figure 1-1 *RasterOp Function*



Pixrects

A *pixrect* is the graphics analogy to an instance of a *class* used in object-oriented programming languages. It consists of bitmap data and the operations that can be performed on that data. The implementation of the operations and the data itself is hidden from the programmer (the only exception is memory pixrects, whose bitmap data can be directly manipulated. See Chapter 5 for details.) The pixrect is manipulated by using one of the functions in the pixrect library valid for that pixrect (analogous to sending it a *message* in object-oriented Programming.)

A pixrect object can reside on a variety of devices; including different types of graphics displays, memory, and printers. Since the available operations are the same regardless of the device the pixrect resides in, the programmer can ignore device particularities while writing the application.

1.3. Using Pixrects

The general procedure for drawing pictures using pixrects takes three steps:

1. Open a pixrect object.
2. Draw a picture into the pixrect, using the set of valid operations:

   ```
   pr_put()
   pr_vector()
   pr_rop()
   ```

   etc.
3. Close the pixrect.

Primary Pixrect

If the pixrect resides on a display device, the result of each drawing operation becomes visible immediately. Opening a display pixrect will not erase the previous contents of the display. Closing the pixrect also has no effect on the contents of the display.

Secondary Pixrect

A secondary pixrect is a proper subset of its parent pixrect. The results of drawing operations to a secondary pixrect are displayed immediately, if the parent's pixrect is visible. A secondary pixrect can simplify programming, by allowing the programmer to isolate a section of a larger pixrect, sending drawing commands relative to that pixrect, rather than to its parent. Pixrects can be nested to any depth.

Memory Pixrect

A memory pixrect allocates a section of memory in the workstation. Unlike a primary or secondary pixrect, a memory pixrect does clear its bitmap to zeros when opened. Operations done on memory pixrects don't show on the screen. An image in a memory pixrect can be copied to a display pixrect, allowing a simple form of double buffering. A memory pixrect can also be used a buffer or scratch pad, storing bitmaps for later use, or to save the results of previous operations.

Basic Example

The following example draws a diagonal line near the upper corner of the workstation's default display.

Figure 1-2 *Basic Example Program*

```
#include <pixrect/pixrect_hs.h>

main()
{
    Pixrect *screen;

    screen = pr_open("/dev/fb");
    pr_vector(screen, 10, 20, 70, 80, PIX_SET, 1);
    pr_close(screen);
}
```

The header file `<pixrect/pixrect_hs.h>` `#includes` all of the header files necessary for working with the functions, macros and data structures in the Pixrect library.

Compiling

The example program can be compiled as follows:

```
example% cc line.c -o line -lpixrect
```

This command line compiles the program in `line.c`. The `-lpixrect` option causes the C compiler to link the Pixrect library to the application program and create an executable file named `line`.

The sample program can be executed by the SunOS C-shell:

```
example% line
```

A diagonal line will appear in the upper left hand corner of the screen.

Pixrect `lint` Library

Pixrect provides a `lint`(1) library, which allows `lint` to check your program beyond the capabilities of the C compiler. Using the `-lpixrect` flag provides `lint` with pixrect-specific information that prevents bogus error messages. You could use `lint` to check a program called `box.c` with command like this:

```
example% lint box.c -lpixrect
```

Note that most of the error messages generated by `lint` are warnings, and may not necessarily have any effect on the operation of the program. For a detailed explanation of `lint`, see the discussion on `lint` in the *C Programmer's Guide* manual.

1.4. Pixrect Data Structures

All of the important Pixrect data structures are stored in the header files shown in the table below. They can be found in the `/usr/include/pixrect` directory. Use these files to look up the exact definition of a function or macro you're not sure about.

Table 1-1 *Pixrect Header Files*

`pixrect_hs.h`	`#includes` all pixrect files
`pixrect.h`	most pixrect definitions
`memvar.h`	memory pixrects
`pixfont.h`	text operations
`traprop.h`	traprop definitions
`pr_line.h`	defines wide and textured vectors
`pr_planegroups.h`	frame buffers
`pr_util`	internal definitions

2

Portability Considerations

2

Portability Considerations

This chapter addresses Pixrect portability between different Sun architectures. Since Pixrects is a low-level graphics library, it is not completely device independent. Currently, the only Sun architecture that brings up porting issues is Sun386i, the first Sun system to use the Intel 80386 processor. The pixrect software has been designed to minimize porting difficulties; nevertheless, there are some portability factors to take into consideration.

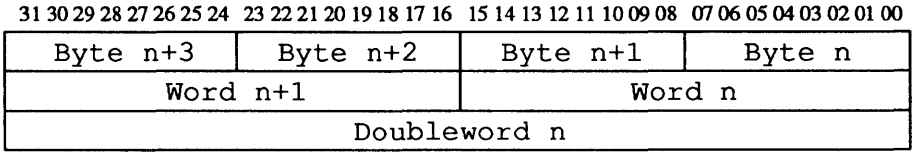
The sections below describe the portability problems caused by the Sun386i system, and their solutions.

2.1. Byte Ordering

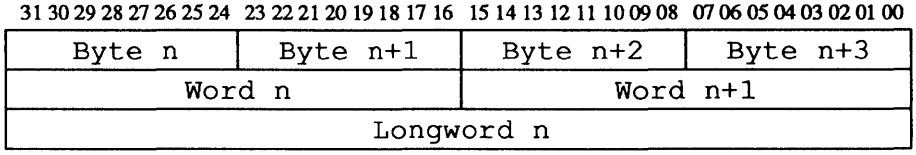
The 80386, 68020, and SPARC are 32-bit processors. This means that all data read or written by these processors pass through 32-bit wide registers. The order in which the data — the bytes and bits — are arranged in the 80386's registers differs from the 680X0 and SPARC families. These differences are illustrated in the figure below:

Figure 2-1 *Byte and Bit Ordering in the 80386, 680X0 and SPARC*

80386



680X0 and SPARC



Byte Swapping and Bit Flipping

The Sun386i is based on the 80386 processor, which handles byte ordering differently than 680X0 and SPARC processors. This affects the Sun386i's interpretation of graphics files — font files, icon files, cursor files, and screendumps — generated by the other two architectures. Typically, frame buffers are accessed as if they were word (i.e., 16-bit integer) devices, or as an array of words. Because the byte ordering of words is different on the two architectures,

transferring a graphics file from one to the other will usually result in a garbled picture.

On 680X0 monochrome frame buffer, the bits are shifted out of the word starting at the most significant bit, bit 15. The upper left-most pixel on the screen is bit 15, word 0 of the frame buffer memory. The next pixel, scanning from left to right as you view the screen, is bit 14. The pixel to the right of the first 16 pixels displayed comes from word 1, bit 15. When interpreted as integers, the most significant and least significant bytes are:

680X0

	MSB	**LSB**
word 0	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
word 1	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0
. . .	. . .	. . .
word n	15 14 13 12 11 10 9 8	7 6 5 4 3 2 1 0

For example, the integer (word) value `0x370D` in word 0 would show up on the 680X0 and SPARC monochrome frame buffer as the pixel sequence: □□■■□■■■□□□□■■□■.

On the 80386 monochrome frame buffer, the bits are shifted out of the word from the least significant bit, bit 0, to the most, bit 15:

80386

	LSB	**MSB**
word 0	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
word 1	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15
. . .	. . .	. . .
word n	0 1 2 3 4 5 6 7	8 9 10 11 12 13 14 15

For example, the integer (word) value `0x370D` in word 0 would show up on the screen with the 80386 frame buffer as the pixel sequence: ■□■■□□□□■■■□■■□□.

The bytes are backward and the bits are in the opposite order. Because graphics files are usually generated as an array of words, the bytes are backward for a typical 80386 frame buffer when handling files generated by 680X0 and SPARC machines. Eight-bit color frame buffers represent each pixel as a byte of data, so the bit order is already correct; conversion only requires byte swapping.

For monochrome frame buffers, each pixel is represented by a single bit; scanning from right to left presents a bit flip and byte swap problem. The right-most (low-order) bit of a bit field now represents the left-most pixel on the screen.

Because of the large number of existing files using it, the 680X0/SPARC format is the standard format for describing graphics images on all Sun systems. This eliminates the need for two sets of files in a mixed-architecture network. Consequently, if you are porting programs to the Sun386i from other Sun systems — programs that access the frame buffer through *SunView* and *Pixrect* — byte and

bit ordering is handled automatically at run time. The 680X0/SPARC format images are converted to 80386 format.

2.2. Flipping Pixrects

Sun386i systems convert 680X0/SPARC format images into 80386 format just before they are used. The procedure that converts them is a new *Pixrect* routine, `pr_flip()`, found only in the Sun386i version of *Pixrect.*

The internal data of a pixrect is referenced by its `pr_data` field.

```
typedef struct pixrect {
        struct  pixrectops *pr_ops;
        struct  pr_size pr_size;
        int     pr_depth;
        caddr_t pr_data;        /*pointer to mpr*/
} Pixrect;
```

If its a memory pixrect, the structure referenced by `pr_data` is:

```
struct mpr_data {
        int     md_linebytes;
        short   *md_image;
        struct  pr_pos md_offset;
        short   md_primary;
        short   md_flags;       /*flag bits*/
    };
```

There are two new flag bits in the `md_flags` word, to control the operation of `pr_flip()`. The flags `MP_REVERSEVIDEO`, `MP_DISPLAY`, and `MP_PLANEMASK` are now followed by `MP_I386` and `MP_STATIC`. If *true*, `MP_I386` indicates that the pixrect in question is already in Sun386i (80386) display format, i.e., it has already been modified by `pr_flip()`. If `MP_STATIC` is *true*, the pixrect in question is a static pixrect. (In practice, this flag is sometimes set for other purposes as well.)

The `pr_flip()` Routine

The `pr_flip()` routine operates on individual pixrects. It takes one argument, a pointer to a pixrect structure, and returns void. When called, it first checks to see if the pixrect has already been flipped (`MP_I386` == TRUE). If not, it flips the image area, 16 bits at a time. First the bit order is reversed, then the bytes are swapped. It will not flip a display pixrect or a secondary pixrect unless it is static (`MP_STATIC` == TRUE).

When a pixrect is modified by a `pr_flip()` call, the changes are limited to the pixrect's image area and the state of the two new `md_flags`. The size of the pixrect structures remains unaltered. The new `md_flags` are ignored by programs running under 680X0 or SPARC.

Pixrects are flipped as they are manipulated by any of the *Pixrect* routines listed below. As an application runs, the rate of pixrect flipping usually declines, since most applications develop a "working set" of active pixrects. Pixrects that are

not used are not flipped.

The routines listed contain checkpoints, where pixrects used in the routines' arguments are examined and flipped (if necessary) by `pr_flip()`:

Table 2-1 *Routines that call* `pr_flip()`

`mem_rop()`
`mem_create()`
`pr_region()`
`pr_vector()`
`pr_dump_init()`
`pf_open()`
`pf_open_private()`
`pr_stencil()`
`pr_batchrop()`
`pr_replrop()`
`pr_get()`
`pr_put()`
`pr_load()`
`pr_dump()`
`icon_display()`
`DEFINE_ICON_FROM_IMAGE`

NOTE *Icons are either static or created with* `icon_load()`. *Static icons can be created with* `DEFINE_ICON_FROM_IMAGE` *Both of these Sunview features are described in the SunView 1 Programmer's Guide.*

Fonts are converted by the `pf_open()` *or* `pf_open_private()` *routines. No other conversions are allowed. The libraries work only with the existing standard font files.*

Guidelines for Sun386i Systems

1. Check code that draws manually into a pixrect. It may not work properly on a Sun386i without modification. The modification required depends on the particulars of the drawing operation.
2. Manual operations (not involving `libpixrect` routines) should be performed on a pixrect **before** converting it to 80386 format.
3. `mem_create()` creates an 80386-format pixrect on Sun386i machines.
4. `mem_point` does **not** set the `MP_I386` flag. The pixrect is still marked **not** flipped.
5. To create an icon, use `mem_point()` to make a pixrect connected to an existing static image or an image that you have created dynamically.
6. Use `DEFINE_ICON_FROM_IMAGE` (SunView) to create static icons. All static icons are initially created in 680X0/SPARC format. They are converted to 80386 format when they are involved in a raster operation.

3

Pixrect Operations

3

Pixrect Operations

Pixrect objects contain procedures to perform the following operations:

- create or destroy a pixrect (`pr_open()`, `pr_region()` and `pr_destroy()`).
- read and write the values of single pixels within a pixrect (`pr_get` and `pr_put()`).
- use RasterOp functions to simultaneously affect multiple pixels within a pixrect:

  `pr_rop` write from a source pixrect to a destination pixrect,

  `pr_stencil` write from a source pixrect to a destination pixrect through a mask pixrect,

  `pr_replrop` replicate a constant source pixrect pattern throughout a destination pixrect,

  `pr_batchrop` write a batch of source pixrects to a sequence of locations within a single destination pixrect,

  `pr_vector`, `pr_line` draw a straight line in a pixrect,

  `pr_polygon_2` draw a polygon in a pixrect.

- draw text (described in chapter 4, *Text Facilities for Pixrects*).
- read write the display's colormap (`pr_getcolormap()`, `pr_putcolormap()`)
- select particular bit-planes in a color pixrect's bitmap for manipulation (`pr_getattributes()`, `pr_putattributes()`)
- control hardware double-buffering (`pr_dbl_get()` and `pr_dbl_set()`).

From an object-oriented viewpoint, all pixrects contain both data and procedures to manipulate its data. This allows pixrects to be device-independent; the pixrect uses the function appropriate for its environment when asked to perform an operation.

From the programmers point of view, pixrects are manipulated using procedure calls embedded in application program. Internally, the pixrect procedures that act the same for all pixrects are implemented by a single procedure for efficiency. The device-dependent calls are macros that access the appropriate procedure within the pixrect object. This is roughly equivalent to passing the pixrect object a *message*, which causes the pixrect to invoke the appropriate *method* (procedure).

Each pixrect object includes an internal pointer to a `pixrectops` structure, that holds the addresses of the particular device-dependent procedures appropriate to that pixrect. Clients may access these procedures in a device-independent fashion, by calling the procedure through the `pixrectops` structure, rather than executing the procedure directly. To simplify this indirection, the *Pixrect* library provides a set of macros which look like simple procedure calls to generic operations, which expand to invocations of the corresponding procedure in the `pixrectops` structure.

In this manual, the description of each operation will specify whether it is a true procedure or a macro, since some of the arguments to macros are expanded multiple times, and could cause errors if the arguments contain expressions with side effects. (In fact, there are two sets of parallel macros, which differ only in how their arguments use the geometry data structures.)

3.1. The `pixrectops` Structure

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

The `pixrectops` structure is a collection of pointers to the device-dependent procedures for a particular device. All other operations are implemented by device-independent procedures. From the object oriented view, this structure provides the procedural interface to the pixrect object, translating messages to methods. This structure is designed to allow expansion; additional functions may be added in future releases.

3.2. Callings Pixrect Procedures

A Pixrect procedure normally expects a number of arguments. These arguments can include: a pointer to the pixrect being manipulated, the dimensions and offset of a subregion within a pixrect, an *ops* argument describing the operation to be performed, among others. This section describes these arguments in detail, and the results returned by the pixrect procedure.

Argument Conventions

In this manual, the conventions listed in Table 3-1 are used in naming the arguments to pixrect operations.

Table 3-1 *Argument Name Conventions*

Argument	*Meaning*
d*suffix*	destination
s*suffix*	source
*prefix***x**	offset to left edge of pixrect
*prefix***y**	offset to top edge of pixrect
*prefix***w**	width of pixrect (0 to 32767)
*prefix***h**	height of pixrect (0 to 32767)

The `x` and `y` values given to functions that operate on a pixrect must be within the boundaries of that pixrect, and be in the range 0 to 32767.

Pixrect Errors

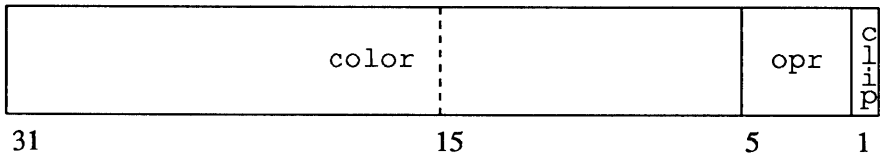
Pixrect operations indicate an error condition in one of two ways, depending on the type of value the operation normally returns. Pixrect operations which return a pointer to a structure return `NULL` when they fail. For pixrect that return an integer status code, a return value of `PIX_ERR` (-1) indicates failure, while 0 indicates the procedure completed successfully. The section describing each pixrect procedure makes note of any exceptions to this convention.

3.3. The Op Argument

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

- A single constant source value may be specified as a `color` in bits 5 – 31 of the `op` argument.
- A RasterOp function is specified in bits 1 – 4 of the `op` argument.
- The clipping which is normally performed by every pixrect operation may be turned off by setting the `PIX_DONTCLIP` flag (bit 0) in the `op`.

Figure 3-1 *Structure of an* `op` *Argument*



Specifying a RasterOp Function

Four bits of the `opr` are used to specify one of the 16 distinct logical functions which combine monochrome source and destination pixels to give a monochrome result. This encoding is generalized to pixels of arbitrary depth by specifying that the function is applied to corresponding bits of the pixels in parallel. Some functions are much more common than others; the most useful are identified in Table 3-2.

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

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

`PIX_SRC` and `PIX_DST` are defined constants, and `PIX_NOT` is a macro. Together, they allow the desired function to be specified by performing the corresponding logical operations on the appropriate constants. Note that `PIX_NOT` must be used in all RasterOp operations; the ones complement (`~`) operator will **not** work.

