Chapter 11

Pixels

This chapter describes the subroutines that allow you to address screen pixels directly and perform pixel operations.

- Section 11.1, “Pixel Formats,”explains the formats used for pixel data.
- Section 11.2, “Reading and Writing Pixels Efficiently,” tells you how to access pixels.
- Section 11.3, “Using pixmode,” tells you how to customize pixel operations.
- Section 11.4, “Subimages within Images,” tells you how to access pixels that are part of larger structures.
- Section 11.5, “Packing and Unpacking Pixel Data,” explains how pixel data is compressed and decompressed.
- Section 11.6, “Order of Pixel Operations,” explains the hierarchy of pixel operations.
- Section 11.7, “Old-Style Pixel Access,” describes methods of pixel access that were used in early releases of the GL, and that are not recommended.

Pixels, like raster fonts, are not nearly as easy to transform (rotate and scale) as geometric figures. Reading and writing pixels on the screen often requires the program to get information about the window dimensions and the screen resolution.

Another problem with reading and writing pixels is that the contents of each pixel can mean different things depending on the display mode for that pixel. The same physical bitplanes are used to store either color map indices or RGB values; accordingly, the mode of the pixel determines whether the contents are interpreted as RGB triples or as indices into a color map.

Three methods of pixel access are supported.

The first method provides a high-performance interface to a flexible set of pixel subroutines—`lrectread()`, `lrectwrite()`, `rectcopy()`, `rectzoom()`, and `pixmode()` These subroutines operate on arbitrarily sized rectangles made up of arbitrarily sized pixels. The behavior of these subroutines can be modified by setting parameters for *offset*, *stride*, *pixel size*, *zoom*, and other features described later. These functions operate in either RGB or color map mode.

The second method is compatible with older versions of the GL. It provides a high-performance interface to operate on arbitrarily sized rectangles. This set of subroutines includes `rectread()`, `rectwrite()`, and `rectzoom()`. The behavior of these subroutines can be modified only for *zoom*. These functions operate in either RGB or color map mode, but because they operate with 16-bit unsigned shorts, they are not generally useful for RGB mode.

The third method is compatible with even older versions of the GL. It provides mode-dependent subroutines to deal with, at most, one scanline at a time, which is positioned according to the system's "current character position". These subroutines are `readpixels()`, `readRGB()`. `writepixels()`, and `writeRGB()`. Continued use of these subroutines is not suggested because higher-performance, more flexible subroutines are available.

11.1 Pixel Formats

The following pixel formats are standard on most IRIS-4D systems:

- RGBA (Red-Green-Blue-Alpha) data is interpreted as four 8-bit values packed into each 32-bit word. Bits 0-7 represent red, bits 8-15 represent green, bits 16-23 represent blue, and bits 24-31 represent alpha.

  For example, 0X01020304 corresponds to a pixel whose RGBA values are 4, 3, 2, and 1, respectively. This is exactly the same format `cpack()` uses.

- CI (Color Index) data is interpreted as 12-bit (low-order) indices into a single, 4096-entry color map. The high-order 20 bits should be zero.

- *z*-buffer data is interpreted as 24-bit (low-order) data. The high-order 8 bits should be zero. Signed *z*-buffer data is sign-extended to 32 bits.

- Other data used in overlay, underlay, and pop-up planes is interpreted as a type of color map data. Because different systems and different configurations support different pixel sizes for these resources, the number of entries contained in the auxiliary color map vary.

On the IRIS Indigo, and the 8-bitplane Personal IRIS, these formats are used:

- RGB values are in 8 bit packed format. The framebuffer stores 8 bits of RGB data (3 bits of red, 3 bits of green, and 2 bits of blue), but RGB pixels are written as 24 bit RGB triples. When read back, the least-significant 5 bits of red and green are zero and the least-significant 6 bits of blue are zero.
- CI values are in 8 bit (3 red + 3 green + 2 blue) packet format in single buffer mode and 4 bit (1 red + 2 green + 1 blue) in double buffer mode.
- Indigo *z*-buffer data is 32 bits in software. The 8-bitplane Personal IRIS does not come standard with a *z*-buffer; however, if you have installed it as an option, the *z*-buffer data is 24 bit, sign-extended to 32 bits.

