Chapter 3

Characters and Fonts

This chapter describes the subroutines that position and draw characters, define fonts, and determine information about the currently defined font.

- Section 3.1, “Drawing Characters,” describes how to draw characters.
- Section 3.2, “Creating a Font,” tells you how to create and enable your own font and how to query the system for font information.

The GL font interface supports the rapid display of raster characters in user-selectable fonts. You can design and use your own fonts, or make use of the default fonts supplied with the system. Typefaces exist or can be created with a variety of point sizes and with a fixed or variable pitch.

The IRIS Font Manager™, described in the *Graphics Library Windowing and Font Library Programming Guide*, offers a more flexible way to display text in a program than the method presented in this chapter. You can use the IRIS Font Manager to access externally created fonts, such as bitmap fonts, screen fonts, and X Window System fonts like *Portable Compiler Format* (PCF) fonts. See the *Graphics Library Windowing and Font Library Programming Guide* for information about loading and using externally created fonts and other facilities available through the IRIS Font Manager.

The font information in this chapter is provided mainly for performance reasons. By using the techniques presented in this chapter, you are using methods close to the hardware level of the system. Performance is as much as four times faster than is possible using the IRIS Font Manager. To get your application to display characters as rapidly as possible, use the techniques presented in this chapter.

There are two versions of the font definition and character drawing routines. The older version uses shorts as parameters and is thus limited to a character

set that can only store 256 bitmap definitions and can handle only single-byte character data. Newer subroutines that use long integers, signified by the letter *l* in the subroutine name, can accommodate multibyte data and a more extensive index space.

Because the long integer subroutines offer a more flexible interface, their use is encouraged.

3.1 Drawing Characters

Use cmov() to position text and `charstr()` / `lcharstr()` to draw text. `cmov()` determines where the system draws text on the screen, `charstr()` draws a string of single-byte characters, and `lcharstr()` draws a string of multibyte characters.

The character string is drawn in the current color and in the current font. *Scaling*, *rotating*, or *translating* characters, operations that are described in Chapter 7, "Coordinate Transformations," has no effect on them. For example, when a geometry that has text labels associated with it shrinks as it moves away from the viewer, its labels remain the same size. Similarly, no matter what rotation is in effect, the character string maintains the same orientation, which is horizontal for any standard font.

3.1.1 Determining the Current Character Position

The *current character position* determines where the system draws text on the screen. `cmov()` moves the current character position to a specified point (*x*, *y*, *z*) in world coordinates. `cmov()` turns the world coordinates into window coordinates for the new character position. `cmov()` does not draw anything—it simply sets the character position where drawing is to occur when `charstr()` is issued. `cmov()` does not affect the current graphics position.

The current character position is *transformed* (see Chapter 7) in exactly the same way as a vertex to position a character string where it belongs.

Table 3-1 lists the `cmov()` subroutines.

Argument Type	2-D	3-D
Short integer	`cmov2s()`	`cmovs()`
Long integer	`cmov2i()`	`cmovi()`
Float	`cmov2()`	`cmov()`

Table 3-1 `cmov()` Subroutines

Note: Character position includes *z* values as well as *x* and *y* values. Because of this, you can associate string origins with 3-D locations and you can use z-buffering, described in Chapter 8, and depth-cueing, described in Chapter 13, with characters.

3.1.2 Drawing Character Strings

Use `charstr()` to draw a string of raster characters. Use `lcharstr()` to draw a character string using a long integer raster font—that is, characters that have been defined with `deflfont()`.

The text string is drawn in the current font using the current color. The origin of the first character in the string is the current character position. After the system draws the string, it updates the current character position to the pixel to the right of the last character in the string. Character strings are null-terminated in ANSI C.

3.1.3 Clipping Character Strings

If the origin of a character string lies outside the *viewport* (see Chapter 7), no characters in the string are drawn.

If the origin of a character string is inside the viewport, the characters are individually *clipped* to the *screenmask*. A screenmask establishes a rectangular boundary on the screen. Clipping means that characters inside the boundary are displayed and characters outside of the boundary are not displayed.

Figure 3-1 shows how characters are clipped to the viewport and screenmask.