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

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

The definition of the `PIX_CLR` operation is always false, and hence clears the result:

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

Other common RasterOp functions are defined in the following table:

Table 3-2 *Useful Combinations of RasterOps*

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

Specifying a Color

A single color value can be encoded in bits 5-31 of the op argument. The following macro supports this encoding:

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

Another macro extracts the color field from an encoded op:

```
#define PIX_OPCOLOR(op)      ((op) >> 5)
```

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

The specified color is used by pixrect functions in two situations:

1. If the source pixrect argument is NULL, the rasterop source operand is taken to an infinite rectangle of pixels with the specified color.
2. If the source pixrect has a depth of 1 bit and the destination pixrect has a greater depth, the rasterop source operand is the specified color for each "1" source pixel and zero for each "0" source pixel. A color of zero is treated as a special case; it is converted to the maximum pixel value for the destination pixrect.

If the destination pixrect has a depth of 1 bit, any nonzero color value is treated as 1; for other depths less significant bits of the color value are used. If the destination pixrect is 32 bits deep the encoded color is sign extended.

Controlling Clipping in a RasterOp

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

```
#define PIX_DONTCLIP 0x1
```

The result of a pixrect operation is undefined and may cause a memory fault if `PIX_DONTCLIP` is set and the operation goes out of bounds.

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

Examples of Complete Op Argument Specification

A very simple `op` argument will specify that source pixels be written to a destination, clipping to both operands:

```
op = PIX_SRC;
```

A more complicated example could be used to flip the color of destination pixels between two values wherever pixels in a 1 bit source pixrect are set, with clipping disabled for maximum performance:

```
op = (PIX_DST ^ PIX_SRC) | PIX_COLOR(color1 ^ color2) \
    | PIX_DONTCLIP;
```

3.4. Creation and Destruction of Pixrects

Pixrects are created by the procedures `pr_open()` and `mem_create()`, by the procedures accessed by the macro `pr_region()`, and at compile-time by the macro `mpr_static()`. Pixrects are destroyed by the procedures accessed by the macros `pr_destroy()` and `pr_close()`. `mem_create()` and `mpr_static()` are discussed in Chapter 5; the rest of these are described here.

Create a Primary Display Pixrect

```
Pixrect *pr_open(devicename)
char *devicename;
```

The properties of a non-memory pixrect depend on an underlying UNIX device. Thus, when creating the first pixrect for a device you need to open it by a call to `pr_open()`. The default device name for your display is `/dev/fb` (`fb` stands for *frame buffer*). Any other device name may be used provided that it is a display device, the kernel is configured for it, it exists in the `/dev` directory, and it has pixrect support. For example; `/dev/bwone0`, `/dev/bwtwo0`, `/dev/cgone0` or `/dev/cgtwo0` all can exist on a Sun Workstation, and can be opened with pixrects.

`pr_open()` does not work for creating a pixrect whose pixels are stored in memory; that function is served by the procedure `mem_create()`, discussed in Chapter 5.

`pr_open()` returns a pointer to a primary `Pixrect` structure which covers the entire surface of the named device. If it cannot, it returns `NULL`, and prints a message on the standard error output.

Getting Screen Parameters

In order to write portable programs, it is important to read the screen characteristics directly, rather than assuming them. The pixrect returned by `pr_open()` contains this information. The two most important values are the dimensions of the screen, and the depth (number of bits) of each pixel. The code sample below opens a screen pixrect, then extracts the width, height and depth (in bits) of the screen.

```
#include <pixrect/pixrect_hs.h>   include the proper definitions
#include <stdio.h>

main()
{
    Pixrect *screen, *pr_open();  screen points to screen pixrect
    int height, width, depth;     variables to make things clearer

    screen = pr_open("/dev/fb");  open the pixrect

    width  = screen->pr_size.x;   extract the data in pr_size;
    height = screen->pr_size.y;   width and height are in pixels
    depth  = screen->pr_depth;    get depth in bits

    (void)printf("width = %d, height = %d, bits/pixel = %d0,
            width, height,depth);   display result

    (void)pr_close(screen);       close the pixrect
}
```

Create Secondary Pixrect

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

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

Given an existing pixrect, it is possible to create another pixrect which refers to some or all of the pixels in the parent pixrect. This *secondary pixrect* is created by a call to the procedures invoked by the macros `pr_region()` and `prs_region()`.

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

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

Release Pixrect Resources

```
#define pr_close(pr)
Pixrect *pr;

#define pr_destroy(pr)
Pixrect *pr;

#define prs_destroy(pr)
Pixrect *pr;
```

The macros `pr_close()`, `pr_destroy()` and `prs_destroy()` invoke device-dependent procedures to destroy a pixrect, freeing resources that belong to it. The procedure returns 0 if successful, `PIX_ERR` if it fails. It may be applied to either primary or secondary pixrects. If a primary pixrect is destroyed before secondary pixrects which refer to its pixels, those secondary pixrects are invalidated; attempting any operation but `pr_destroy()` on them is an error. The three macros are identical; they are all defined for reasons of history and stylistic consistency.

3.5. Single-Pixel Operations

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

Get Pixel Value

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

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

The macros `pr_get` and `prs_get` invoke device-dependent procedures to retrieve the value of a single pixel. `pr` indicates the pixrect in which the pixel is to be found; `x` and `y` are the coordinates of the pixel. For `prs_get`, the same arguments are provided in the single struct `srcprpos`. The value of the pixel is returned as a 32-bit integer; if the procedure fails, it returns `PIX_ERR`.

Set Pixel Value

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

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

The macros `pr_put()` and `prs_put()` invoke device-dependent procedures to store a value in a single pixel. `pr` indicates the pixrect in which the pixel is to be found; `x` and `y` are the coordinates of the pixel. For `prs_put()`, the same arguments are provided in the single struct `dstprpos`. `value` is truncated on the left if necessary, and stored in the indicated pixel. If the procedure fails, it returns `PIX_ERR`.

3.6. Multi-Pixel Operations

The following operations all apply to multiple pixels at one time: `pr_rop()`, `pr_stencil()`, `pr_replrop()`, `pr_batchrop()`, `pr_polygon_2()`, and `pr_vector()`. With the exceptions of `pr_vector()` and `pr_polygon_2()`, they refer to rectangular areas of pixels. They all use a common mechanism, the `op` argument described in the previous section, to specify how pixels are to be set in the destination. Appendix D. describes the `pr_traprop()` curve rendering function.

RasterOp Source to Destination

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

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

The `pr_rop()` and `prs_rop()` macros invoke device-dependent procedures that perform the indicated raster operation from a source to a destination pixrect. `dpr` addresses the destination pixrect, whose pixels will be affected; `(dx, dy)` is the origin (the upper-left pixel) of the affected rectangle; `dw` and `dh` are the width and height of that rectangle. `spr` specifies the source pixrect, and `(sx, sy)` an origin within it. `spr` may be `NULL`, to indicate a constant source specified in the `op` argument, as described previously; in this case `sx` and `sy` are ignored. The `op` argument specifies the operation which is performed; its construction is described in preceding sections.

`pr_rop()` is the only pixrect function that can have its source and destination be overlapping areas of the same pixrect. Doing this with any other operation generates an error.

For `prs_rop()`, the `dpr`, `dx`, `dy`, `dw` and `dh` arguments are all collected in a `pr_subregion` structure.

Raster operations are clipped to the source dimensions, if those are smaller than the destination size given. `pr_rop()` procedures return `PIX_ERR` if they fail, 0 if they succeed.

Source and destination pixrects generally must be the same depth. The only exception allows monochrome pixrects to be sources to a destination of any depth. In this case, source pixels = 0 are interpreted as 0 and source pixels = 1 are written as the color value from the `op` argument. If the color value in the `op` argument is 0, source pixels = 1 are written as the maximum value which can be stored in a destination pixel.

See the example program in Figure 5-2 for an illustration of `pr_rop()`.

RasterOps through a Mask

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

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

The `pr_stencil` and `prs_stencil` macros invoke device-dependent procedures that perform the indicated raster operation from a source to a destination pixrect only in areas specified by a third (stencil) pixrect. `pr_stencil()` is identical to `pr_rop()` except that the source pixrect is written through a stencil pixrect which functions as a spatial write-enable mask. The stencil pixrect must be a monochrome memory pixrect. The indicated raster operation is applied only to destination pixels where the stencil pixrect is non-zero. Other destination pixels remain unchanged. The rectangle from (`sx, sy`) in the source pixrect `spr` is aligned with the rectangle from (`stx, sty`) in the stencil pixrect `stpr`, and written to the rectangle at (`dx, dy`) with width `dw` and height `dh` in the destination pixrect `dpr`. The source pixrect `spr` may be `NULL`, in which case the color specified in `op` is painted through the stencil. Clipping restricts painting to the intersection of the destination, stencil and source rectangles. `pr_stencil()` procedures return `PIX_ERR` if they fail, 0 if they succeed.

Replicating the Source Pixrect

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

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

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

The `pr_replrop()` procedure replicates a source pattern repeatedly to cover a destination area. `dpr` indicates the destination pixrect. The area affected is described by the rectangle defined by `dx, dy, dw, dh`. `spr` indicates the source pixrect, and the origin within it is given by (`sx, sy`). The corresponding `prs_replrop()` macro generates a call to `pr_replrop()`, expanding its `dsubreg` into the five destination arguments, and `sprpos` into the three source arguments. `op` specifies the operation to be performed, as described above in Section 3.3, *The Op Argument.*

The effect of `pr_replrop()` is the same as though an infinite pixrect were constructed using copies of the source pixrect laid immediately adjacent to each other in both dimensions, and then a `pr_rop()` was performed from that source

to the destination. For instance, a standard gray pattern may be painted across a portion of the screen by constructing a pixrect that contains exactly one tile of the pattern, and by using it as the source pixrect.

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

pr_replrop() returns PIX_ERR if it fails, or 0 if it succeeds. Internally pr_replrop() may use pr_rop() procedures. In this case, pr_rop() errors are detected and returned by pr_replrop().

Multiple Source to the Same Destination

```
#define pr_batchrop(dpr, dx, dy, op, items, n)
Pixrect *dpr;
int dx, dy, op, n;
struct pr_prpos items[];

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

Applications such as displaying text perform the same operation from a number of source pixrects to a single destination pixrect in a fashion that is amenable to global optimization.

The pr_batchrop and prs_batchrop macros invoke device-dependent procedures that perform raster operations on a sequence of sources to successive locations in a common destination pixrect. items is an array of pr_prpos structures used by a pr_batchrop() procedure as a sequence of source pixrects. Each item in the array specifies a source pixrect and an advance in x and y. The whole of each source pixrect is used, unless it needs to be clipped to fit the destination pixrect. The advance is used to update the destination position, not as an origin in the source pixrect.

pr_batchrop() procedures take a destination, specified by dpr, dx and dy, or by dstpos in the case of prs_batchrop(); an operation specified in op, as described in Section 3.3. and an array of pr_prpos addressed by the argument items, and whose length is given in the argument n.

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

The most common application of `pr_batchrop()` procedures is in painting text; additional facilities to support this application are described in Chapter 4. Note that the definition of `pr_batchrop()` procedures supports variable-pitch and rotated fonts, and non-Roman writing systems, as well as simpler text.

`pr_batchrop()` procedures return `PIX_ERR` if they fail, 0 if they succeed. Internally `pr_batchrop()` may use `pr_rop()` procedures. In this case, `pr_rop()` errors are detected and returned by `pr_batchrop()`.

Draw Vector

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

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

The `pr_vector` and `prs_vector` macros invoke device-dependent procedures that draw a vector one unit wide between two points in the indicated pixrect. `pr_vector()` procedures draw a vector in the pixrect indicated by `pr`, with endpoints at (`x0,y0`) and (`x1,y1`), or at `pos0` and `pos1` in the case of `prs_vector()`. Portions of the vector lying outside the pixrect are clipped as long as `PIX_DONTCLIP` is 0 in the `op` argument. The `op` argument is constructed as described in Section 3.3. and `value` specifies the resulting value of pixels in the vector. If the color in `op` is non-zero, it takes precedence over the `value` argument.

Any vector that is not vertical, horizontal or 45 degree will contain *jaggies*. This phenomenon, known as *aliasing*, is due to the digital nature of the bitmap screen. It can be visualized by imagining a vertical vector. Displace one endpoint horizontally by a single pixel. The resulting line will have to jog over a pixel at some point in the traversal to the other endpoint. Balancing the vector guarantees that the jog will occur in the middle of the vector. `pr_vector()` draws *balanced* vectors. (The technique used is to balance the Bresenham error term). The vectors are balanced according to their endpoints as given and not as clipped, so that the same pixels will be drawn regardless of how the vector is clipped.

See the example program in Figure 1-2 for an illustration of `pr_vector()`.

Draw Textured Polygon

```
pr_polygon_2(dpr, dx, dy, nbnds, npts, vlist, op, spr, sx, sy
Pixrect *dpr, *spr;
int dx, dy
int nbnds, npts[];
struct pr_pos *vlist;
int op, sx, sy;
```

The `pr_polygon_2()` function performs a raster operation on a polygonal area of the destination pixrect. The source can be a pattern or a constant color value.

The destination polygon is described by `nbnds`, `npts` and `vlist`. `nbnds` is the number of individual closed boundaries (vertex lists) in the polygon. A complex polygon may have one boundary for its exterior shape and several boundaries delimiting interior holes. The boundaries may intersect themselves or each other. Only those destination pixels having an odd *winding number* are painted. That is, if any line connecting a pixel to infinity crosses an odd number of boundary edges, the pixel will be painted.

For each of the `nbnds` boundaries, `npts` specifies the number of points in the boundary. The `vlist` array contains the boundary points for all of the boundaries, in order. The total number of points in `vlist` is equal to the sum of the `nbnds` elements in the npts array. `pr_polygon_2()` automatically joins the last point and first point to close each boundary. If any boundary has fewer than 3 points, `pr_polygon_2()` returns `PIX_ERR`.

The destination coordinates `dx`, and `dy` are added to each point in `vlist`, so the same `vlist` can be used to draw polygons in different destination locations.

If the source pixrect `spr` is non-null, it is replicated in the x and y directions to cover the entire destination area. The point (`sx`, `sy`) in this extended source pixrect is aligned with the point (`dx`, `dy`) in the destination pixrect.

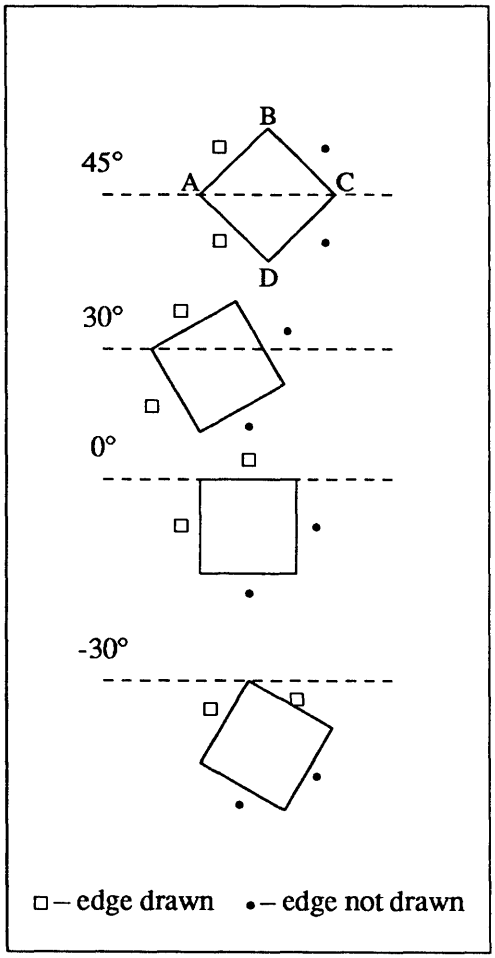
Polygons drawn by `pr_polygon_2()` are *semi-open* in the sense that on some of the edges, pixels are not drawn where a vector drawn with same coordinates would go. The reason is to allow identical polygons (same size and orientation) to exactly tile the destination pixrect with no gaps and no overlaps.

In Figure 3-3 the edges AB and DA are drawn, whereas edges BC and CD aren't.

Figure 3-2 *Example Program using* `pr_polygon_2()`

```
#include <pixrect/pixrect_hs.h>

#define CENTERX(pr) ((pr)->pr_size.x / 2)
#define NULLPR      ((Pixrect *) 0)

static struct pr_pos
    /* 45 degrees */
    vlist0[4] = { {0, 0}, { 71, -71}, {141,   0}, { 71,  71} },
    /* 30 degrees */
    vlist1[4] = { {0, 0}, { 87, -50}, {137,  37}, { 50,  87} },
    /* 0 degrees */
    vlist2[4] = { {0, 0}, {100,   0}, {100, 100}, {  0, 100} },
    /* -30 degrees */
    vlist3[4] = { {0, 0}, { 87,  50}, { 37, 137}, {-50,  87} };

main()
{
    Pixrect *pr;
    static int npts[1] = { 4 };

    if (!(pr = pr_open("/dev/fb")))
        exit(1);

    pr_polygon_2(pr, CENTERX(pr), 100, 1, npts, vlist0,
        PIX_SET, NULLPR, 0, 0);

    pr_polygon_2(pr, CENTERX(pr), 300, 1, npts, vlist1,
        PIX_SET, NULLPR, 0, 0);

    pr_polygon_2(pr, CENTERX(pr), 500, 1, npts, vlist2,
        PIX_SET, NULLPR, 0, 0);

    pr_polygon_2(pr, CENTERX(pr), 700, 1, npts, vlist3,
        PIX_SET, NULLPR, 0, 0);

    pr_close(pr);
    exit(0);
}
```

Figure 3-3 *Four Polygons Drawn with* `pr_polygon_2()`



Draw Textured or Solid Lines with Width