The pixel formats described above are frame buffer formats. The format of the pixel data in host memory can be packed in a more efficient format if `pixmode()` is used with `lrectread()` and `lrectwrite()`. See Section 11.3, “Using pixmode,” for information on `pixmode()` features.

11.2 Reading and Writing Pixels Efficiently

This section describes subroutines that read and write pixels with the highest possible performance. The subroutines described in this section may not be available on your system, see the man pages to find out if your system supports these commands.

Note: No color mode checking is done, so RGB data read / written in color map mode, and vice-versa, can return undesired results.

11.2.1 Pixel Addressing

Pixel coordinates on the IRIS workstation are at the center of each pixel, as mentioned in Chapters 2 and 7. Because of this, you need to specify pixel boundaries in half-steps, so pixels will be centered on integer values. If you do not do this, your pixel operations can be off by a pixel or more due to round-off errors in the floating-point calculations.

Pixel operations can occur in any of the 4 GL framebuffers (bitplanes), as selected by `drawmode()`. These subroutines establish sources and destinations for pixel operations, as well as other drawing subroutines. Sources apply to pixel reads and copies; destinations apply to pixel writes and copies.

drawmode

`drawmode()` determines the drawing mode, thereby, the destination for pixel operations. `NORMALDRAW` is the default, `OVERDRAW`, `UNDERDRAW`, and `PUPDRAW` are options. In `NORMALDRAW` mode, `frontbuffer()`, `backbuffer()`, and `zbuffer()` apply. If you assert more than one destination in `NORMALDRAW` mode, more than one destination is written.

readsource

`readsource()` determines the GL framebuffer source of pixels read by `rectread()`, `lrectread()`, `rectcopy()`, `readpixels()`, and `readRGB()`. The source depends on the current drawing mode, as set by `drawmode()`.

The default value of *src* is `SRC_AUTO`, which selects the front buffer of the current GL framebuffer in single buffer mode and the back buffer in double buffer mode. `SRC_FRONT` reads from the front buffer, and `SRC_BACK` reads from the back buffer. `SRC_ZBUFFER` reads 24-bit data (32 bit for Indigo) from the *z*-buffer. Other sources such as `SRC_FRAMEGRABBER` are available if special hardware is installed. Some `readsource()` parameters are valid only on certain models; see the `readsource()` man page for specific information about source parameters.

11.2.2 Reading Pixels

The following subroutines read screen pixels.

readdisplay

`readdisplay()` reads a rectangular screen region and returns displayed pixels in packed RGB format. `readdisplay()` returns the displayed value of each addressed pixel for all display bitplanes and modes. You specify the *x* and *y* coordinates of the rectangular region. The pixels are read into *parry* left-to-right, then bottom-to-top according to the *hints* provided. The hints are modifiers, not directives; hence, they are ignored on systems that do not support them.

Table 11-1 list the hints available and their descriptions.

Hint	Description
`RD_FREEZE`	freezes the screen by blocking all graphics calls until the read is completed. This assures that the returned data accurately represents the data displayed at the time of the call.
`RD_ALPHAONE`	returns all alpha values set to one (represented as 0xff for this command.
`RD_IGNORE_PUP`	ignores the contents of the popup framebuffer.
`RD_IGNORE_OVERLAY`	ignores the contents of the overlay framebuffer.
`RD_IGNORE_UNDERLAY`	ignores the contents of the underlay framebuffer.
`RD_OFFSCREEN`	returns an error when attempting to read beyond the framebuffer boundary.

Table 11-1 Hints for `readdisplay()`

rectread and lrectread

`rectread()` reads a rectangular array of pixels from the window where (*x1*,*y1*) are the coordinates for the lower-left corner of the rectangle and (*x2*,*y2*) are the coordinates for the upper-right corner. All coordinates are relative to the lower-left corner of the window in screen coordinates. *sarray* is an array of 16-bit values. Only the low-order 16 bits of each pixel are read, so `rectread()`

is useful primarily for windows drawn in color map mode. The data is loaded into *sarray* left to right and bottom to top.