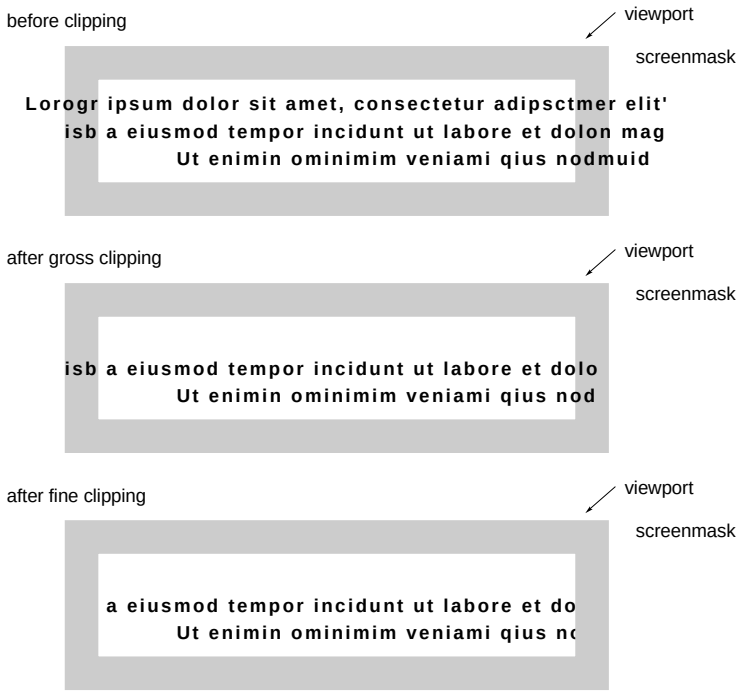

Figure 3-1 How Character Strings are Clipped

In *gross clipping*, character strings that appear outside the viewport are clipped out (not displayed).

There is a "gray area" between the screenmask and the viewport when the viewport is larger than the screenmask. In *fine clipping*, character strings that begin inside the viewport are clipped to the screenmask.

3.1.4 Getting Character Information

Use `getcpos()` to return the screen coordinates of the current character position into the locations pointed to by *ix* and *iy*:

```
void getcpos(short *ix, short *iy)
```

Use `strwidth()` / `lstrwidth()` to return the width of a text string in pixels:

```
long strwidth(String str)

long lstrwidth(long type, void *str)
```

The string can be any null-terminated ASCII string of characters. The value returned does not necessarily represent the width of a character times the number of characters in the string, because in some fonts, the character width varies from one character to another. The default font has a fixed width of nine pixels, so for that font, `strwidth()` does return nine times the string's length.

The *rasterchars.c* sample program draws two lines of text.

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

main()
{
    prefsize(400, 400);
    winopen("rasterchars");
    color(BLACK);
    clear();
    color(RED);
    cmov2i(50, 80);
    charstr("The first line is drawn ");
    charstr("in two parts. ");
    cmov2i(50, 80 - 14);
    charstr("This line is 14 pixels lower. ");
    sleep(10);
    gexit();
    return 0;
}
```

The first line of text in *rasterchars.c* is drawn in two parts. The first `cmov2i()` sets the current character position to 50 pixels to the right and 80 pixels up from the lower-left corner of the window. After the first string is drawn, the current character position is automatically advanced to follow the space character at the end of the line. When the character string "`in two parts.`" is drawn, it continues from the current character position. Next, the character position is set to start 14 pixels below the beginning of the top line, and the second line is drawn.

The characters are drawn in the current color (RED). Because nothing was mentioned in the program about fonts, all the strings are drawn in the default font (font 0), which is defined when `winopen()` is called.

The next sample program, *rasterchars2.c*, shows that character strings are drawn in the same orientation no matter where they move, and that the current character position is transformed like any other geometry.

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

float p[3][2] = {
   {0.0, 0.0},
   {0.6, 0.0},
   {0.0, 0.6}
};

main()
{
   int i;

   prefsize(400, 400);
   winopen("rasterchars2");
   ortho2(-1.0, 1.0, -1.0, 1.0);
   for (i = 0; i < 40; i++) {
      color(BLACK);
      clear();
      rotate(50, 'z');
      color(RED);
      bgnpolygon();
         v2f(p[0]);
         v2f(p[1]);
         v2f(p[2]);
      endpolygon();
      color(GREEN);
      cmov2(p[0][0], p[0][1]);
      charstr("vert0");
      cmov2(p[1][0], p[1][1]);
      charstr("vert1");
      cmov2(p[2][0], p[2][1]);
      charstr("vert2");
      sleep(1);
   }
   gexit();
   return 0;
}
```

The output of *rasterchars2.c* shows a red triangle with its three vertices labeled *vert1*, *vert2*, and *vert3*. As the triangle rotates about *vert1*, the labels *vert2* and *vert3* move along with the triangle is if they were pinned to their respective vertices. The labels are drawn horizontally, no matter what position the triangle is in.