```
#define pr_line(pr, x0, y0, x1, y1, brush, tex, op)
Pixrect *pr;
int x0, y0, x1, y1;
struct pr_brush *brush;
struct pr_texture *tex;
int op;
```

The `pr_line` macro draws a textured line based on the Bresenham line drawing algorithm, using a pen-up, pen-down approach. The programmer can define an pattern (of arbitrary length), or use a predefined default pattern (dash-dot, dotted, etc.). All pattern segments (and their corresponding offsets) can automatically adjust, according to the angle at which the line is drawn.

If the brush pointer is NULL, or if the width is 0 or 1, a single width vector is drawn.

The line is drawn in the pixrect indicated by `pr`, with endpoints at (`x0, y0`) and (`x1, y1`). The `brush` field is a pointer to a structure of type `pr_brush` which holds the width of the line segments to be rendered. The `pr_brush` structure is defined in the header file <`pixrect/pr_line.h`> as follows:

```
typedef struct pr_brush {
    int width;
} Pr_brush;
```

If the `tex` pointer is NULL, a solid vector is drawn. The `tex` field is a pointer to a structure of type `pr_texture`. The `pr_texture` structure is defined in the header file <pixrect/pr_line.h> as follows (fields that begin with the prefix `res_` are reserved for program internals, and are not user-definable):

```
typedef struct pr_texture {
    short *pattern;
    short offset;
    struct pr_texture_options {
        unsigned startpoint : 1,
        endpoint : 1,
        balanced : 1,
        givenpattern : 1,
        res_fat : 1,
        res_poly: 1,
        res_mvlist : 1,
        res_right : 1,
        res_close : 1;
    } options;
    short res_polyoff;
    short res_oldpatln;
    short res_fatoff;
} Pr_texture;
```

`pattern` is a pointer to an array of short integers which contain the length of each segment in the pattern. The lengths are in units of pixels. If the line is drawn at an angle, the lengths drawn are automatically adjusted (if the `given-pattern` field set to 0) to correspond to the length of the pattern if a horizontal or vertical line was drawn. This array must be null-terminated. The first segment of the `pattern` array is assumed to be pen-down, and following segments alternate.

The addresses of the following predefined `pattern` arrays may be stored in the `pattern` field of the texture structure as well:

```
extern short pr_tex_dotted[];
extern short pr_tex_dashed[];
extern short pr_tex_dashdot[];
extern short pr_tex_dashdotdotted[];
extern short pr_tex_longdashed[];
```

The programmer-defined elements of the `pattern` array are not altered within the routine, allowing multiple calls using the same pattern. `offset` is an integer offset into the pattern, specified in pixels. Since the first segment of the `pattern` array is assumed to be pen-down, you must specify an `offset` to

start on a pen-up segment. `offset` is adjusted according to the angle at which the line is drawn if the original pattern was adjusted (dependent upon the `givenpattern` bit, described later). Because of integer approximation, the adjusted `offset` could vary plus or minus one pixel from the exact adjusted `offset`.

In the options bit fields, if `startpoint` is set, the first point is always drawn, and if `endpoint` is set, the last point is drawn; if these are not specified, the line will be drawn with no extra pixels set. The `balanced` bit field effectively centers the pattern within the line by computing an offset into the pattern. If the `givenpattern` bit is set, the pattern is drawn without true length correction, at any angle; this increases performance. However, the pattern of radiating lines from a common center will form concentric squares instead of circles. If the `givenpattern` bit is not set, the segment length of each element of the pattern is adjusted according to the angle at which the line is drawn. The true (angle-dependent) segment lengths are computed for one period of the pattern, using an incremental algorithm which approximates the formula:

angle_pattern_length = *given_pattern_length* * cos (*angle*)

where all units are in pixels, and *angle* is measured from the positive *x*-axis. Since the algorithm angle-corrects for one period of the pattern, the longer its period, the more exact the results are.

The `op` argument specifies the raster operations used to produce destination pixel values and color.

Draw Textured or Solid Polylines with Width

```
pr_polyline(dpr, dx, dy, npts, ptlist, mvlist, brush, tex, op)
Pixrect *dpr;
int dx, dy, npts;
struct pr_pos *ptlist;
u_char *mvlist;
struct pr_brush *brush;
struct pr_texture *tex;
int op;
```

`pr_polyline` draws a polyline, or a series of disjoint polylines, using the features available in `pr_line`. The polyline is drawn in the destination pixrect indicated by `dpr`, with `dx` and `dy` being the offset into the destination pixrect for vertices to be translated in x and y, respectively. `npts` is the number of vertices in the polyline (which is always the number of lines plus 1). The `ptlist` field is an array of `npts` structures of type `pr_pos` (which hold vertices). The `mvlist` field is a pointer to an array of `npts` elements in which if any element after the first is non-zero, a segment is not drawn to that vertex. The first element of the `mvlist` array controls whether the polyline(s) are automatically closed; if set, each continuous polyline is closed. If disjoint polylines are not desired (no `mvlist` is specified), the constants `POLY_CLOSE` and `POLY_DONTCLOSE` determine this behavior. `POLY_CLOSE` and `POLY_DONTCLOSE` are defined as follows:

```
#define POLY_CLOSE   ((u_char *) 1)
#define POLY_DONTCLOSE   ((u_char *) 0)
```

The `brush` field is a pointer to a structure of type `pr_brush`, and the `tex` field is a pointer to a structure of type `pr_texture`. If the `tex` pointer is null, a solid vector is drawn. If the `brush` structure is null, single-width vectors are drawn. `op` specifies the raster operations used to produce destination pixel values and color. `brush` and `tex` are described in detail under `pr_line`.

Draw Multiple Points

```
pr_polypoint(dpr, dx, dy, npts, ptlist, op)
Pixrect *dpr;
int dx, dy, npts;
struct pr_pos *ptlist;
int op;
```

The `pr_polypoint` routine draws an array of points on the screen under the control of the `op` argument. The array of points is drawn in the destination pixrect `dpr`, with an offset specified by the arguments `dx` and `dy`. `Npts` is the number of points to be rendered, and `ptlist` is a pointer to an array of structures of type `pr_pos`, which hold the vertices for each point. Color is encoded in the `op` argument. Portions of the array outside the pixrect are clipped unless the `PIX_DONTCLIP` flag is set in the `op` argument.

3.7. Colormap Access

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

Sun grayscale workstations normally use the red video signal to drive the monitor. However, when writing an application to run on a grayscale workstation it is a good idea to load the red, green, and blue components of each colormap entry with the same value. This will ensure that the application will also run properly on a color workstation.

Get Colormap Entries

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

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

The macros `pr_getcolormap` and `prs_getcolormap` invoke device-dependent procedures to read all or part of a colormap into arrays in memory.

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

`pr` identifies the pixrect whose colormap is to be read; the `count` entries starting at `index` (zero origin) are read into the three arrays.

For monochrome pixrects the same value is read into corresponding elements of the `red`, `green` and `blue` arrays. These array elements will have their bits either all cleared, indicating `black`, or all set, indicating `white`. By default,

the 0th (*background*) element is white, and the 1st (*foreground*) element is black. Colormap procedures return (-1) if the index or count are out of bounds, and 0 if they succeed.

Set Colormap Entries

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

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

The macros `pr_putcolormap` and `prs_putcolormap` invoke device-dependent procedures to store from memory into all or part of a colormap. These two macros have identical definitions; both are defined to allow consistent use of one set of names for all operations. The `count` elements starting at `index` (zero origin) in the colormap for the pixrect identified by `pr` are loaded from corresponding elements of the three arrays. For monochrome pixrects, the only value considered is `red[0]`. If this value is 0, then the pixrect will be set to a dark background and light foreground. If the value is non-zero, the foreground will be dark, e.g. black-on-white. Monochrome pixrects are dark-on-light by default.

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

Inverted Video Pixrects

```
pr_blackonwhite(pr, min, max)
Pixrect *pr;
int min, max;

pr_whiteonblack(pr, min, max)
Pixrect *pr;
int min, max;

pr_reversevideo(pr, min, max)
Pixrect *pr;
int min, max;
```

Video inversion is accomplished by manipulation of the colormap of a pixrect. The colormap of a monochrome pixrect has two elements. The procedures `pr_blackonwhite`, `pr_whiteonblack` and `pr_reversevideo` provide video inversion control. These procedures are ignored for memory pixrects.

In each procedure, `pr` identifies the pixrect to be affected; `min` is the lowest index in the colormap, specifying the background color, and `max` is the highest index, specifying the foreground color. These will most often be 0 and 1 for monochrome pixrects; the more general definitions allow colormap-sharing schemes.

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

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

`pr_reversevideo()` exchanges the `min` and `max` color intensities.

Note: These procedures are intended for global foreground/background control, not for local highlighting. For monochrome frame buffers, `subsequent` operations will have inverted intensities. For color frame buffers, the colormap is modified immediately, which affects everything in the display.

3.8. Attributes for Bitplane Control

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

Get Plane Mask Attributes

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

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

The macros `pr_getattributes()` and `prs_getattributes()` invoke device-dependent procedures that retrieve the mask which controls which planes in a pixrect are affected by other pixrect operations. `pr` identifies the pixrect; its current bitplanes mask is stored into the word addressed by `planes`. If `planes` is `NULL`, no operation is performed.

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

Put Plane Mask Attributes

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

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

The macros `pr_putattributes()` and `prs_putattributes()` invoke device-dependent procedures that manipulate a mask which controls which planes in a pixrect are affected by other pixrect operations. The two macros are

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

`pr` identifies the pixrect to be affected. The `planes` argument is a pointer to a bitplane write-enable mask. Only those planes corresponding to mask bits having a value of 1 will be affected by subsequent pixrect operations. If `planes` is `NULL`, no operation is performed.

Note: If any `planes` are masked off by a call to `pr_putattributes()`, no further write access to those planes is possible until a subsequent call to `pr_putattributes()` unmasks them. However, these planes can still be read.

3.9. Plane Groups

A *plane group* is a subset of a frame buffer pixrect. Each plane group is a collection of one or more related bit planes with stored state (plane mask, color map, etc.). Each pixrect has a current plane group which is the target of attribute, color map, and rendering operations.

A plane group is described by a small constant in the header file `<pixrect/pr_planegroups.h>`:

```
#define PIXPG_CURRENT           0
#define PIXPG_MONO              1
#define PIXPG_8BIT_COLOR        2
#define PIXPG_OVERLAY_ENABLE    3
#define PIXPG_OVERLAY           4
```

Plane group `0` is the currently active plane group for the pixrect.

A plane group is encoded as a 7-bit field in the pixrect attribute word.

Determine Supported Plane Groups

```
ngroups = pr_available_plane_groups(pr, maxgroups, groups);
Pixrect *pr;
int maxgroups;
char groups[maxgroups]
```

`pr_available_plane_groups` provides a means by which you determine which plane groups are supported by the machine you are working on. `pr_available_plane_groups` fills the character array `groups` with true (`1`) values for the plane groups implemented by the pixrect `pr`. The entry for the current plane group (`groups[0]`) array is always set to false (`0`). The size of `groups` is passed to the function as `maxgroups` to avoid overwriting the end of the array.

`pr_available_plane_groups` returns the index of the highest-numbered implemented plane group plus one.

Get Current Plane Group

```
group = pr_get_plane_group(pr);
Pixrect *pr;
```

`pr_get_plane_group` returns the current plane group number for the pixrect `pr`. If the current plane group is unknown, the function returns `PIXPG_CURRENT`.

Set Plane Group and Mask

```
void pr_set_plane_group(pr, group);
Pixrect *pr;
int group;

void pr_set_planes(pr, group, planes)
Pixrect *pr;
int group;
int planes;
```

`pr_set_plane_group` sets the current plane group for the pixrect `pr` to the value given by `group`. If this plane group is `PIXPG_CURRENT` or unimplemented, `pr_set_plane_group` does nothing.

The `pr_set_planes` function is equal to a `pr_set_plane_group(pr, group)` followed by `pr_putattributes(pr, &planes)`. `planes` contains a bitplane write-enable mask. Only those planes corresponding to mask bits having a value of `1` will be affected by subsequent pixrect operations. However, the other planes can still be read.

3.10. Double Buffering

Some frame buffers have double buffering support implemented in hardware. Two pixrect commands, `pr_dbl_get()`, and `pr_dbl_set()` allow you to inquire about and control a double-buffered display device. The pixrect interface assigns two names to the buffers in the display; `PR_DBL_A` for one, and `PR_DBL_B` for the other.

A buffer can be *displayed*, *read*, or *written*. When a buffer is displayed, its stored image is shown on the screen. If the software requests that the other buffer be displayed, the hardware doesn't switch to the new buffer until the next vertical retrace of the screen. This prevents any flicker from showing on the screen during the change between buffers. A buffer can be read or written, using pixrect commands, at any time.

Get Double Buffering Attributes

```
state = pr_dbl_get(pr, attribute)
Pixrect *pr;
int attribute;
```

This function shows the current attributes of the double buffer. You can inquire about the state of the display device by executing `pr_dbl_get` with a particular attribute value, then examining the function's return value. The legal attributes are listed below:

```
#define PR_DBL_AVAIL    1
#define PR_DBL_DISPLAY  2
#define PR_DBL_WRITE    3
#define PR_DBL_READ     4
```

The `PR_DBL_AVAIL` returns `PR_DBL_EXISTS` if display device has hardware double buffering capacity; otherwise, it returns `NULL`. The other attributes indicate which buffer on the device is being displayed, which can be written to, etc. The possible state values for these attributes is given below:

```
#define PR_DBL_A     2
#define PR_DBL_B     3
#define PR_DBL_BOTH  4
#define PR_DBL_NONE  5
```

Not all return values are possible with each attribute. The values that can be returned for a given attribute a shown in the table below:

Table 3-3 `pr_dbl_get()` *Attributes*

Attribute	*Possible Values Returned*
PR_DBL_AVAIL	PR_DBL_EXISTS
PR_DBL_DISPLAY	PR_DBL_A, PR_DBL_B
PR_DBL_WRITE	PR_DBL_A, PR_DBL_B, PR_DBL_BOTH, PR_DBL_NONE
PR_DBL_READ	PR_DBL_A, PR_DBL_B

Set Double Buffering Attributes

```
void pr_dbl_set(pr, attribute_list)
Pixrect *pr;
int *attribute_list;
```

The `pr_dbl_set()` function changes the state of the double buffering display. It controls the buffer being displayed, and selects the buffer(s) affected by pixrect reads and writes. The possible attributes for `pr_dbl_set()` are given below:

```
#define PR_DBL_DISPLAY              2
#define PR_DBL_WRITE                3
#define PR_DBL_READ                 4
#define PR_DBL_DISPLAY_DONTBLOCK    5
```

An attribute list is an integer array consisting of pairs of attributes and the value the attribute should be set to. The last element of the array should be zero. If the display is already in the state requested, the function simply returns.

If the `PR_DBL_DISPLAY` attribute is in the list, then the function may block for up to a single video frame's time (15 ms), waiting for the next vertical retrace. This insures that the next pixrect operation won't alter the buffer while it's still being displayed. Applications that won't write to the buffer for at least 15 ms after changing the displayed buffer, and who need maximum throughput can use `PR_DBL_DISPLAY_DONTBLOCK`. This attribute changes the display without blocking the process until the next vertical retrace.

NOTE *Programmers should use* `PR_DBL_DISPLAY_DONTBLOCK` *with caution. If the application starts writing too early, it will modify the buffer while it is still being displayed.*

The values that can be paired with the attributes are shown below:

```
#define PR_DBL_A    2
#define PR_DBL_B    3
#define PR_DBL_BOTH 4
```

Not all of the values can be paired with all of the attributes; the allowed pairings are shown in the table below:

Table 3-4 `pr_dbl_set()` *Attributes*

Attribute	*Possible Values to Set*
PR_DBL_WRITE	PR_DBL_A, PR_DBL_B, PR_DBL_BOTH
PR_DBL_READ	PR_DBL_A, PR_DBL_B
PR_DBL_DISPLAY_DONTBLOCK	PR_DBL_A, PR_DBL_B
PR_DBL_DISPLAY	PR_DBL_A, PR_DBL_B

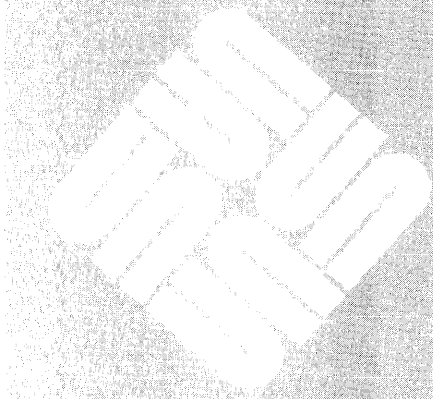
3.11. Efficiency Considerations

For maximum execution speed, remember the following points when you write pixrect programs:

- `pr_get` and `pr_put()` are relatively slow. For fast random access of pixels it is usually faster to read an area into a memory pixrect and address the pixels directly.
- `pr_rop()` is fast for large rectangles.
- `pr_vector()` is fast.
- functions run faster when clipping is turned off. Do this only if you can guarantee that all accesses are within the pixrect bounds.
- `pr_rop()` is three to five times faster than `pr_stencil()`.
- `pr_batchrop()` cuts down the overhead of painting many small pixrects.
- For small standard shapes `pr_rop()` should be used instead of `pr_polygon_2()`.
- `pr_polyline()` is an efficient way to draw a series of vectors.
- `pr_polypoint()` is faster than a series of `pr_puts()` or single pixel `pr_rops()`. It is useful for implementing new primitives such as curves.
- The `PR_DBL_DISPLAY_DONTBLOCK` attribute of `pr_dbl_set()`, if used appropriately, can speed up animation sequences.

4

Text Facilities for Pixrects



4

Text Facilities for Pixrects

The *Pixrect* library contains higher-level facilities for displaying text. These facilities fall into two main categories: a standard format for describing fonts and character images, including routines for processing them; and a set of routines that take a string of text and a font, and handle various parts of painting that string in a pixrect.

4.1. Pixfonts and Pixchars

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

The `pixchar` structure defines the format of a single character in a font. The actual image of the character is a pixrect (a separate pixrect for each character) addressed by `pc_pr`. The entire pixrect gets painted. Characters that do not have a displayable image will have NULL in their entry in `pc_pr`. `pc_home` is the origin of pixrect `pc_pr` (its upper left corner) relative to the character origin. A character's origin is the leftmost end of its *baseline*, that is the lowest point on characters without descenders. Figure 4-1 illustrates the `pc_pr` origin and the character origin.

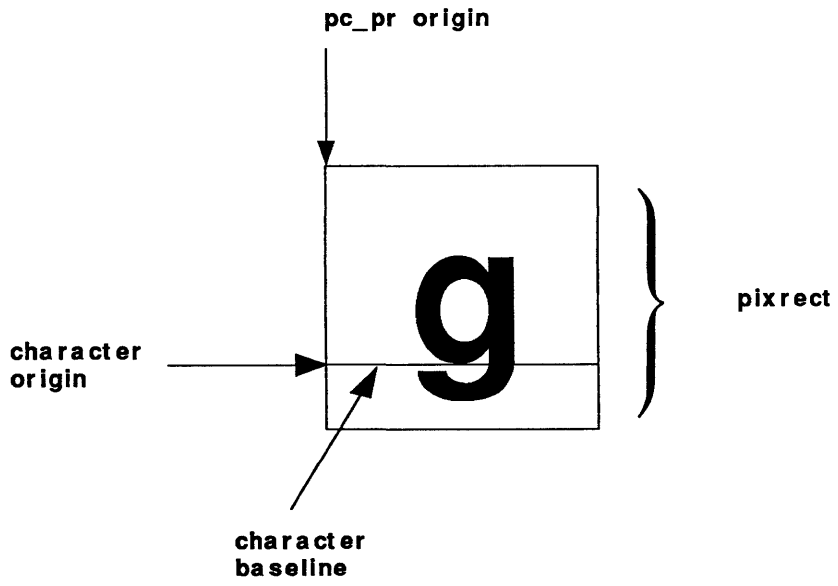
The leftmost point on a character is normally its origin, but *kerning* or mandatory letter spacing may move the origin right or left of that point. `pc_adv` is the amount the destination position is changed by this character; that is, the amounts in `pc_adv` added to the current character origin will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

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

The `Pixfont` structure contains an array of `pixchars`, indexed by the character code; it also contains the size (in pixels) of its characters when they are all the same. If the size of a font's characters varies in one dimension, that value in `pf_defaultsize` will not have anything useful in it; however, the other may

still be useful. Thus, for non-rotated variable-pitch fonts, `pf_defaultsize.y` will still indicate the unleaded interline spacing for that font.

Figure 4-1 *Character and* `pc_pr` *Origins*



4.2. Operations on Pixfonts

The commands listed below allow you to load a font to display. A font must be loaded before using a text operation.

Load a Font