If the pixel data on the screen looks like this:

```
1  2  3  4
5  6  7  8
9 10 11 12
```

sarray contains {9,10,11,12,5,6,7,8,1,2,3,4}, that is, sarray[0]=9, sarray[1]=10, etc. `rectread()` returns the number of pixels successfully read.

Normally, the number of pixels read is

(x2 – x1 + 1)*(y2 – y1 + 1)

If any part of the specified rectangle is off the screen, or if the coordinates are mixed up, the behavior of `rectread()` is undefined.

Errors occur only outside the screen, not outside the window. It is possible to read pixels outside a window, as long as they are on the physical screen. This can be useful for certain applications that magnify data from other windows, or perform image processing on images produced by other programs. The main difficulty is that the data can come from areas of the screen that are in different color modes (color map or RGB mode). Because `rectread()` is not restricted to the current window, any or all of the coordinates can be negative.

`lrectread()` is similar to `rectread()` except that *larray* contains 32-bit quantities and the behavior of `lrectread()` is affected by `pixmode()` settings. Using `pixmode()` with `lrectread()` provides useful data manipulation functions as well as data packing functions to store and transfer data more efficiently.

11.2.3 Writing Pixels

The following subroutines write to screen pixels:

rectwrite and lrectwrite

`rectwrite()` writes a rectangular array of pixels to the window, where (*x1*,*y1*) are the coordinates for the lower-left corner of the rectangle and (*x2*,*y2*) are the coordinates for the upper-right corner. All coordinates are relative to the lower-left corner of the window in screen coordinates. *sarray* is an array of 16-bit values. Only the low-order 16 bits of each pixel are written, so `rectwrite()` is useful primarily for windows in color map mode. The data is written from *sarray* from left to right, then bottom to top `rectwrite()` obeys the zoom factors set by `rectzoom()` (see `rectzoom()` below). `writemask()` and `scrmask()` apply as with other drawing primitives.

`lrectwrite()` is similar to `rectwrite()` except that *larray* contains 32-bit quantities and the behavior of `lrectwrite()` is affected by `pixmode()` settings. Using `pixmode()` with `lrectwrite()` provides useful data manipulation functions as well as data unpacking functions to store and transfer data more efficiently.

rectcopy

`rectcopy()` copies the pixels from a rectangular region of the screen to a new region. As with `rectread()` and `lrectread()`, the source rectangle must be on the physical screen, but not necessarily constrained to the current window. The bitplane source is determined by `readsource()`, and the bitplane destination is determined by `drawmode()`, `frontbuffer()`, `backbuffer()`, and `zdraw()`.

During a `rectcopy()`, the source rectangle can be zoomed by parameters established with `rectzoom()`. In addition, data manipulation and mirroring can be accomplished through `pixmode()` settings.

rectzoom

`rectzoom()` sets the independent *x* and *y* zoom factors for `rectwrite()`, `lrectwrite()`, and `rectcopy()`. *xzoom* and *yzoom* are positive floating point values. Values less than 1.0 are allowed and cause rectangles to shrink. Some

hardware platforms are not capable of providing noninteger zoom, so only the integer portion of the zoom parameters apply in that case. See the `rectzoom()` man page for system dependencies.

If the following rectangle is copied after calling `rectzoom(2.0, 3.0)`:

```
1 2
3 4
```

the following copy is made; the pixel on the bottom left-hand corner is at (*new x*, *new y*):

```
1 1 2 2
1 1 2 2
1 1 2 2
3 3 4 4
3 3 4 4
3 3 4 4
```