```
Pixfont *pf_open(name)
char *name;
```

`pf_open()` returns a pointer to a *shared* copy of a font in virtual memory. A NULL is returned if the font cannot be opened. The path name of the font file should be specified, for example:

```
myfont = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.r.7");
```

`name` should be in the format described in *vfont*(5): the file is converted to pixfont format, allocating memory for its associated structures and reading in the data for it from disk. The utility `fontedit`(1) is a font editor for designing pixel fonts in *vfont*(5) format.

The data from a small selection of commonly used fonts is compiled into the pixrect library. The names of these built-in fonts are checked against the last component of the name. To guarantee that the font is loaded from the disk file

instead, use `pf_open_private()` instead of `pf_open()`.

Load Private Copy of Font

```
Pixfont *pf_open_private(name)
char *name;
```

`pf_open()` returns a pointer to a *private* copy of a font in virtual memory. A NULL is returned if the font cannot be opened.

Default Fonts

```
Pixfont *pf_default()
```

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

Close Font

```
pf_close(pf)
Pixfont *pf;
```

When a client is finished with a font, it should call `pf_close()` to free the memory associated with it. `pf` should be a font handle returned by a previous call to `pf_open()` or `pf_default()`.

4.3. Text Functions

The following functions manage various tasks involved in displaying text.

Pixrect Text Display

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

Characters are written into a pixrect with the `pf_text()` procedure. `where` is the destination for the start of the text (nominal left edge, baseline; see Section 4.1) `op` is the raster operation to be used in writing the text, as described in Section 3.3, *The Op Argument*; `font` is a pointer to the font in which the text is to be displayed; and `text` is the actual null-terminated string to be displayed. The color specified in the `op` specifies the color of the ink. The background of the text is painted 0 (background color).

Transparent Text

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

`pf_ttext` paints "transparent" text: it doesn't disturb destination pixels in blank areas of the character's image. The arguments to this procedure are the same as for `pf_text()`. The characters' bitmaps are used as a stencil, and the color specified in `op` is painted through the stencil.

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

Auxiliary Pixfont Procedures

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

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

`pf_textbatch()` is used internally by `pf_text()`; it constructs an array of `pr_pos` structures and records its length, as required by `batchrop` (see Section 3.6). `where` should be the address of the array to be filled in, and `lengthp` should point to a maximum length for that array. `text` addresses the null-terminated string to be put in the batch, and `font` refers to the `Pixfont` to be used to display it. When the function returns, `lengthp` will refer to a word containing the number of `pr_pos` structures actually used for `text`. The `pr_size` returned is the sum of the `pc_adv` fields in their `pixchar` structures.

`pf_textwidth()` returns a `pr_size` that is computed by taking the product of `len`, is the number of characters, and `pc_adv`, the width of each character.

Text Bounding Box

```
pf_textbound(bound, len, font, text)
struct pr_subregion *bound;
int len;
Pixfont *font;
char *text;
```

`pf_textbound` may be used to find the bounding box for a string of characters in a given font. `bound->pos` is the top-left corner of the bounding box, `bound->size.x` is the width, and `bound->size.y` is the height. `bound->pr` is not modified. `bound->pos` is computed relative to the location of the character origin (base point) of the first character in the text.

Unstructured Text

```
pr_text(pr, x, y, op, font, text)
Pixrect *pr;
int x, y, op;
Pixfont *font;
char *text;

pr_ttext(pr, x, y, op, font, text)
Pixrect *pr;
int x, y, op;
Pixfont *font;
char *text;
```

These unstructured text functions correspond to the *Pixwin* functions `pw_text()` and `pw_ttext()`. `prs_text()` and `prs_ttext()` macros are also provided, although they are identical to `pf_text()` and `pf_ttext()`, respectively.

4.4. Example

Here is an example program that writes text on the display surface with pixel fonts.

Figure 4-2 *Example Program using Text*

```
#include <pixrect/pixrect_hs.h>

main()
{
    Pixrect *pr;
    Pixfont *pf;

    if (!(pr = pr_open("/dev/fb")) ||
        !(pf = pf_open("/usr/lib/fonts/fixedwidthfonts/screen.r.12")))
        exit(1);

    pr_text(pr, 400, 400, PIX_SET, pf, "This is a string.");

    pr_close(pr);
    pf_close(pf);
    exit(0);
}
```

5

Memory Pixrects

5

Memory Pixrects

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

5.1. The `mpr_data` Structure

```
struct mpr_data {
    int md_linebytes;
    short *md_image;
    struct pr_pos md_offset;
    short md_primary;
    short md_flags;
};
#define MP_REVERSEVIDEO 1
#define MP_DISPLAY      2
#define MP_PLANEMASK    4
#define MP_I386         8   /* used only on Sun386i, */
#define MP_STATIC       16  /* ignored on all others. */
```

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

`linebytes` is the number of bytes stored in a row of the primary pixrect. This is the difference in the addresses between two pixels at the same *x*-coordinate, one row apart. Because a secondary pixrect may not include the full width of its primary pixrect, this quantity cannot be computed from the width of the pixrect — see Section 3.4. The actual pixels of a memory pixrect are stored someplace else in memory, usually an array, which `md_image` points to; the format of that area is described in the next section. The creator of the memory pixrect must ensure that `md_image` contains an even address. `md_offset` is the *x,y*

position of the first pixel of this pixrect in the array of pixels addressed by `md_image`. `md_primary` is 1 if the pixrect is primary and had its image allocated dynamically (e.g. by `mem_create()`). In this case, `md_image` will point to an area not referenced by any other primary pixrect. This flag is interrogated by the `pr_destroy()` routine: if it is 1 when that routine is called, the pixrect's image memory will be freed.

The MP_DISPLAY bit will be set in `md_flags` if the memory pixrect is actually a memory mapped frame buffer. The MP_REVERSEVIDEO bit will be set if reversevideo is currently in effect for the pixrect (this is only valid if the pixrect depth is 1 bit). The MP_386I bit is non-zero if the pixrect image data is in 80386 format.

NOTE *This flag is ignored on 680X0 based machines.*

The MP_STATIC is non-zero if the pixrect is static.

NOTE *This flag is ignored on 680X0 based machines.*

`md_flags` is present to support memory-mapped display devices like the Sun–2 monochrome video device, and the bit flipping necessary for Sun386i machines. See Chapter 2 for details on 80386 format, and the MP_386I and MP_STATIC flags.

Several useful macros are defined in `<pixrect/memvar.h>`. These macros will greatly increase the productivity of the programmer using memory pixrects, as well as the reliability of the code. Two commonly used macros are described here; see the others in `memvar.h`.

To access a memory pixrect's bitmap and functions, use the `mpr_d()` macro. It generates a pointer to the private data of a memory pixrect:

```
#define mpr_d(pr)    ((struct mpr_data *) (pr)->pr_data)
```

The `mpr_linebytes` macro computes the bytes per line of a primary memory pixrect given its width in pixels and the bits per pixel. This includes the padding to word bounds. It is useful for incrementing pixel addresses in the *y* direction, or calculating line padding in the bitmap.

```
#define mpr_linebytes(width, depth)
    ( ((pr_product(width, depth)+15)>>3) &~1)
```

Example

Here is an example program that uses a memory pixrect to do bit manipulations on the screen. It opens the frame buffer and copies the bitmap to a memory pixrect of the same size. It then goes through each byte of the memory pixrect, left-shifting each byte. Finally, it copies the modified memory pixrect back into the screen pixrect.

Note how the `mpr_linebytes` macro is used to find the number of bytes used to hold a line of the memory pixrect. The `mpr_d()` macro is also used to simplify access to the image area of the memory pixrect.

Figure 5-1 *Example Program using Memory Pixrects*

```
#include <pixrect/pixrect_hs.h>
#include <stdio.h>
main ()
{
    Pixrect *scrn, *mem;
    int ht,wid;
    char *start, *ptr;
    scrn = pr_open("/dev/fb");
    wid = scrn->pr_size.x;
    ht = scrn->pr_size.y;
    mem = mem_create(wid,ht,1);
    pr_rop(mem,0,0,wid,ht,PIX_SRC,scrn,0,0);
    start = (char *) mpr_d(mem)->md_image;
    for(ptr = start;ptr < start + mpr_linebytes(wid,1) * ht;ptr++) *ptr <<= 2;
    pr_rop(scrn,0,0,wid,ht,PIX_SRC,mem,0,0);
    pr_close(mem);
    pr_close(scrn);
}
```

5.2. Creating Memory Pixrects

The `mem_create()` and `mem_point()` functions allow a client program to create memory pixrects.

Create Memory Pixrect

```
Pixrect *mem_create(w, h, depth)
int w, h, depth;
```

A new primary pixrect is created by a call to the procedure `mem_create()`. `w`, `h` and `depth` specify the width and height in pixels, and depth in bits per pixel of the new pixrect. Sufficient memory to hold those pixels is allocated and cleared to 0, new `mpr_data` and `Pixrect` structures are allocated and initialized, and a pointer to the pixrect is returned. If this can not be done, the return value is NULL. On Sun386i systems, the memory pixrects created by `mem_create()` set the `MP_I386` flag to 1 (true).

On 32 bit systems (such as the Sun-3 and Sun-4) the created pixrect will have each scan line padded out to a 32 bit boundary, unless it is only 16 bits wide; that is, the `md_linebytes` structure member will contain either 2 or a multiple of 4. In older Sun releases pixrects created by `mem_create()` were always padded to a 16 bit boundary.

Create Memory Pixrect from an Image

```
Pixrect *mem_point(width, height, depth, data)
int width, height, depth;
short *data;
```

The `mem_point()` routine builds a pixrect structure that points to a dynamically created image in memory. Client programs may use this routine as an alternative to `mem_create()` if the image data is already in memory. `width` and `height` are the width and height of the new pixrect, in pixels. `depth` is the depth of the new pixrect, in number of bits per pixel. `data` points to the image

to be associated with the pixrect. Unlike the `mem_create()` routine, the `mem_point()` routine does **not** set the `MP_386I` flag; the pixrect remains in `680X0` format.

Note that `mem_point()` expects each line of the memory image to be padded to a 16 bit boundary. Also, `mem_point()` does not set the `md_primary` flag so the image will not be automatically freed when the pixrect is destroyed.

Example

Here is an example program which uses a memory pixrect to invert the frame buffer contents from top to bottom. It opens the default frame buffer and creates a memory pixrect of the same size. It then copies rows of pixels from the frame buffer to the memory pixrect in reverse order. Finally, it copies the memory pixrect back to the frame buffer.

Figure 5-2 *Example Program using Memory Pixrects*

```
#include <pixrect/pixrect_hs.h>

main()
{
    Pixrect *pr, *tmp;
    int yin, yout;

    if (!(pr = pr_open("/dev/fb")) ||
        !(tmp =
        mem_create(pr->pr_size.x, pr->pr_size.y, pr->pr_depth)))
        exit(1);

    for (yin = 0, yout = pr->pr_size.y - 1; yout >= 0; yin++, yout--)
        pr_rop(tmp, 0, yout, pr->pr_size.x, 1, PIX_SRC, pr, 0, yin);

    pr_rop(pr, 0, 0, pr->pr_size.x, pr->pr_size.y, PIX_SRC, tmp, 0, 0);

    exit(0);
}
```

5.3. Static Memory Pixrects

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

A memory pixrect may be created at compile time by using the `mpr_static()` macro. `name` is a token to identify the generated data objects; `w`, `h`, and `depth` are the width and height in pixels, and depth in bits of the pixrect; and `image` is the address of an even-byte aligned data object that contains the pixel values in the format described below, with each line padded to a 16 bit boundary.

If static structures are desired, the macro `mpr_static_static` should be used instead.

The macro generates two structures:

```
struct mpr_data name_data;
Pixrect name;
```

The `mpr_data` is initialized to point to all of the image data passed in; the `Pixrect` then refers to `mem_ops` and to `name_data`. On a Sun386i machine, the `MP_STATIC` flag will be set in the md_flags byte of the pixrect data structure; see Chapter 2 for details. *Note:* Contrary to its name, this macro generates structures of storage class `extern`.

5.4. Pixel Layout in Memory Pixrects

In memory, the upper-left corner pixel is stored at the lowest address. This address must be even. That first pixel is followed by the remaining pixels in the top row, left-to-right. Pixels are stored in successive bits without padding or alignment.

Each row of pixels is rounded to at least a 16 bit boundary. For best performance on 32 bit systems, pixel rows should be rounded to 32 bit boundaries (mem_create does this automatically). However, 16 bit rounding is required for static pixrects and mem_point.

Memory pixrects with depths of 1, 8, 16, 24, and 32 bits are currently supported by the pixrect library. If source and destination are both memory pixrects they must have an equal number of bits per pixel.

NOTE *If you are running a Sun386i machine. A pixrect's image data will be converted to 80386 format before being displayed. See Chapter 2 for details.*

5.5. Using Memory Pixrects

Memory pixrects can be used to get data from and send data to the display device. Several routines exist for interfacing *Pixwins* with memory pixrects. These include `pw_read()`, `pw_rop()` and `pw_write()`. Refer to the *SunView 1 Programmer's Guide* for more details. For applications using the raw device without *SunView*, `pr_rop()` can be used for operations on memory pixrects.

Another use of memory pixrects is for processing images that not intended for display. User programs can write directly into a pixrect using parameters found in the `mpr_data` structure, or they can use `mem_point()` for a previously created image. Memory pixrects can also be written to raster files using the facilities described in Chapter 6.

6

File I/O Facilities for Pixrects

6

File I/O Facilities for Pixrects

Sun Microsystems, Inc. has specified a file format for files containing raster images. The format is defined in the header file <rasterfile.h>. The pixrect library contains routines to perform I/O operations between pixrects and files in this raster file format. This I/O is done using the routines of the C Library Standard I/O package, requiring the caller to include the header file <stdio.h>.

The raster file format allows multiple types of raster images. Unencoded, and run-length encoded formats are supported directly by the pixrect library. Support for customer defined formats is implemented by passing raster files with non-standard types through filter programs. Sun supplied filters are found in the directory /usr/lib/rasfilters. This directory also includes sample source code for a filter that corresponds to one of the standard raster file types to facilitate writing new filters.

6.1. Writing and Reading Raster Files

The sections that follow describe how to store and retrieve an image in a rasterfile.

Run Length Encoding

The run-length encoding used in raster files is of the form

```
<byte><byte>...<ESC><0>...<byte><ESC><count><byte>...
```

where the counts are in the range 0..255 and the actual number of instances of <byte> is <count>+1 (i.e. actual is 1..256). One- or two-character sequences are left unencoded; three-or-more character sequences are encoded as <ESC><count><byte>. <ESC> is the character code 128. Each single <ESC> in the input data stream is encoded as <ESC><0>, because the <count> in this scheme can never be 0 (the actual count can never be 1). <ESC><ESC> is encoded as <ESC><1><ESC>.

This algorithm will fail (make the compressed data bigger than the original data) only if the input stream contains an excessive number of one- and two-character sequences of the <ESC> character.

Write Raster File

```
int pr_dump(input_pr, output, colormap, type, copy_flag)
Pixrect *input_pr;
FILE *output;
colormap_t *colormap;
int type, copy_flag;
```

The `pr_dump()` procedure stores the image described by a pixrect onto a file. It normally returns 0, but if any error occurs it returns PIX_ERR. The caller can write a rectangular sub-region of a pixrect by first creating an appropriate `input_pr` via a call to `pr_region()`. The output file is specified via `output`. The specified output type should either be one of the following standard types or correspond to a customer provided filter.

```
#define RT_OLD      0
#define RT_STANDARD 1
#define RT_BYTE_ENCODED 2
```

The RT_STANDARD type is the common raster file format in the same sense that memory pixrects are the common pixrect format: every raster file filter is required to read and write this format. The RT_OLD type is very close to the RT_STANDARD type; it was the former standard generated by old versions of Sun software. The RT_BYTE_ENCODED type implements a run-length encoding of bytes of the pixrect image. This usually results in shorter files, although pathological images may expand by 50%.

Specifying any other output type causes `pr_dump()` to pipe a raster file of RT_STANDARD type to the filter named `convert.`*type*, looking first in directories in the user's `$PATH` environment variable, and then in the directory `/usr/lib/rasfilters`. `type` is the ASCII corresponding to the specified `type` in decimal. The output of the filter is then copied to `output`.

It is strongly recommended that customer-defined formats use a *type* value of 100 or more, to avoid conflicts with additions to the set of standard types. The RT_EXPERIMENTAL type is reserved for use in the development of experimental filters, although it is no longer treated specially.

```
#define RT_EXPERIMENTAL 65535
```

`pr_dump()` and other functions that start filters wait until the filter process exits before returning, so caution is advisable when working with experimental filters.

For pixrects displayed on devices with colormaps, the values of the pixels are not sufficient to recreate the displayed image. Thus, the image's colormap can also be specified in the call to `pr_dump()`. If the `colormap` is specified as NULL but `input_pr` is a non-monochrome display pixrect, `pr_dump()` will attempt to write the colormap obtained from `input_pr` (via `pr_getcolormap`). The following structure is used to specify the colormap associated with `input_pr`:

```
typedef struct {
    int type;
    int length;
    unsigned char *map[3];
} colormap_t;
```

The colormap type should be one of the Sun supported types:

```
#define RMT_NONE 0
#define RMT_EQUAL_RGB 1
#define RMT_RAW 2
```

If the colormap type is RMT_NONE, then the colormap length must be 0. This case usually arises when dealing with monochrome displays and 1-bit deep memory pixrects. If the colormap type is RMT_EQUAL_RGB, then the map array should specify the red (`map[0]`), green (`map[1]`) and blue (`map[2]`) colormap values, with each vector in the map array being of the same specified colormap length. If the colormap type is RMT_RAW, the first map array (`map[0]`), should hold `length` bytes of colormap data, which will not be interpreted by the pixrect library.

Finally, `copy_flag` specifies whether or not `input_pr` should be copied to a temporary pixrect before the image is output. The `copy_flag` value should be non-zero if `input_pr` is a pixrect in a frame buffer that is likely to be asynchronously modified. The copy flag is also automatically set non-zero for secondary pixrects, to simplify the code. Note that use of `copy_flag` still will not guarantee that the correct image will be output unless the `pr_rop()` to copy from the frame buffer is made uninterruptible.

Figure 6-1 *Example Program using* `pr_dump()`

```
#include <stdio.h>
#include <sys/types.h>
#include <pixrect/pixrect.h>
#include <pixrect/pr_io.h>

main()
{
    Pixrect *screen, *icon;
    FILE *output = stdout;
    colormap_t *colormap = 0;
    int type = RT_STANDARD;
    int copy_flag = 1;

    if (!(screen = pr_open("/dev/fb")) ||
        !(icon = pr_region(screen, 1050, 10, 64, 64)))
        exit(1);

    pr_dump(icon, output, colormap, type, copy_flag);
    pr_close(screen);

    exit(0);
}
```

Read Raster File

```
Pixrect *pr_load(input, colormap)
FILE *input;
colormap_t *colormap;
```

The `pr_load()` function can be used to retrieve the image stored in a raster file into a pixrect. The raster file's header is read from `input`, a pixrect of the appropriate size is dynamically allocated, the colormap is read and placed in the location addressed by `colormap`, and finally the image is read into the pixrect and the pixrect returned. If any problems occurs, `pr_load()` returns NULL.

As with `pr_dump()`, if the specified raster file is not of standard type, `pr_load()` first runs the file through the appropriate filter to convert it to RT_STANDARD type and then loads the output of the filter.

Additionally, if `colormap` is NULL, `pr_load()` will simply discard any and all colormap information contained in the specified input raster file. If `colormap` is non-null `pr_load()` will load the colormap data even if the type and length specified do not match that of the file (see `pr_load_colormap()` below).

Figure 6-2 *Example Program using* `pr_load()`

```
#include <stdio.h>
#include <sys/types.h>
#include <pixrect/pixrect.h>
#include <pixrect/pr_io.h>

main()
{
    Pixrect *screen, *icon;
    FILE *input = stdin;
    colormap_t colormap;

    colormap.type = RMT_NONE;

    if (!(screen = pr_open("/dev/fb")) ||
        !(icon = pr_load(input, &colormap)))
        exit(1);

    if (colormap.type == RMT_EQUAL_RGB)
        pr_putcolormap(screen, 0, colormap.length,
            colormap.map[0], colormap.map[1],
            colormap.map[2]);

    pr_rop(screen, 1050, 110, 64, 64, PIX_SRC, icon, 0, 0);
    pr_close(screen);

    exit(0);
}
```

6.2. Details of the Raster File Format

A handful of additional routines are available in the pixrect library for manipulating pieces of raster files. In order to understand what they do, it is necessary to understand the exact layout of the raster file format.

The raster file is in three parts: first, a small header containing eight 32-bit `int`'s; second, a (possibly empty) set of colormap values; third, the pixel image, stored a line at a time, in increasing `y` order.

The image is essentially laid out in the file the exact way that it would appear in a static memory pixrect. In particular, each line of the image is rounded out to a multiple of 16 bits, corresponding to the rounding convention used by static pixrects.

The header is defined by the following structure:

```
struct rasterfile {
    int ras_magic;
    int ras_width;
    int ras_height;
    int ras_depth;
    int ras_length;
    int ras_type;
    int ras_maptype;
    int ras_maplength;
};
```

The `ras_magic` field always contains the following constant:

```
#define RAS_MAGIC 0x59a66a95
```

The `ras_width`, `ras_height` and `ras_depth` fields contain the image's width and height in pixels, and its depth in bits per pixel, respectively. The depth is usually either 1 or 8, corresponding to the standard frame buffer depths.

The `ras_length` field contains the length in bytes of the image data. For an unencoded image, this number is computable from the `ras_width`, `ras_height`, and `ras_depth` fields, but for an encoded image it must be explicitly stored in order to be available without decoding the image itself. Note that the length of the header and of the possibly empty colormap values are not included in the value in the `ras_length` field; it is only the image data length. For historical reasons, files of type RT_OLD will usually have a 0 in the `ras_length` field, and software expecting to encounter such files should be prepared to compute the actual image data length if it is needed. The `ras_maptype` and `ras_maplength` fields contain the type and length in bytes of the colormap values, respectively.

If the `ras_maptype` is not RMT_NONE and the `ras_maplength` is not 0, then the colormap values are the `ras_maplength` bytes immediately after the header. These values are either uninterpreted bytes (usually with the `ras_maptype` set to RMT_RAW) or the equal length red, green and blue vectors, in that order (when the `ras_maptype` is RMT_EQUAL_RGB). In the latter case, the `ras_maplength` must be three times the size in bytes of any one of the vectors.

6.3. Writing Parts of a Raster File

The following routines are available for writing the various parts of a raster file. Many of these routines are used to implement `pr_dump()`. First, the raster file header and the colormap can be written by calling `pr_dump_header()`.

Write Header to Raster File

```
int pr_dump_header(output, rh, colormap)
FILE *output;
struct rasterfile *rh;
colormap_t *colormap;
```

`pr_dump_header()` returns PIX_ERR if there is a problem writing the header or the colormap, otherwise it returns 0. If the colormap is NULL, no colormap

values are written.

Initialize Raster File Header

```
Pixrect *pr_dump_init(input_pr, rh, colormap,
    type, copy_flag)
Pixrect *input_pr;
struct rasterfile *rh;
colormap_t *colormap;
int type, copy_flag;
```

For clients that do not want to explicitly initialize the rasterfile struct this routine can be used to set up the arguments for `pr_dump_header()`. The arguments to `pr_dump_init()` correspond to the arguments to `pr_dump()`. However, `pr_dump_init()` returns the pixrect to write, rather than actually writing it, and initializes the structure pointed to by `rh` rather than writing it. If colormap is NULL, the `ras_maptype` and `ras_maplength` fields of `rh` will be set to RMT_NONE and 0, respectively.

If any error is detected by `pr_dump_init()`, the returned pixrect is NULL. If there is no error, the `copy_flag` is zero, and the input pixrect is suitable for direct dumping (a primary memory pixrect), the returned pixrect is simply `input_pr`. However, if `copy_flag` is non-zero, or the input pixrect cannot be dumped directly, the returned pixrect is dynamically allocated and the caller is responsible for deallocating it with `pr_destroy()` when it is no longer needed.

Write Image Data to Raster File

```
int pr_dump_image(pr, output, rh)
Pixrect *pr;
FILE *output;
struct rasterfile *rh;
```

The actual image data can be output via a call to `pr_dump_image()`. This routine returns 0 unless there is an error, in which case it is PIX_ERR. It cannot write the image in a non-standard (filtered) format, since by the time it is called the raster file header has already been written.

Since these routines sequentially advance the output file's write pointer, `pr_dump_image()` must be called after `pr_dump_header()`.

6.4. Reading Parts of a Raster File

The following routines are available for reading the various parts of a raster file. Many of these routines are used to implement `pr_load()`. Since these routines sequentially advance the input file's read pointer, rather than doing random seeks in the input file, they should be called in the order presented below.

Read Header from Raster File

```
int pr_load_header(input, rh)
FILE *input;
struct rasterfile *rh;
```

The raster file header can be read by calling `pr_load_header()`. This routine reads the header from the specified input, checks it for validity and initializes the specified `rasterfile` structure from the header. The return value is 0 unless there is an error, in which case it is PIX_ERR.

Read Colormap from Raster File

```
int pr_load_colormap(input, rh, colormap)
FILE *input;
struct rasterfile *rh;
colormap_t *colormap;
```

If the header indicates that there is a non-empty set of colormap values, they can be read by calling `pr_load_colormap()`. If the specified colormap is NULL, this routine will skip over the colormap values by reading and discarding them. If the type and length values in `colormap` do not match the input file, `pr_load_colormap()` will allocate space for the colormap with `malloc`, read the colormap, and modify `colormap` before returning. If this occurs, the space allocated can be released with a `free(colormap->map[0])`.

The return value is 0 unless there is an error, in which case it is PIX_ERR.

Read Image from Raster File

```
Pixrect *pr_load_image(input, rh, colormap)
FILE *input;
struct rasterfile *rh;
colormap_t *colormap;
```

An image can be read by calling `pr_load_image()`. If the input is a standard raster file type, this routine reads in the image directly. Otherwise, it writes the header, colormap, and image into the appropriate filter and then reads the output of the filter. In this case, both the rasterfile and the colormap structures will be modified as a side-effect of calling this routine. In either case, a pixrect is dynamically allocated to contain the image, the image is read into the pixrect, and the pixrect is returned as the result of calling the routine. If there is an error, the return value is NULL.

Read Standard Raster File

```
Pixrect *pr_load_std_image(input, rh, colormap)
FILE *input;
struct rasterfile *rh;
colormap_t colormap;
```

If it is known that the image is from a standard raster file type, then it can be read in by calling `pr_load_std_image()`. This routine is identical to `pr_load_image()`, except that it will not invoke a filter on non-standard raster file types.

A

Writing a Pixrect Driver

A

Writing a Pixrect Driver

Sun has defined a common programming interface to pixel addressable devices that enables, in particular, device independent access to all Sun frame buffers. This interface is called the `pixrect` interface. Existing Sun supported software systems access a frame buffer through the pixrect interface. Sun encourages customers with other types of frame buffers (or other types of pixel addressable devices) to provide a pixrect interface to these devices.

This chapter describes how to write a pixrect driver. It is assumed that you have already read Chapter 3, *Pixrect Operations*; it describes the programming interface to the basic operations that must be provided in order to generate a complete pixrect implementation. It is also assumed that you have a copy of *Writing Device Drivers* The section in that manual on writing the kernel device driver portion of the pixrect implementation is important.

This chapter contains auxiliary material of interest only to pixrect driver implementors, not programmers accessing the pixrect interface. This document explains how to install a new pixrect driver into the software architecture so that it may be used in a device independent manner. Also, utilities and conventions that may be of use to the pixrect driver implementor are discussed.

This chapter walks through the source code for the *CG-1* pixrect driver. The CG-1 is the Sun-1 color frame buffer. Using this particular driver as an example has no significance; another pixrect driver would have worked just as well.

The actual source code that is presented here is boiler-plate, i.e., almost every pixrect driver implementation will be similar. You should be able to make your own driver just from the documentation alone. However, a complete source example for an existing pixrect driver would probably expedite the development of your own driver. The complete device specific source files for the Sun-1 color frame buffer pixrect driver is available as a source code purchase option (available without a UNIX source license).

A.1. What You'll Need

These are the tools and pieces that you'll need before assembling your pixrect driver:

- You need the correct documentation:

  SunView 1 Programmer's Guide

SunView 1 System Programmer's Guide

Writing Device Drivers

- You need to know how to drive the hardware of your pixel addressable device. At a minimum, a pixel addressable device must have the ability to read and write single pixel values. (One could imagine a device that doesn't even meet the minimum requirements being used as a pixel addressable device. We will not discuss any of the ways that such a device might fake the minimum requirements).
- You must have a UNIX kernel building environment. No extra source is required.
- You must have the current pixrect library file and its accompanying header files. No extra source is required.

A.2. Implementation Strategy

This is one possible step-by-step approach to implementing a pixrect driver:

- Write and debug pixrect creation and destruction. This involves the pixrect kernel device driver that lets you `open`(2) and `mmap`(2) the physical device from a user process. The private `cg1_make` routine must be written. The `cg1_region` and `cg1_destroy` pixrect operation must be written.
- Write and debug the basic pixel rectangular region function. The `cg1_putattributes` and `cg1_putcolormap` pixrect operations must be written in addition to the `cg1_rop` routine.
- Write and debug batchrop routines. The `cg1_batchrop` pixrect operation must be written.
- Write and debug vector drawer. The `cg1_vector` pixrect operation must be written.
- Write and debug remaining pixrect operations: `cg1_stencil`, `cg1_get`, `cg1_put`, `cg1_getattributes` and `cg1_getcolormap`.
- If the device is to run with *SunView*, build a kernel with minimal basic pixel rectangle function for use by the cursor tracking mechanism in the *SunView* kernel device driver. Also include the colormap access routines for use by the colormap segmentation mechanism in the *SunView* kernel device driver.
- Load and test *SunView* programs with new pixrect driver. Experience has shown that when you are able to run released *SunView* programs that your pixrect driver is in pretty good shape.

A.3. Files Generated

Here is the list of source files generated that implement the example pixrect driver:

- `cg1reg.h` - A header file describing the structure of the raster device. It contains macros used to address the raw device.
- `cg1var.h` - A header file describing the private data of the pixrect. It contains external references to pixrect operation of this driver.

- `/sys/sundev/cgone.c` - The pixrect kernel device driver code.
- `cg1.c` - The pixrect creation and destruction routines.
- `cg1_region.c` - The region creation routine.
- `pr_makefun.c` - Replaces an existing module and contains the vector of pixrect make operations.
- `cg1_batch.c` - The batchrop routine.
- `cg1_colormap.c` - The colormap access and attribute setting routines .
- `cg1_getput.c` - The single pixel access routines.
- `cg1_rop.c` - The basic pixel rectangle manipulation routine.
- `cg1_stencil.c` - The stencil routine.
- `cg1_vec.c` - The vector drawer.
- `cg1_curve.c` The curved shape routine.
- `cg1_polyline.c` The polyline routine.

Memory Mapped Devices

Some devices are memory mapped; a good example is the *bw2*, the Sun-2 monochrome video frame buffer. With such devices, their pixels are manipulated directly as main memory; there are no device specific registers through which the pixels are accessed. Memory mapped devices are able to rely on the memory pixrect driver for many of its operations. The only files that the Sun 2 monochrome video frame buffer supplies are:

- `bw2var.h` - A header file describing the private data of the pixrect. It contains external references to pixrect operation of this driver.
- `/sys/sundev/bwtwo.c` - The pixrect kernel device driver code.
- `bw2.c` - The pixrect creation and destruction routines.

The operations vector for the Sun 2 monochrome pixrect driver is:

```
struct  pixrectops bw2_ops = {
    mem_rop, mem_stencil, mem_batchrop,
    0, bw2_destroy, mem_get, mem_put, mem_vector,
    mem_region, mem_putcolormap, mem_getcolormap,
    mem_putattributes, mem_getattributes
};
```

A.4. Pixrect Private Data

Each pixrect device must have a private data object that contains instance specific data about the state of the driver. It is not acceptable to have global data shared among all the pixrects objects. The device specific portion of the pixrect data must contain certain information:

- An offset from the upper left-hand corner of the pixel device. This offset, plus the width and height of the pixrect from the public portion, is used to determine the clipping rectangle used during pixrect operations.

- A flag for distinguishing between primary and secondary pixrects. Primary pixrects are the owners of dynamically allocated resources shared between primary and secondary pixrects.
- A file descriptor to the pixrect kernel device. Usually, the file descriptor is used while mapping pages into the user process address space so that the device may be addressed. One could imagine a pixrect driver that had some of its pixrect operations implemented inside the kernel. The file descriptor would then be the key to communicating with that portion of the package via `read`(2), `write`(2) and `ioctl`(2) system calls.

Here is other possible data maintained in the pixrect's private data:

- For many devices, a virtual address pointer is part of the private data so that the device can be accessed from user code.
- For color devices, there is a mask to enable access to specific bit planes.
- For monochrome devices, there is a video invert flag. This replaces the colormap of color devices.

A.5. Creation and Destruction

This section covers the code for pixrect object creation and destruction. Code for the Sun-1 color frame buffer pixrect driver is presented as an example.

There are two public pathways to creating a pixrect:

- `pr_open()` creates a primary pixrect.
- `pr_region()` creates a secondary pixrect which specifies a subregion in an existing pixrect.

There are two public pathways to destroying a pixrect:

- `pr_destroy()` destroys a primary or secondary pixrect. Clients of the pixrect interface are responsible for destroying all extant secondary pixrects before destroying the primary pixrect from which they were derived.
- `pr_close()` simply calls `pr_destroy()`.

Creating a Primary Pixrect

In this section, the private `cg1_make` pixrect operation is described. This is the flow of control for `pr_open()`:

- Higher levels of software call `pr_open()`, which takes a device file name (e.g., `/dev/cgone0`).
- `pr_open()` opens the device and finds out its type and size via an FBIOGTYPE `ioctl`(2) call (see `<sun/fbio.h>`).
- `pr_open()` uses the type of pixel addressable device to index into the `pr_makefun` array of procedures (more on this later) and calls the referenced pixrect make function, `cg1_make`.
- `cg1_make` returns the primary pixrect (it workings are discussed below).
- `pr_open()` closes its handle on the device and the pixrect is returned.

Here is a partial listing of `cg1.c` that contains code that is important to the `cg1_make` procedure. As it is for other code presented in this document, it is here so you can refer back to it as you read the subsequent explanation. Some lines are numbered for reference and normal C comments have been removed in favor of the accompanying text.