The following sample program, *zoom.c*, is a simple magnification program. It magnifies the rectangular area above and to the right of the cursor to fill the window.

```
#include <gl/gl.h>
#include <gl/device.h>

#define X           0
#define Y           1
#define XY          2

#define ZOOM        3

main()
{
    long org[XY], size[XY], readsize[XY];
    Device dev;
    short val;
    Device mdev[XY];
    short mval[XY];
    Boolean run;

    prefsize(400, 400);
    winopen("zoom");
    qdevice(ESCKEY);
    qdevice(TIMER0);
    noise(TIMER0, getgdesc(GD_TIMERHZ)/10);/* 10 samples/sec */
    getorigin(&org[X], &org[Y]);
```

```
    getsize(&size[X], &size[Y]);
    readsize[X] = size[X] / ZOOM;
    readsize[Y] = size[Y] / ZOOM;
    rectzoom((float)ZOOM, (float)ZOOM);
    mdev[X] = MOUSEX;
    mdev[Y] = MOUSEY;
    run = TRUE;
    while (run) {
    switch (qread(&val)) {
    case TIMER0:
        getdev(XY, mdev, mval);
        mval[X] -= org[X];
        mval[Y] -= org[Y];
        rectcopy(mval[X], mval[Y],
            mval[X] + readsize[X], mval[Y] + readsize[Y], 0, 0);
    break;
    case ESCKEY:
     if (val == 0)
        run = FALSE;
     break;
    }
 }
 gexit();
 return 0;
}
```

After determining the size and shape of the window, the program simply loops, copying an appropriately sized rectangle above and to the right of the cursor into the window magnified by a factor of 3 in each direction. The expressions *x-xorg* and *y-yorg* convert the cursor's screen coordinates into window coordinates.

To be a useful tool, the program should have a mechanism for changing to and from RGB mode, perhaps a method to change zoom factor, and perhaps code to avoid `rectcopy()` if the mouse has not moved since the last time. It might also make a better user interface if the region around the cursor is magnified rather than the area above and to the right of it. Using this tool as is, note that regions of the screen drawn in RGB mode appear incorrect, and color-mapped portions look fine. Also, notice that with double-buffered programs, the zoom window appears to blink. This is caused by buffer swapping in the double-buffered program, while *zoom* is always reading from the same buffer. If the zoom window is magnified, a *zoom* recursion takes place and the effects are interesting.

11.3 Using pixmode

`pixmode()` allows you to customize pixel operations when you use `lrectwrite()`, `lrectread()`, or `rectcopy()`. `pixmode()` is only available on certain systems, see the `pixmode()` man page for It can be used to specify pixel operations such as shifting, expansion, offsetting, and packing and unpacking. Each function is selected by calling `pixmode()` with an argument indicating the operation and a value for that operation. Any or all functions can be used in combination. Once you select a `pixmode()` operation, it remains in effect until you change it. `pixmode()` operations have no effect on `rectread()` or `rectwrite()` or any of the old-style pixel access subroutines.

11.3.1 Shifting Pixels

32-bit pixel data can be shifted left or right before being written to a frame buffer destination or before being read into memory using the `pixmode()` operation `PM_SHIFT`.

For example, to shift all pixels written with `lrectwrite()` left by eight bits so that the red byte is placed in the green byte's position, the green byte is placed in the blue byte's position, the blue byte is placed in the alpha byte's position, and the alpha byte is lost, before calling `lrectwrite()` to write the pixel data, call:

```
pixmode(PM_SHIFT,8);
```

To disable shifting after you have enabled it, use:

```
pixmode(PM_SHIFT,0);
```

You can achieve the same effect when copying pixel data using `rectcopy()`. The same call also affects `lrectread()`, but in this case a right shift is indicated with a negative value. Thus, if you perform the sequence of operations:

```
pixmode(PM_SHIFT,8);
lrectwrite();
```

then call `lrectread()` to read the pixels you just wrote, the pixels you get are exactly the same (unshifted) pixels you wrote, but with the alpha byte stripped off. This is because the pixels were shifted left eight bits when they were written and then shifted right eight bits when they were read back.

The allowable values for PM_SHIFT are 0, 1, 4, 8, 12, 16, and24. A positive value indicates a left shift for lrectwrite() and rectcopy() and a right shift for lrectread(); a negative value indicates a right shift for lrectwrite() and rectcopy() and a left shift for lrectread(). The default value is 0.

11.3.2 Expanding Pixels

You can expand a single-bit pixel into one of two 32-bit values using PM_EXPAND with PM_C0 and PM_C1. When you set the pixmode() value PM_EXPAND to 1, the least significant bit of a pixel's value controls the conversion of that bit into PM_C0 or PM_C1. If the bit is 0, PM_C0 is selected; if it is 1, PM_C1 is selected. For example, to convert a series of single-bit pixels (32-bit pixels whose most significant 31 bits are ignored) into blue for 0 and green for 1 in RGB mode; before calling lrectwrite() call:

```
pixmode(PM_EXPAND, 1);
pixmode(PM_C0, 0x00ff0000);
pixmode(PM_C1, 0x0000ff00);
```

The call pixmode(PM_EXPAND,1) turns expansion on. The next two calls set the expansion values for the single-bit values 0 and 1, respectively.

To turn expansion off, call pixmode(PM_EXPAND, 0). This is the default. You can set PM_C0 and PM_C1 to any 32-bit value. In color map mode, the value of the color index is replaced by either PM_C0 or PM_C1. Their default values are 0.

PM_SHIFT can be used with PM_EXPAND to cause expansion based on a bit other than the least significant one.

11.3.3 Adding Pixels

You can add a constant signed 32-bit value to the least significant 24 bits of each pixel transferred by calling:

```
pixmode(PM_ADD24, value);
```

where *value* is the signed 32-bit value. This feature is most effectively used when transferring *z* data, but it also affects color data. To turn off pixel addition, call pixmode(PM_ADD24, 0). This is the default.

11.3.4 Pixels Destined for the *z*-Buffer

You can specify that pixels be sent to the *z*-buffer by setting:

```
pixmode(PM_ZDATA,1);
```

Unlike setting `zdraw(TRUE)`, using `pixmode(PM_ZDATA,1)` treats transferred pixels as *z* data rather than color data. If you have called `zbuffer(TRUE)`, the writing is conditional based on a comparison of the pixel's value with the value present in the corresponding location of the *z*-buffer. The current setting of `zfunction()` determines the write condition.

To turn off this feature, call:

```
pixmode(PM_ZDATA, 0);
```

This is the default. `PM_ZDATA` has no effect on `lrectread()`.

11.3.5 Changing Pixel Fill Directions

The default directions for reading, writing, and copying pixel rectangles are left-to-right and bottom-to-top. For example, when writing a rectangle, the first pixel is placed in the lower left-hand corner of the specified rectangle, the next at the same screen *y* value but at a screen *x* value of one greater, and so on until a line of pixels is complete. The next line is placed on top of the last until the rectangle is complete.

You can change the default reading, writing, and copying directions. To make the pixel transfer direction top-to-bottom, use the following command:

```
pixmode(PM_TTOB, 1);
```

Top-to-bottom mode provides faster pixel read / write performance on IRIS Indigo Entry, XS, XS24, and Elan systems than does the default.

To make the fill direction right-to-left, use:

```
pixmode(PM_RTOL, 1);
```

For instance, you can mirror a pixel rectangle that is already in the framebuffer about a vertical line to produce a new rectangle by calling:

```
pixmode(PM_RTOL, 1);
rectcopy( ... );
```

You can set both PM_TTOB and PM_RTOL to 1 if you choose. Reset the directions to the defaults with:

```
pixmode(PM_TTOB, 0);
pixmode(PM_RTOL, 0);
```

These functions change the order in which pixels are filled, but do not affect the fundamental row-major ordering of pixels. That is, a horizontal line of pixels always occupies a set of contiguous words in memory, and a group of words representing one line of pixels is followed in memory by a group of words representing the next. Pixels are always arranged so that a group of contiguous words forms a horizontal line, never a vertical line. Changing fill directions does not allow interchanging a horizontal line of pixels for a vertical one.

Fill direction does not affect the location of the destination rectangle for rectcopy(). The destination rectangle is always specified by its lower-left pixel, regardless of fill direction.

There are no restrictions on using these modes with lrectwrite() or lrectread().