```
#include <sys/types.h>
#include <stdio.h>
#include <pixrect/pixrect.h>
#include <pixrect/pr_util.h>
#include <pixrect/cg1reg.h>
#include <pixrect/cg1var.h>

static  struct pr_devdata *cg1devdata;  /* cg1.1*/

struct  pixrectops cg1_ops = {  /* cg1.2*/
    cg1_rop, cg1_stencil, cg1_batchrop, 0, cg1_destroy, cg1_get,
    cg1_put, cg1_vector, cg1_region, cg1_putcolormap, cg1_getcolormap,
    cg1_putattributes, cg1_getattributes,
};

struct pixrect *
cg1_make(fd, size, depth)   /* cg1.3*/
    int fd;     /* cg1.4*/
    struct  pr_size size;
    int depth;
{
    struct pixrect *pr;
    register struct cg1pr *cgpr;    /* cg1.5*/
    struct  pr_devdata *dd; /* cg1.6*/

    if (depth != CG1_DEPTH || size.x != CG1_WIDTH
            || size.y != CG1_HEIGHT) {  /* cg1.7*/
        fprintf(stderr, "cg1_make sizes wrong %D %D %D\n",
            depth, size.x, size.y);
        return (0);
    }
    if (!(pr = pr_makefromfd(fd, size, depth, &cg1devdata, &dd,/* cg1.8*/
        sizeof(struct cg1fb), sizeof(struct cg1pr), 0)))
        return (0);
    pr->pr_ops = &cg1_ops;          /* cg1.9*/
    cgpr = (struct cg1pr *)pr->pr_data; /* cg1.10*/
    cgpr->cgpr_fd = dd->fd;         /* cg1.11*/
    cgpr->cgpr_va = (struct cg1fb *)dd->va; /* cg1.12*/
    cgpr->cgpr_planes = 255;        /* cg1.13*/
    cgpr->cgpr_offset.x = cgpr->cgpr_offset.y = 0;  /* cg1.14*/
    cg1_setreg(cgpr->cgpr_va, CG_STATUS, CG_VIDEOENABLE);   /* cg1.15*/
    return (pr);        /* cg1.16*/
}
```

Line *cg1.7* does some consistency checking to make sure that the dimensions of the pixel addressable device and the client's idea about the dimensions of the device match.

```
struct *pixrect pr_makefromfd(fd, size, depth, devdata,
    curdd, mmapbytes, privdatabytes, mmapoffsetbytes)
    struct  pr_size size;
    struct  pr_devdata **devdata, **curdd;
    int     fd, depth, mmapbytes, privdatabytes,
            mmapoffsetbytes;
```

Line *cg1.8* calls the pixrect library routine `pr_makefromfd` to do most of the work:

- Allocates a `pixrect` structure object using the `calloc` library call. The pixrect is filled in with *size* and *depth* parameters.
- Allocates an object of the size *privdatabytes* using the `calloc` library call and placing a pointer to it in the *pr_data* field of the allocated pixrect.
- `dup`(2)s the passed in file descriptor *fd* so that when the caller closes the file descriptor the device wouldn't close.
- `mmap`(2) allocates and maps to the device *mmapbytes* of space.
- If an error is detected during any of the above calls, an error is written to the standard error output. A NULL pixrect handle is returned in this case.
- Returns the allocated pixrect.

This brings us to the issue of minimizing resources used by the pixrect driver. and`pr_open`, `cg1_make`, can be (and are) called many times thus creating a situation in which there are many primary pixrects open at a time. A pixrect should maintain an open file descriptor and (usually) a non-trivial amount of virtual address space mapped into the user process's address space. Both the number of open file descriptors and the virtual address space (maximum 16 megabytes) are finite resources. However, multiple open pixrects can share all these resources.

The pixrect library supports a resource sharing mechanism, part of which is implemented in `pr_makefromfd`. The `devdata` parameter passed to `pr_makefromfd` is the head of a linked list of `pr_devdata` structures of which there is one per pixrect driver. It is sufficient to say that through the data maintained on this list, sharing of the scarce resources described above can be accomplished.

The `curdd` parameter passed to `pr_makefromfd` is set to be the `pr_devdata` structure that applies to the device identified by `fd`.

Lines *cg1.9* through *cg1.14* are concerned with initializing the pixrect's private data with dynamic information described in `dd` (`curdd` in the previous paragraph) and static information about the pixel addressable device.

Line *cg1.15* is where the video signal for the device is enabled. By convention, every raster device should make sure that it is enabled.

Creating a Secondary Pixrect

In this section, the `cg1_region` pixrect operation is described. Here is all of `cg1_region.c`.

```
struct pixrect *cg1_region(src)
    struct pr_subregion src;
{
    register struct pixrect *pr;
    register struct cg1pr *scgpr = cg1_d(src.pr), *cgpr;
    int zero = 0;

    pr_clip(&src, &zero);   /* cg1_region.1*/
    if ((pr = (struct pixrect *)calloc(1, sizeof (struct pixrect))) == 0)
        return (0); /* cg1_region.2*/

    if ((cgpr = (struct cg1pr *)calloc(1, sizeof (struct cg1pr))) == 0) {
        free(pr);   /* cg1_region.3*/
        return (0);
    }
    pr->pr_ops = &cg1_ops;  /* cg1_region.4*/
    pr->pr_size = src.size; /* cg1_region.5*/
    pr->pr_depth = CG1_DEPTH;   /* cg1_region.6*/
    pr->pr_data = (caddr_t)cgpr;    /* cg1_region.7*/
    cgpr->cgpr_fd = -1; /* cg1_region.8*/
    cgpr->cgpr_va = scgpr->cgpr_va; /* cg1_region.9*/
    cgpr->cgpr_planes = scgpr->cgpr_planes; /*cg1_region.10*/
    cgpr->cgpr_offset.x = scgpr->cgpr_offset.x + src.pos.x; /*cg1_region.11*/
    cgpr->cgpr_offset.y = scgpr->cgpr_offset.y + src.pos.y; /*cg1_region.12*/
    return (pr);
}
```

`cg1_region` is less complex then `cg1_make`. The first thing done is to clip the requested subregion to fall within the source pixrect (line *cg1_region.1*).

```
pr_clip(dstp, srcp)
    struct  pr_subregion *dstp;
    struct  pr_prpos *srcp;
```

`pr_clip` adjusts the position and size of `dstp`, the destination pixrect subregion, to fall within `dstp->pr`. If `*scrp`, the source pixrect position, is not zero then the position of the source is clipped to fall within `dstp`.

Next, objects are allocated for the pixrect and the pixel addressable device's private data (line *cg1_region.2* and *cg1_region.3*). Then, similarly to the later part of `cg1_make`, the two new data objects are initialized (lines *cg1_region.4* through *cg1_region.12*). One thing to note is that the cg1 driver uses a –1 in the file descriptor field of the pixrect's private data to indicate that this pixrect is secondary (line *cg1_region.8*).

Destroying a Pixrect

In this section, the `cg1_destroy` pixrect operation is described. It works on secondary and primary pixrects. Here is more of `cg1.c`.

```
cg1_destroy(pr)
    struct pixrect *pr;
{
    register struct cg1pr *cgpr;

    if (pr == 0)
        return (0);
    if (cgpr = cg1_d(pr)) { /*cg1.30*/
        if (cgpr->cgpr_fd != -1) {  /*cg1.31*/
            pr_unmakefromfd(cgpr->cgpr_fd, &cg1devdata);    /*cg1.32*/
        }
        free(cgpr); /*cg1.33*/
    }
    free(pr);       /*cg1.34*/
    return (0);
}
```

Note that dynamic memory is freed (lines *cg1.33* and *cg1.34*). Also, note that only a primary pixrect (as indicated by a file descriptor that is not –1) invokes a call to `pr_unmakefromfd` (line *cg1.32*).

```
pr_unmakefromfd(fd, devdata)
    struct  pr_devdata **devdata;
    int     fd;
```

This pixrect library routine is the counterpart of `pr_makefromfd()`. If the device identified by the file descriptor `fd` has no more pixrects associated with it (as determined from `devdata`) then the resources associated with it are released.

The `pr_makefun()` Operations Vector

As mentioned above, `pr_open()` calls `cg1_make()` through the `pr_makefun()` procedure vector. This is what `pr_makefun()` looks like (it is the sole contents of `pr_makefun.c`):

```
#include <pixrect/pixrect_hs.h>
#include <sun/fbio.h>

Pixrect *(*pr_makefun[FBTYPE_LASTPLUSONE])() = {
        bw1_make,
        cg1_make,
        bw2_make,
        cg2_make,
        gp1_make,
        0 /* bw3_make */ ,
        0 /* cg3_make */ ,
        0 /* bw4_make */ ,
        cg4_make
};
```

`pr_makefun()` is the routine that pulls in all the code from the different frame buffers. If a site is not going to use programs on more than one kind of display, the unused slots can be commented out to prevent the code for the unused display from being loaded. This has the advantage of reducing disk space usage. However, working set size will presumably not be affected due to virtual memory not touching unused code.

For both the case of adding and deleting drivers, loading a compiled version of this edited file will have the effect of ignoring the commented out device drivers.

When adding some new pixrect driver, you need to assign it some unused constant from `<sun/fbio.h>`, e.g., FBTYPE_NOTSUN1. This then becomes the device identifier for your new pixrect driver. You need to generate a new version of the source file `pr_makefun.c` with the above data structure except that the array entry `pr_makefun` [FBTYPE_NOTSUN1] would contain the pixrect make procedure for your FBTYPE_NOTSUN1 pixrect driver (line `pr_makefun.1`). The old `pr_makefun.o` in the pixrect library could be replaced with your new `pr_makefun.o` using `ar`(1).

A.6. Pixrect Kernel Device Driver

A pixrect kernel device driver supports the pixel addressable device as a complete UNIX device. It also supports use of this device by the *SunView* driver so that the cursor can be tracked and the colormap loaded within the kernel. The document *Writing Device Drivers for the Sun Workstation* contains the details of device driver construction. It also contains an overview.

The code in this section comes from `cgone.c`. In the kernel, suffixes that end with a number (like `cg1`) confuse the conventions surrounding device driver names. A number suffix refers to the minor device number of a device. Therefore, in our example, `cg1` becomes `cgone` where the naming has something to do with the pixrect kernel device driver.

Configurable Device Support

Raster devices typically hang off a high speed bus (e.g., Multibus) or are plugged into a high speed communications port. At kernel building time the UNIX auto-configuration mechanism is told what devices to expect and where they should be found. At boot time the auto-configuration mechanism checks to see if each of the devices it expects are present.

This section deals with the auto-configuration aspects of the driver. This driver is written in the conventional style that supports multiple units of the same device type. It is recommended that you follow this style even if you aren't anticipating multiple pixel addressable devices of your type on a single UNIX system.

```
#include "cgone.h"
#include "win.h"
#if NCGONE > 0
#include "../h/param.h"
#include "../h/systm.h"
#include "../h/dir.h"
#include "../h/user.h"
#include "../h/proc.h"
#include "../h/buf.h"
#include "../h/conf.h"
```

```
#include "../h/file.h"
#include "../h/uio.h"
#include "../h/ioctl.h"
#include "../machine/mmu.h"
#include "../machine/pte.h"
#include "../sun/fbio.h"
#include "../sundev/mbvar.h"
#include "../pixrect/pixrect.h"
#include "../pixrect/pr_util.h"
#include "../pixrect/cg1reg.h"
#include "../pixrect/cg1var.h"

#if NWIN > 0
#define CG1_OPS &cg1_ops
struct  pixrectops cg1_ops = {
    cg1_rop,
    cg1_putcolormap,
    cg1_putattributes,
};
#else
#define CG1_OPS (struct pixrectops *)0
#endif

#define CG1SIZE (sizeof (struct cg1fb))
struct  cg1pr cgoneprdatadefault =
    { 0, 0, 255, 0, 0 };
struct  pixrect cgonepixrectdefault =
    { CG1_OPS, { CG1_WIDTH, CG1_HEIGHT }, CG1_DEPTH, /* filled in later */ 0 };


/*
 * Driver information for auto-configuration stuff.
 */
int cgoneprobe(), cgoneintr();
struct  pixrect cgonepixrect[NCGONE];
struct  cg1pr cgoneprdata[NCGONE];
struct  mb_device *cgoneinfo[NCGONE];
struct  mb_driver cgonedriver = {
    cgoneprobe, 0, 0, 0, 0, cgoneintr,
    CG1SIZE, "cgone", cgoneinfo, 0, 0, 0,
};

/*
 * Only allow opens for writing or reading and writing
 * because reading is nonsensical.
 */
cgoneopen(dev, flag)
    dev_t dev;
{
    return(fbopen(dev, flag, NCGONE, cgoneinfo));
}

/*
```

```
 * When close driver destroy pixrect.
 */
/*ARGSUSED*/
cgoneclose(dev, flag)
    dev_t dev;
{
    register int unit = minor(dev);

    if ((caddr_t)&cgoneprdata[unit] == cgonepixrect[unit].pr_data) {
        bzero((caddr_t)&cgoneprdata[unit], sizeof (struct cg1pr));
        bzero((caddr_t)&cgonepixrect[unit], sizeof (struct pixrect));
    }
}

/*ARGSUSED*/
cgoneioctl(dev, cmd, data, flag)
    dev_t dev;
    caddr_t data;
{
    register int unit = minor(dev);

    switch (cmd) {

    case FBIOGTYPE: {
        register struct fbtype *fb = (struct fbtype *)data;

        fb->fb_type = FBTYPE_SUN1COLOR;
        fb->fb_height = 480;
        fb->fb_width = 640;
        fb->fb_depth = 8;
        fb->fb_cmsize = 256;
        fb->fb_size = 512*640;
        break;
        }
    case FBIOGPIXRECT: {
        register struct fbpixrect *fbpr = (struct fbpixrect *)data;
        register struct cg1fb *cg1fb =
            (struct cg1fb *)cgoneinfo[(unit)]->md_addr;

        /*
         * "Allocate" and initialize pixrect data with default.
         */
        fbpr->fbpr_pixrect = &cgonepixrect[unit];
        cgonepixrect[unit] = cgonepixrectdefault;
        fbpr->fbpr_pixrect->pr_data = (caddr_t) &cgoneprdata[unit];
        cgoneprdata[unit] = cgoneprdatadefault;
        /*
         * Fixup pixrect data.
         */
        cgoneprdata[unit].cgpr_va = cg1fb;
        /*
         * Enable video
         */
```

```
        cg1_setreg(cg1fb, CG_FUNCREG, CG_VIDEOENABLE);
        /*
         * Clear interrupt
         */
        cg1_intclear(cg1fb);
        break;
        }

    default:
        return (ENOTTY);
    }
    return (0);
}

/*
 * We need to handle vertical retrace interrupts here.
 * The color map(s) can only be loaded during vertical
 * retrace; we should put in ioctls for this to synchronize
 * with the interrupts.
 * FOR NOW, see comments in the code.
 */
cgoneintclear(cg1fb)
    struct  cg1fb *cg1fb;
{
    /*
     * The Sun-1 color frame buffer doesn't indicate that an
     * interrupt is pending on itself so we don't know if the interrupt
     * is for our device.  So, just turn off interrupts on the cgone board.
     * This routine can be called from any level.
     */
    cg1_intclear(cg1fb);
    /*
     * We return 0 so that if the interrupt is for some other device
     * then that device will have a chance at it.
     */
    return(0);
}

int
cgoneintr()
{
    return(fbintr(NCGONE, cgoneinfo, cgoneintclear));
}

/*ARGSUSED*/
cgonemmap(dev, off, prot)
    dev_t dev;
    off_t off;
    int prot;
{
    return(fbmmap(dev, off, prot, NCGONE, cgoneinfo, CG1SIZE));
}
```

```
#include "../sundev/cgreg.h"
    /*
     * Note: using old cgreg.h to peek and poke for now.
     */
/*
 * We determine that the thing we're addressing is a color
 * board by setting it up to invert the bits we write and then writing
 * and reading back DATA1, making sure to deal with FIFOs going and coming.
 */
#define DATA1 0x5C
#define DATA2 0x33
/*ARGSUSED*/
cgoneprobe(reg, unit)
    caddr_t reg;
    int unit;
{
    register caddr_t CGXBase;
    register u_char *xaddr, *yaddr;