11.4 Subimages within Images

Using pixmode() allows you to read and write pixel subrectangles to and from a larger pixel rectangle. Suppose you have a 2000×1500 pixel image and want to work with a 100×200 subimage whose origin is at (150,500) in the larger rectangle. You need to tell the GL the width of the larger rectangle with pixmode():

```
pixmode(PM_STRIDE, 2000);
```

PM_STRIDE specifies the number of 32-bit words per scanline of your rectangle. Next you need to compute the address of the starting pixel of your subrectangle within the large rectangle. If *p* points to the lower-left pixel of the large rectangle, then, assuming the default pixel fill directions, this address is

```
p + (2000 * 500) + 150
```

You can then call lrectread() or lrectwrite():

```
lrectread(x, y, x + 99, y + 199, p + (2000 * 500) + 150 );
lrectwrite(x, y, x + 99, y + 199, p + (2000 * 500) + 150 );
```

to read or write the subimage from or to the location (*x*,*y*) on the screen. The GL figures out where the appropriate subimage is located in CPU memory to effect the desired transfer within the larger rectangle. You can use this method to work with sub-rectangles of any size and offset as long as you tell the GL the width of the whole rectangle using `PM_STRIDE`.

The default value for `PM_STRIDE` is 0. `PM_STRIDE` has no effect on `rectcopy()`.

11.5 Packing and Unpacking Pixel Data

You can specify a number of bits-per-pixel other than 32 for pixels in CPU memory using the `pixmode()` function `PM_SIZE`. You can use this feature to obtain more efficient packing of pixel data in CPU memory. Setting `PM_SIZE` to a value other than 32 (the default value) unpacks pixels from CPU memory when written using `lrectwrite()` and packs pixels into CPU memory when using `lrectread()`.

If you are ignoring alpha values and are using RGB mode, you can specify a `PM_SIZE` of 24 and pack four RGB values into three words. If you set `PM_SIZE` to *n* (allowable values are 1, 4, 8, 12, 16, 24, and 32), the first pixel goes in the *n* most significant bits of the first word. The next pixel is packed adjacent to the first, so that if *n* is less than 32, its bits are placed in the next most significant bits of the first word after the first *n*. If there is not enough room in the first word for all the bits of this second pixel, the leftover bits fill the most significant bits of the following word. The second word is packed tightly in the same way, and so on for the rest of the words until the end of a line of pixels.

Note: The next line of pixels starts in the most significant bit of a new word, even if the last pixel of the previous line did not completely fill the last word of the previous line.

For instance, for a 3×3 rectangle with `PM_SIZE` set to 12, the first pixel occupies the first 12 most significant bits of the first word, the second pixel occupies the next most significant 12 bits of the first word, and the third pixel occupies the last 8 bits of the first word and the first 4 most significant bits of the second word. The next pixel begins a new line, so it occupies the 12 most significant bits of the third word, and so on.

The packing scheme makes 8- and 16-bit packing equivalent to *char* and *short* arrays, respectively, for a line of pixels. Recall, however, that an address passed to `lrectwrite()` or `lrectread()` must be long-word aligned.

If `PM_SIZE` is not 32, pixels might not begin on a word boundary. This might require using the `pixmode()` function `PM_OFFSET` when accessing subrectangles within rectangles.

Calling `pixmode(PM_OFFSET, n)` where *n* is a value between 0 and 31, indicates that the most significant *n* bits of the first word of each scanline are to be ignored. Assume you have a subrectangle of 100×200. The whole image is 1250×1250, the origin is at (150,500), and the pixels are packed at 24 bits per pixel instead of 32. In this case, there are 1250 pixels at 24 bits per pixel, or 30,000 bits in each scanline. Thus there are 30,000 / 32 (rounded up to the nearest integer, because each scanline begins with a new word), or 938 words per scanline. To find the address of the beginning of the sub-rectangle, you must find the offset of the 150th pixel from the first word of a scanline in the large rectangle. This offset is (150 ∗ 24) / 32, or 111, when rounded down. But 111 words is actually (111 ∗ 32) / 24 = (exactly) 148 pixels. The 149th pixel occupies the most significant 24 bits of the 112th word, so the 150th pixel begins 24 bits from the beginning of the 112th word.

Therefore, to access the subimage in this example, call:

```
pixmode(PM_SIZE, 24);
pixmode(PM_STRIDE, 938);
pixmode(PM_OFFSET, 24);
```

and give the pointer:

```
p + (500 * 938) + 112
```

to `lrectread()` or `lrectwrite()` as the location of the first pixel in the subimage. The offset of (500 ∗ 938) accounts for the first 500 lines of the large rectangle that must be skipped, while the offset of 112 brings the pointer to the word containing the 150th pixel in the scanline. The call to `pixmode(PM_OFFSET, 24)` effects the additional required offset of 24 bits to get to the 150th pixel itself.

Setting packing or unpacking parameters, `PM_SIZE` or `PM_OFFSET`, has no effect on `rectcopy()`.

11.6 Order of Pixel Operations

Various `pixmode()` functions can be used together. The order of calls to `pixmode()` is of no consequence. This is because `pixmode()` functions happen in a predefined order. For `lrectwrite()`, this order is

`unpack` → `shift` → `expand` → `add24` → `zoom`

For `lrectread()`, the order is

`shift` → `expand` → `add24` → `pack`

For `rectcopy()` the order is

`shift` → `expand` → `add24` → `zoom`.

As an example, one useful combination is to display a black and white image in RGB mode from an efficiently packed 1-bit-per-pixel encoding in memory. To do this, make the following calls:

```
pixmode(PM_SIZE, 1);
pixmode(PM_EXPAND, 1);
pixmode(PM_C0, 0x00000000);
pixmode(PM_C1, 0x00ffffff);
```

before writing the image with `lrectwrite()`. You might also set `PM_STRIDE` and `PM_OFFSET` if you want to handle a subimage.

You can also read and pack a black and white image using a similar method with `lrectread()`, but you would have to be certain that the black pixels had least significant bits of 0 and the white pixels least significant bits of 1, because the packing discards all but one bit. Also, when reading back the image, there would be no need for expansion. In any case, the order in which you make the `pixmode()` calls is unimportant.

11.7 Old-Style Pixel Access

The subroutines in this section read and write pixels. They determine the location of the pixels on the basis of the current character position (see `cmov()` and `getcpos()`). They attempt to read or write up to *n* pixel values, starting from the current character position and moving along a single scanline (constant *y*) in the direction of increasing *x*. The system updates that position

to the pixel that follows the last one read or written. The current character position becomes undefined if the next pixel position is greater than `getgdesc(GD_XMAX)`. The system paints pixels from left to right and clips them to the current screenmask.

These subroutines do not automatically wrap from one line to the next, and they ignore zoom factors set by `rectzoom()`. They are sensitive to color mode. The current color mode should be set appropriately when calling the following subroutines. The behavior of `readpixels()` and `writepixels()` is undefined in RGB mode. The behaviors of `readRGB()` and `writeRGB()` are undefined in color map mode.

11.7.1 Reading Pixels

This section describes the old-style pixel reading subroutines.

readpixels

`readpixels(n, colors)` reads up to *n* pixel values from the bitplanes in color map mode, to an array of shorts, *colors*. it returns the number of pixels the system actually reads. The values of pixels read outside the physical screen are undefined.

readRGB

`readRGB(n, red, green, blue)` reads up to *n* pixel values from the bitplanes in RGB mode. It returns the number of pixels the system actually reads in the arrays of chars *red*, *green*, and *blue*. The values of pixels read outside the physical screen are undefined.

11.7.2 Writing Pixels

This section describes the old-style pixel writing subroutines.

writepixels

`writepixels(n, colors)` paints a row of pixels on the screen in color map mode. n specifies the number of pixels to paint and *colors* specifies an array containing a color for each pixel.

writeRGB

`writeRGB(n, red, green, blue)` paints a row of pixels on the screen in RGB mode. n specifies the number of pixels to paint. r*ed*, *green*, and *blue* specify arrays of colors for each pixel. `writeRGB()` supplies a 24-bit RGB value (8 bits each for red, green, and blue) for each pixel and writes that value directly into the bitplanes.