    CGXBase = reg;
    if (pokec((caddr_t)GR_freg, GR_copy_invert))
        return (0);
    if (pokec((caddr_t)GR_mask, 0))
        return (0);
    xaddr = (u_char *)(CGXBase + GR_x_select + GR_update + GR_set0);
    yaddr = (u_char *)(CGXBase + GR_y_select + GR_set0);
    if (pokec((caddr_t)yaddr, 0))
        return (0);
    if (pokec((caddr_t)xaddr, DATA1))
        return (0);
    (void) peekc((caddr_t)xaddr);
    (void) pokec((caddr_t)xaddr, DATA2);
    if (peekc((caddr_t)xaddr) == (~DATA1 & 0xFF)) {
        /*
         * The Sun-1 color frame buffer doesn't indicate that an
         * interrupt is pending on itself.
         * Also, the interrupt level is user program changable.
         * Thus, the kernel never knows what level to expect an
         * interrupt on this device and doesn't know is an interrupt
         * is pending.
         * So, we add the cgoneintr routine to a list of interrupt
         * handlers that are called if no one handles an interrupt.
         * Add_default_intr screens out multiple calls with the same
         * interrupt procedure.
         */
        add_default_intr(cgoneintr);
        return (CG1SIZE);
    }
    return (0);
}

#endif
```

This is how the driver is plugged into the auto-configuration mechanism. `/etc/config` reads a line in the configuration file for a Sun-1 color frame buffer:

```
device          cgone0 at mb0 csr 0xec000 priority 3
```

An external reference to `cgonedriver` (line *cgone.4*) is made in a table maintained by the auto-configuration mechanism. At boot time, if the auto-configuration mechanism can resolve the reference to `cgonedriver` then the contents of this structure are used to configure in the device:

- `cgoneprobe` - The name of the probe procedure (line *cgone.5*).
- `cgoneintr` - The name of the interrupt procedure (line *cgone.6*).
- CG1SIZE - The size in bytes of the address space of the device.
- `cgone` - The prefix of the device. Used in status and error messages.
- `cgoneinfo` - The array of devices pointers of the driver's type (line *cgone.2*).
- The other field's defaults suffice for most pixel addressable devices.

`cgoneprobe` is called to let the driver decide if the virtual address at `reg` is indeed a device that this driver recognizes as one of its own. The `unit` argument is the minor device number of this device. Writing a good probe routine can be difficult. The trick is to use some idiosyncrasy of the device that differentiates it from others. The real driver for the Sun-1 color frame buffer determines that it is addressing a Sun-1 color frame buffer by setting it up to invert the data written to it and reading back the result. The details of this code are not important to this discussion and is not included. Zero is returned if the probe fails and CG1SIZE is returned if the probe succeeds.

`cgoneintr` is called when an interrupt is generated at the beginning of the vertical retrace. There are a variety of things that one might want to synchronize with such an interrupt, e.g., load the colormap or move the cursor. Currently, the utility `fbintr` simply disables the interrupt from happening again (line *cgone.6*).

```
int fbintr(numdevs, mb_devs, intclear)
    int     numdevs;
    struct  mb_device **mb_devs;
    int     (*intclear)();
```

`numdevs` is the maximum number of devices of these type configured. `mb_devs` is the array of devices descriptions. `intclear` is called back to actually turn off the interrupt for a particular device. `intclear` must have the same calling sequence as `cgoneintclear` (line *cgone.7*), i.e., it take the virtual address of the device to disable interrupts. `cg1_intclear` (line *cgone.8*) is a macro that actually disables the interrupts of `cg1fb`.

Open

When an open system call is made at the user level `cgoneopen()` is called.

```
cgoneopen(dev, flag)
    dev_t dev;
{
    return(fbopen(dev, flag, NCGONE, cgoneinfo));
}
```

`cgoneopen()` uses the utility `fbopen()`.

```
int fbopen(dev, flag, numdevs, mb_devs)
    dev_t   dev;
    int     flag, numdevs;
    struct  mb_device **mb_devs;
```

`fbopen()` checks to see if `dev` is available for opening. If not the error ENXIO is returned. If `flag` doesn't ask for write position (FWRITE) then the error EINVAL is returned. Normally, zero is returned on a successful open.

Mmap

The memory map routine in a device driver is responsible for returning a single physical page number of a portion of a device.

```
/*ARGSUSED*/
cgonemmap(dev, off, prot)
    dev_t dev;
    off_t off;
    int prot;
{
    return(fbmmap(dev, off, prot, NCGONE, cgoneinfo, CG1SIZE));
}
```

`cgonemmap()` used the utility `fbmmap()`.

```
int fbmmap(dev, off, prot, numdevs, mb_devs, size)
    dev_t   dev;
    off_t   off;
    int     prot, numdevs, size;
    struct  mb_device **mb_devs;
```

The parameters to `fbmmap()` are similar to `fbopen()`. However, `off` is the offset in bytes from the beginning of the device. `prot` is passed through but currently not used.

Ioctl

A pixrect kernel device driver *must* respond to two input/output control requests:

- FBIOGTYPE — Describe the characteristics of the pixel addressable device.
- FBIOGPIXRECT — Hand out a pixrect that may be used in the kernel. This `ioctl` call is made from within the kernel. This is only required of frame buffers.

```
#if NWIN > 0            /* cgone.9*/
#define  CG1_OPS  &cg1_ops
struct  pixrectops cg1_ops = {
    cg1_rop,        /*cgone.10*/
    cg1_putcolormap,
};
#else
#define  CG1_OPS  (struct pixrectops *)0
#endif
struct  cg1pr cgoneprdatadefault =
    { 0, 0, 255, 0, 0 };
struct  pixrect cgonepixrectdefault =
    { CG1_OPS, { CG1_WIDTH, CG1_HEIGHT }, CG1_DEPTH, /* filled in later */ 0 };

struct  pixrect cgonepixrect[NCGONE];   /*cgone.11*/
struct  cg1pr cgoneprdata[NCGONE];

cgoneioctl(dev, cmd, data, flag)
    dev_t dev;
    caddr_t data;
{
    register int unit = minor(dev);

    switch (cmd) {
    case FBIOGTYPE: {
        register struct fbtype *fb = (struct fbtype *)data;
        fb->fb_type = FBTYPE_SUN1COLOR;
        fb->fb_height = CG1_HEIGHT;
        fb->fb_width = CG1_WIDTH;
        fb->fb_depth = 8;
        fb->fb_cmsize = 256;
        fb->fb_size = CG1_HEIGHT*CG1_WIDTH;
        break;
        }
    case FBIOGPIXRECT: {
        register struct fbpixrect *fbpr = (struct fbpixrect *)data;
        register struct cg1fb *cg1fb =
            (struct cg1fb *)cgoneinfo[(unit)]->md_addr;
        fbpr->fbpr_pixrect = &cgonepixrect[unit];   /*cgone.12*/
        cgonepixrect[unit] = cgonepixrectdefault;   /*cgone.13*/
        fbpr->fbpr_pixrect->pr_data = (caddr_t) &cgoneprdata[unit];/*cgone.14*/
        cgoneprdata[unit] = cgoneprdatadefault; /*cgone.15*/
        cgoneprdata[unit].cgpr_va = cg1fb;  /*cgone.16*/

        cg1_setreg(cg1fb, CG_FUNCREG, CG_VIDEOENABLE);  /*cgone.17*/
```

```
        cg1_intclear(cg1fb);    /*cgone.18*/
        break;
        }

    default:
        return (ENOTTY);
    }
    return (0);
}
```

The *SunView* driver isn't configured into the system when NWIN = 0 (line *cgone.9*). When there is no *SunView* driver, don't reference the pixrect operations `cg1_rop()` and `cg1_putcolormap()`. The kernel version of `cg1_rop()` (line *cgone.10*) only needs to be able to read and write memory pixrects for cursor management. Thus, you can

```
#ifndef KERNEL
/* code not associated with reading and writing */
/* memory pixrects */
#endif KERNEL
```

to reduce the size of the code.

Memory for pixrect public (`pixrect` structure) and private (`cg1pr` structure) objects is provided by arrays of each (line *cgone.11*) NCGONE long. A device `n` in these correspond to device `n` in `cgoneinfo`.

Lines *cgone.12* through *cgone.16* initialize a pixrect for a particular device. This `ioctl` call should enable video for a frame buffer (line *cgone.17*) and disable interrupts as well (line *cgone.18*).

Close

When the device is no longer being referenced, `cgoneclose()` is called. All that is done is that the pixrect data structures of the device are zeroed.

```
cgoneclose(dev, flag)
    dev_t dev;
{
    register int unit = minor(dev);

    if ((caddr_t)&cgoneprdata[unit] == cgonepixrect[unit].pr_data) {
        bzero((caddr_t)&cgoneprdata[unit], sizeof (struct cg1pr));
        bzero((caddr_t)&cgonepixrect[unit], sizeof (struct pixrect));
    }
}

#endif
```

Plugging Your Driver into UNIX

You need to add the device driver procedures to `cdevsw` in `/sys/sun/conf.c` after assigning a new major device number to your driver:

```
#include "cgone.h"
#if NCGONE > 0
int cgoneopen(), cgonemmap(), cgoneioctl();
int cgoneclose();
#else
#define cgoneopen  nodev
#define cgonemmap  nodev
#define cgoneioctl  nodev
#define cgoneclose  nodev
#endif

    {
    cgoneopen,  cgoneclose,  nodev,  nodev,  /*14*/
    cgoneioctl,  nodev,  nodev,  0,
    seltrue,  cgonemmap,
    },
```

Also, you need to add the new files associated with your driver to `/sys/conf/files.sun`:

```
pixrect/cg1_colormap.c optional cgone win device-driver
pixrect/cg1_rop.c optional cgone win device-driver
sundev/cgone.c optional cgone device-driver
```

A.7. Access Utilities

This section describes utilities used by pixrect drivers. The pixrect header files `memvar.h`, `pixrect.h` and `pr_util.h` contain useful macros that you should familiarize yourself with; they are not documented here.

```
pr_clip(dstp, srcp)
    struct  pr_subregion *dstp;
    struct  pr_prpos *srcp;
```

`pr_clip` adjusts the position and size of `dstp`, the destination pixrect subregion, to fall within `dstp->pr`. If `*scrp`, the source pixrect position, is not zero then the position of the source is clipped to fall within `dstp`.

Two operations on operations, `pr_reversesrc()` and `pr_reversedst()`, are provided for adjusting the operation code to take into account video reversing of monochrome pixrects of either the source or the destination.

```
char    pr_reversedst[16];
char    pr_reversesrc[16];
```

These are implemented by table lookup in which the index into the tables is `(op>>1)&0xF` where `op` is the operation passed into pixrect public procedures. This process can be iterated, e.g.,

`pr_reversedst[pr_reversesrc[op]]`.

A.8. Rop

These are the major cases to be considered with the `pwo_rop()` operation:

- Case 1 -- we are the source for the pixel rectangle operation, but not the destination. This is a pixel rectangle operation from the frame buffer to another kind of pixrect. If the destination is not memory, then we will go indirect by allocating a memory temporary, and then asking the destination to operate from there into itself.
- Case 2 -- writing to your frame buffer. This consists of 4 different cases depending on where the data is coming from: from nothing, from memory, from some other pixrect, and from the frame buffer itself. When the source is some other pixrect, other than memory, ask the other pixrect to read itself into temporary memory to make the problem easier.

A.9. Batchrop

A simple batchrop implementation could iterate on the batch items and call rop for each. Even in a more sophisticated implementation, while iterating on the batch items, you might also choose to bail out by calling rop when the source is skewed, or if clipping causes you to chop off in left-x direction.

A.10. Vector

There are some notable special cases that you should consider when drawing vectors:

- Handle length 1 or 2 vectors by just drawing endpoints.
- If vector is horizontal, use fast algorithm.
- If vector is vertical, use fast algorithm.

Importance of Proper Clipping

The hard part in vector drawing is clipping, which is done against the rectangle of the destination quickly and with proper interpolation so that the jaggies in the vectors are independent of clipping.

A.11. Colormap

Each color raster device has its own way of setting and getting the colormap.

Monochrome

For monochrome raster devices, when `pr_putcolormap()` is called, the convention is that if `red[0]` is zero then the display is light on dark, otherwise dark on light. For monochrome raster devices, when `pr_getcolormap()` is called, the convention is that if the display is light on dark then zero is stored in `red[0]`, `green[0]` and `blue[0]` and -1 is stored in other positions in the color map. Otherwise, if the display is dark on light, then zero and -1 are reversed.

A.12. Attributes

`pr_getattributes()` and `pr_putattributes()` operations get or set a bitplane mask in color pixrects, respectively.

Monochrome

Monochrome devices ignore `pr_putattribute()` calls that are setting the bitplane mask. Monochrome devices always return 1 when `pr_getattribute()` asking for the bitplane mask.

A.13. Pixel

`pwo_get()` and `pwo_put()` operations get or set a single pixel, respectively.

A.14. Stencil

In its most efficient implementation, stencil code parallels rop code, all the while considering the 2 dimensional stencil. One way to implement stencil is to use rops. We pay a small efficiency penalty for this. You may not consider writing the special purpose code worthwhile for the bitmap stencils since they probably won't get used nearly as much as rop. Here's the basic idea (`Temp` is a temporary memory pixrect):

```
Temp = Dest
Temp = Dest op Source
Temp = Temp & Stencil
Dest = Dest & ~Stencil
Dest = Dest | Temp
```

i.e.,

```
Dest = (Dest & ~Stencil) | ((Dest op Source) & Stencil)
```

A.15. Polygon

`pr_polyline()` is a natural extension to `pr_vector()`. It is especially useful for devices that can optimize this operation.

B

Pixrect Functions and Macros

B

Pixrect Functions and Macros

B.1. Making Pixrects

Table B-1 *Pixrects*

Name	*Function*
Create Pixrect	`Pixrect *pr_open(devicename)` `char *devicename;`

Create Secondary Pixrect	`#define Pixrect *pr_region(pr, x, y, w, h)`
	`Pixrect *pr;`
	`int x, y, w, h;`

Release Pixrect Resources	`#define pr_close(pr)` `Pixrect *pr;`
Release Pixrect Resources	`#define pr_destroy(pr)` `Pixrect *pr;`
Subregion Create Secondary Pixrect	`#define Pixrect *prs_region(subreg)` `struct pr_subregion subreg;`
Subregion Release Pixrect Resources	`#define prs_destroy(pr)` `Pixrect *pr;`
Convert 680X0 pixrect to 386i pixrect	`void pr_flip(pr)` `Pixrect *pr;`

B.2. Text

Table B-2 *Text*

Name	*Function*

Compute Bounding Box of Text String	`pf_textbound(bound, len, font, text)`
	`struct pr_subregion *bound;`
	`int len;`
	`Pixfont *font;`
	`char *text;`

Compute Location of Characters in Text String	`struct pr_size pf_textbatch(where, lengthp, font, text)`
	`struct pr_pos where[];`
	`int *lengthp;`
	`Pixfont *font;`
	`char *text;`

Compute Width and Height of Text String	`struct pr_size pf_textwidth(len, font, text)`
	`int len;`
	`Pixfont *font;`
	`char *text;`

Load Font	`Pixfont *pf_open(name)` `char *name;`
Load Private Copy of Font	`Pixfont *pf_open_private(name)` `char *name;`
Load System Default Font	`Pixfont *pf_default()`
Release Pixfont Resources	`pf_close(pf)` `Pixfont *pf;`

Unstructured Text	`pr_text(pr, x, y, op, font, text)`
	`Pixrect *pr;`
	`int x, y, op;`
	`Pixfont *font;`
	`char *text;`
	
	`pr_ttext(pr, x, y, op, font, text)`
	`Pixrect *pr;`
	`int x, y, op;`
	`Pixfont *font;`
	`char *text;`

Write Text and Background	`pf_text(where, op, font, text)`
	`struct pr_prpos where;`
	`int op;`
	`Pixfont *font;`
	`char *text;`

Table B-2 *Text— Continued*

Name	*Function*

Write Text	`pf_ttext(where, op, font, text)`
	`struct pr_prpos where;`
	`int op;`
	`Pixfont *font;`
	`char *text;`

B.3. Raster Files

Table B-3 *Raster Files*

Name	*Function*

Initialize Raster File Header	`Pixrect *pr_dump_init(input_pr, rh, colormap, type, copy_flag)`
	`Pixrect *input_pr;`
	`struct rasterfile *rh;`
	`colormap_t *colormap;`
	`int type, copy_flag;`

Read Colormap from Raster File	`int pr_load_colormap(input, rh, colormap)`
	`FILE *input;`
	`struct rasterfile *rh;`
	`colormap_t *colormap;`

Read Header from Raster File	`int pr_load_header(input, rh)`
	`FILE *input;`
	`struct rasterfile *rh;`

Read Image from Raster File	`Pixrect *pr_load_image(input, rh, colormap)`
	`FILE *input;`
	`struct rasterfile *rh;`
	`colormap_t *colormap;`

Read Raster File	`Pixrect *pr_load(input, colormap)`
	`FILE *input;`
	`colormap_t *colormap;`

Read Standard Raster File	`Pixrect *pr_load_std_image(input, rh, colormap)`
	`FILE *input;`
	`struct rasterfile *rh;`
	`colormap_t colormap;`

Write Header to Raster File	`int pr_dump_header(output, rh, colormap)`
	`FILE *output;`
	`struct rasterfile *rh;`
	`colormap_t *colormap;`

Write Image Data to Raster File	`int pr_dump_image(pr, output, rh)`
	`Pixrect *pr;`
	`FILE *output;`
	`struct rasterfile *rh;`

Write Raster File	`int pr_dump(input_pr, output, colormap, type, copy_flag)`
	`Pixrect *input_pr;`
	`FILE *output;`
	`colormap_t *colormap;`
	`int type, copy_flag;`

B.4. Memory Pixrects

Table B-4 *Memory Pixrects*

Name	***Function***

Create Memory Pixrect from an Image	`Pixrect *mem_point(width, height, depth, data)`
	`int width, height, depth;`
	`short *data;`

Create Memory Pixrect	`Pixrect *mem_create(w, h, depth)` `int w, h, depth;`

Create Static Memory Pixrect	`#define mpr_static(name, w, h, depth, image)`
	`int w, h, depth;`
	`short *image;`

Get Memory Pixrect Data Bytes per Line	`#define mpr_linebytes(width, depth)` `( ((pr_product(width, depth)+15)>>3) &~1)`
Get Pointer to Memory Pixrect Data	`#define mpr_d(pr)` `((struct mpr_data *)(pr)->pr_data)`

Variations for the Sun386i:

- `mem_point()` on the Sun386i does not flip the bitmap pointed to by `*data`. The pixrect structure returned does not have the `MP_STATIC` or the `MP_I386` flag set.
- `mem_create()` on the Sun386i creates an empty pixrect with the `MP_I386` flag set.
- `mpr_static()` on the Sun386i creates a pixrect with both the `MP_I386` and `MP_STATIC` flags set.

B.5. Colormaps and Bitplanes

Table B-5 *Colormaps and Bitplanes*

Name	*Function*

Exchange Foreground and Background Colors	`pr_reversevideo(pr, min, max)`
	`Pixrect *pr;`
	`int min, max;`

Get Colormap Entries	`#define pr_getcolormap(pr, index, count, red, green, blue)`
	`Pixrect *pr;`
	`int index, count;`
	`unsigned char red[], green[], blue[];`

Get Plane Mask	`#define pr_getattributes(pr, planes)`
	`Pixrect *pr;`
	`int *planes;`

Set Background and Foreground Colors	`pr_blackonwhite(pr, min, max)`
	`Pixrect *pr;`
	`int min, max;`

Set Colormap Entries	`#define pr_putcolormap(pr, index, count, red, green, blue)`
	`Pixrect *pr;`
	`int index, count;`
	`unsigned char red[], green[], blue[];`

Set Foreground and Background Colors	`pr_whiteonblack(pr, min, max)`
	`Pixrect *pr;`
	`int min, max;`

Set Plane Mask	`#define pr_putattributes(pr, planes)`
	`Pixrect *pr;`
	`int *planes;`

Subregion Get Colormap Entries	`#define prs_getcolormap(pr, index, count, red, green, blue)`
	`Pixrect *pr;`
	`int index, count;`
	`unsigned char red[], green[], blue[];`

Subregion Get Plane Mask	`#define prs_getattributes(pr, planes)`
	`Pixrect *pr;`
	`int *planes;`

Subregion Set Colormap Entries	`#define prs_putcolormap(pr, index, count, red, green, blue)`
	`Pixrect *pr;`
	`int index, count;`
	`unsigned char red[], green[], blue[];`

Table B-5 *Colormaps and Bitplanes— Continued*

Name	*Function*

Subregion Set Plane Mask	`#define prs_putattributes(pr, planes)`
	`Pixrect *pr;`
	`int *planes;`

B.6. Rasterops

Table B-6 *Rasterops*

Name	*Function*

Draw Textured or Solid Lines with Width	`#define pr_line(pr, x0, y0, x1, y1, brush, tex, op)`
	`Pixrect *pr;`
	`int x0, y0, x1, y1;`
	`struct pr_brush *brush;`
	`struct pr_texture *tex;`
	`int op;`

Draw Textured Polygon	`pr_polygon_2(dpr, dx, dy, nbnds, npts, vlist, op, spr, sx, sy)`
	`Pixrect *dpr, *spr;`
	`int dx, dy`
	`int nbnds, npts[];`
	`struct pr_pos *vlist;`
	`int op, sx, sy;`

Draw Vector	`#define pr_vector(pr, x0, y0, x1, y1, op, value)`
	`Pixrect *pr;`
	`int x0, y0, x1, y1, op, value;`

Get Pixel Value	`#define pr_get(pr, x, y)`
	`Pixrect *pr;`
	`int x, y;`

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

Multiple RasterOp	`#define pr_batchrop(dpr, dx, dy, op, items, n)`
	`Pixrect *dpr;`
	`int dx, dy, op, n;`
	`struct pr_prpos items[];`

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

Replicated Source RasterOp	`pr_replrop(dpr, dx, dy, dw, dh, op, spr, sx, sy)`
	`Pixrect *dpr, *spr;`
	`int dx, dy, dw, dh, op, sx, sy;`

Set Pixel Value	`#define pr_put(pr, x, y, value)`
	`Pixrect *pr;`
	`int x, y, value;`

Table B-6 *Rasterops— Continued*

Name	*Function*

Subregion Draw Vector	`#define prs_vector(pr, pos0, pos1, op, value)`
	`Pixrect *pr;`
	`struct pr_pos pos0, pos1;`
	`int op, value;`

Subregion Get Pixel Value	`#define prs_get(srcprpos)` `struct pr_prpos srcprpos;`

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

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

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

Subregion Replicated Source RasterOp	`#define prs_replrop(dsubreg, op, sprpos)`
	`struct pr_subregion dsubreg;`
	`struct pr_prpos sprpos;`

Subregion Set Pixel Value	`#define prs_put(dstprpos, value)`
	`struct pr_prpos dstprpos;`
	`int value;`

Trapezon RasterOp	`pr_traprop(dpr, dx, dy, t, op, spr, sx, sy)`
	`Pixrect *dpr, *spr;`
	`struct pr_trap t;`
	`int dx, dy, sx, sy op;`

B.7. Double Buffering

Table B-7 *Double Buffering*

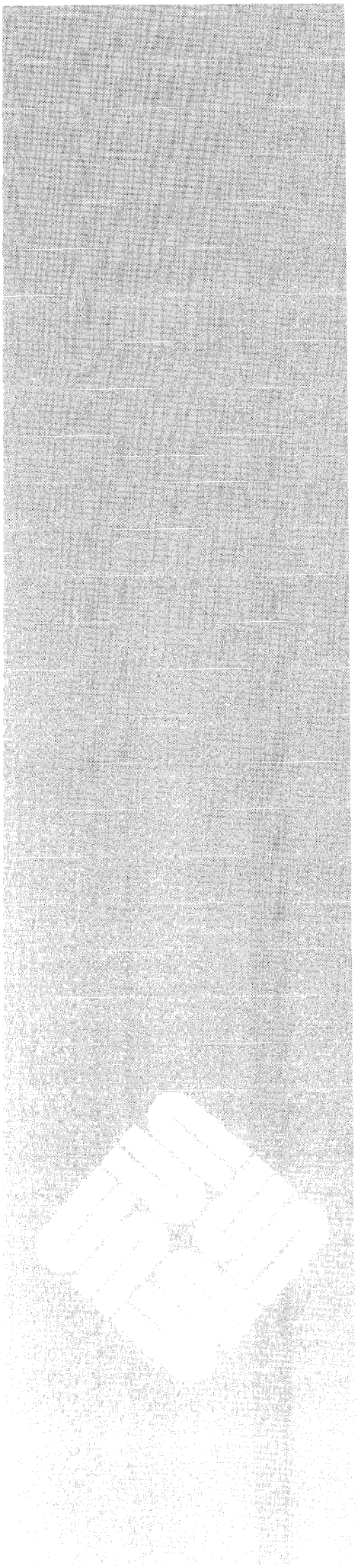
Name	*Function*

Get Double Buffering Attributes	`pr_dbl_get(pr, attribute)`
	`Pixrect *pr;`
	`int attribute;`

Set Double Buffering Attributes	`pr_dbl_set(pr, attribute_list)`
	`Pixrect *pr;`
	`int *attribute_list;`

C

Pixrect Data Structures



C

Pixrect Data Structures

Table C-1 *Pixrect Data Structures*

Name	*Data Structure*

Brush	`typedef struct pr_brush {`
	`  int width;`
	`} Pr_brush;`

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

Font Descriptor	`typedef struct pixfont {`
	`  struct pr_size pf_defaultsize;`
	`  struct pixchar pf_char[256];`
	`} Pixfont;`

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

Table C-1 *Pixrect Data Structures— Continued*

Name	*Data Structure*

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

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

Position Within a Pixrect	`struct pr_prpos {`
	`  struct pixrect *pr;`
	`  struct pr_pos pos;`
	`};`

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

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

Table C-1 *Pixrect Data Structures— Continued*

Name	*Data Structure*

Texture	`typedef struct pr_texture {`
	`  short *pattern;`
	`  short offset;`
	`  struct pr_texture_options {`
	`  unsigned startpoint : 1,`
	`  endpoint : 1,`
	`  balanced : 1,`
	`  givenpattern : 1,`
	`  res_fat : 1,`
	`  res_poly: 1,`
	`  res_mvlist : 1,`
	`  res_right : 1,`
	`  res_close : 1;`
	`  } options;`
	`  short res_polyoff;`
	`  short res_oldpatln;`
	`  short res_fatoff;`
	`} Pr_texture;`

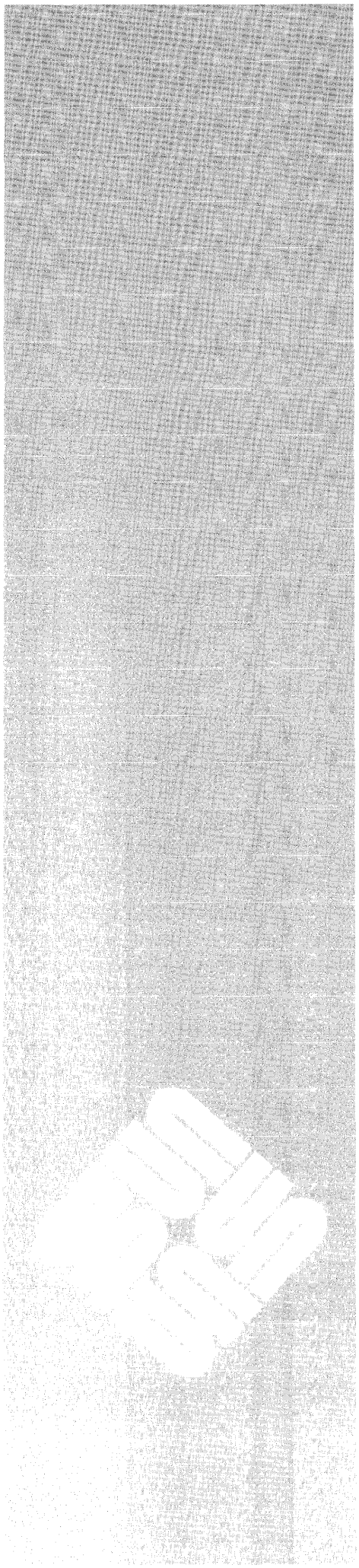
Trapezon	`struct pr_trap {`
	`  struct pr_fall *left, *right;`
	`  int y0, y1;`
	`};`

Trapezon Chain	`struct pr_chain {`
	`  struct pr_chain *next;`
	`  struct pr_size size;`
	`  int *bits;`
	`};`

Trapezon Fall	`struct pr_fall {`
	`  struct pr_pos pos;`
	`  struct pr_chain *chain;`
	`};`

D

Curved Shapes



D

Curved Shapes

This appendix describes `pr_traprop()`, a function for rendering curved shapes with *Pixrect*. `pr_traprop()` is an advanced pixrect operation analogous to `pr_rop()`.

The curve to be rendered must first be stored in a data structure called `pr_trap` which is based on a region called a *trapezon*, rather than on a rectangle. A trapezon is a region with an irregular boundary. Like a rectangle, a trapezon has four sides: top, bottom, left, and right. The top and bottom sides of a trapezon are straight and horizontal. A trapezon differs from a rectangle in that its left and right sides are irregular curves, called *falls*, rather than straight lines.

A fall is a line of irregular shape. Vertically, a fall may only move downward. Horizontally, a fall may move to the left or to the right, and this horizontal motion may reverse itself. A fall may also sustain pure horizontal motion, that is, horizontal motion with no vertical motion.

The figures below show a typical trapezon with source and destination pixrects, and some examples of filled regions that were drawn by `pr_traprop()`.

Figure D-1 *Typical Trapezon*

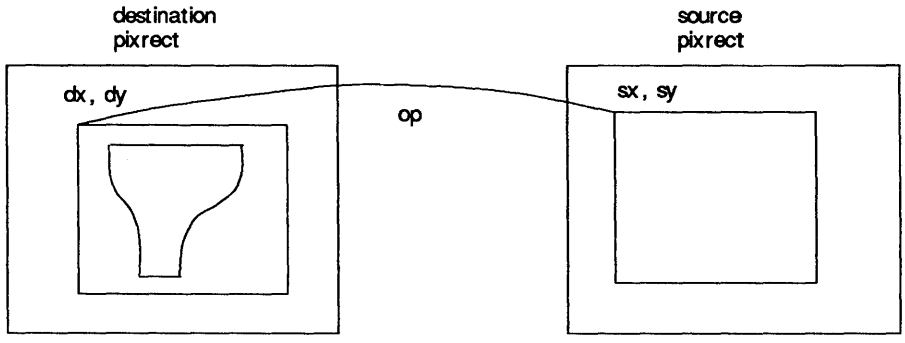
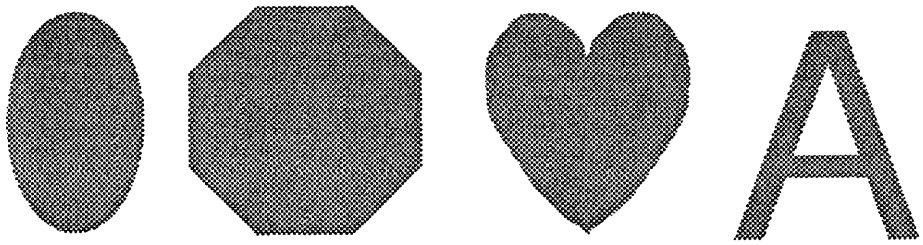


Figure D-2 *Some Figures Drawn by* `pr_traprop()`



```
pr_traprop(dpr, dx, dy, t, op, spr, sx, sy)
struct pixrect *dpr, *spr;
struct pr_trap t;
int dx, dy, sx, sy op;
```

`dpr` and `spr` are pointers to the destination and source pixrects, respectively. `t` is the trapezon to be used. `dx` and `dy` specify an offset into the destination pixrect. `sx` and `sy` specify an offset into the source pixrect. `op` is an op-code as specified previously (see Section 3.3, *The Op Argument*).

```
struct pr_trap {
    struct pr_fall *left, *right;
    int y0, y1;
};

struct pr_fall {
    struct pr_pos pos;
    struct pr_chain *chain;
};

struct pr_chain {
    struct pr_chain *next;
    struct pr_size size;
    int *bits;
};
```

`pr_traprop()` performs a rasterop from the source to the destination, clipped to the trapezon's boundaries. A program must call `pr_traprop()` once per trapezon; therefore this procedure must be called at least twice to draw the letter `A` in Figure D-2.

The source pixrect is aligned with the destination pixrect; the pixel at (`sx, sy`) in the source pixrect goes to the pixel at (`dx, dy`) in the destination pixrect (see Figure D-2).

Positions within the trapezon are relative to position (`dx, dy`) in the destination pixrect. Thus, a position defined as (0,0) in the trapezon would actually be at

(dx, dy) in the destination pixrect.

The structure pr_trap defines the boundaries of a trapezon. A trapezon consists of pointers to two falls (left and right) and two *y* coordinates specifying the top and bottom of the trapezon (y0 and y1). Note that the trapezon's top and bottom may be of zero width; y0 and y1 may simply serve as points of reference.

Each fall consists of a starting position (pos) and a pointer to the head of the list of chains describing the path the fall is to take (chain). A fall may start anywhere above the trapezon and end anywhere below it. pr_traprop() ignores the portions of a fall that lie above and below the trapezon. If a fall is shorter than the trapezon, pr_traprop() will clip the trapezon horizontally to the endpoint of the fall in question. Figure D-3 illustrates the way this works.

A chain is a member of a linked list of structures that describes the movement of the fall. Each chain describes a single segment of the fall. Each chain consists of a pointer to the next member of the chain (next), the size of the bounding box for the chain (size), and a pointer to a bit vector containing motion commands (bits).

Each chain may specify motion to the right and/or down, or motion to the left and/or down; however, a single chain may not specify both rightward and leftward motion. Remember that motion may not proceed upward, and that straight horizontal motion is permitted.

The *x* value of the chain's size determines the direction of the motion: a positive *x* value indicates rightward motion, while a negative *x* value indicates leftward motion. The *y* value of the chain's size must always be positive, since a fall may not move upward (in the direction of negative y).

A chain's bit vector is a command string that tells pr_traprop() how to draw each segment of the fall. Each set (1) bit in the vector is a command to move one pixel horizontally and each clear (0) bit is a command to move one pixel vertically. The bits within the bit vector are stored in byte order, from most significant bit to least significant bit. This ordering corresponds to the left-to-right ordering of pixels within a memory pixrect.

The fall begins at the starting position specified in pr_fall. The motion proceeds downward as specified in the first bit vector in the chain, from the high-order bit to the low-order bit. When the fall reaches the bottom of the bounding box, it continues at the top of the next chain's bounding box. Note that the fall will always begin and end at diagonally opposite corners of a given bounding box.

If a bit vector specifies a segment of the fall that would run outside of the bounding box, pr_traprop() clips that segment of the fall to the bounding box. This would occur when the sum of the 1's in a chain's bit vector exceeds the chain's *x* size, or when the sum of the 0's in the chain's bit vector exceeds the chain's *y* size. When this happens, the segment in question runs along the edge of the bounding box until it reaches the corner of the bounding box diagonally opposite to the corner in which it started.

Index

R

S

T

U

V

W

Notes

Notes

Notes

Notes

Notes

Notes

Notes