The `rotate()` subroutine rotates the scene about the *z* axis (coming directly out of the screen) by 5 degrees each time. The rotation is about the origin, so vertex *p1* remains fixed. See Chapter 7 for a description of `rotate()`.

3.2 Creating a Font

A font is a collection of rectangular arrays of *masks*. Masks determine whether or not a pixel is turned on. If a 1 appears in a mask, the corresponding pixel is turned on to the current color; if a 0 appears, the pixel is left as it is. For example, the following bitmask might be used to draw the character A:

```
Binary              Hexadecimal
0000011000000000  =  0x0600
0000011000000000  =  0x0600
0000111100000000  =  0x0F00
0000111100000000  =  0x0F00
0001100110000000  =  0x1980
0001100110000000  =  0x1980
0011000011000000  =  0x30C0
0011111111000000  =  0x3FC0
0110000001100000  =  0x6060
0110000001100000  =  0x6060
1100000000110000  =  0xC030
1100000000110000  =  0xC030
```

Figure 3-2 Bitmask for the Character A

A font made up of single-byte characters can have definitions for any character value between 1 and 255. Typically, the bitmask entry for each ASCII character is a mask that draws that character. For example, the ASCII value of A is 65 (decimal), so entry 65 in the font is associated with the bitmask for A shown in Figure 3-2. If this font were defined, the string "AAA" would draw three copies of the character whose bitmask appears in Figure 3-2.

In addition to the bitmask information for each character, you need to know the width and height of the character in pixels. The width cannot be inferred from the bitmask, because all bitmask data comes in 16-bit words. In the letter A example in Figure 3-2, the width is 12 and the height is also 12.

Normally, a character's origin is at the lower-left corner of the bitmask, as is the case for the A. The character is drawn by placing its bitmask so that the bitmask's origin is at the current character position.

Figure 3-3 shows a sample character definition for the letter g.

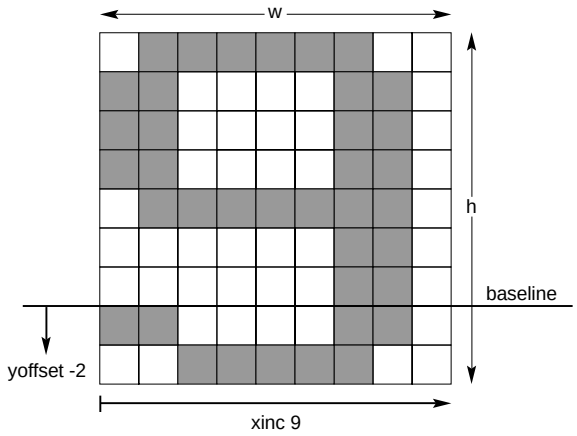

Figure 3-3 Character Bitmap Created with `defrasterfont()`

To define a single character like the one in Figure 3-3, you need the bitmask itself, *width*, *height*, *xoffset*, *yoffset*, and *xincrement*. The `defrasterfont()` and `deflfont()` subroutines allow you to define a collection of such characters.

For a character with a descender, such as g, j, or y, the two bottom lines of the bitmask should lie below the current character position, so the origin should not be at the lower-left corner. Two values, the *xoffset* and the *yoffset*, tell how far the character's origin must be moved to bring it to the lower-left corner. For characters with descenders, *yoffset* is typically negative (see Figure 3-3).

Finally, another number for each character indicates how far to the right the current character position must be advanced after drawing the character. This is usually different from the width, and is labeled the *x* increment. In the A example above, the character position would probably be advanced by 14 pixels to leave a little space between it and the next character.

To simplify matters, the character bitmasks are packed together in one array of 16-bit values, so the bitmask is determined by the offset into the bitmask array.

For example, if a single-byte font contains the letter A from Figure 3-2 as its first character, and a bitmask for B as its second, the offset for B is 12 shorts (the length of the bitmask definition of A). The length and width together determine the number of shorts in a character's definition.

3.2.1 Defining the Font

There are two different subroutines for defining a font: `defrasterfont()` and `deflfont()`.

The `defrasterfont()` command is limited to 256 bitmap definitions within the raster array and uses `charstr()` to operate on the input string as a stream of single-byte character data.

Extended character sets require the use of *multibyte* character data, which is supported by `deflfont()`. The interface for using multibyte character sets is analogous to the single-byte interface, but is capable of containing more font information. This interface was designed primarily to support international fonts; however, you may find it useful for other purposes as well.

Using Single Byte Character Data

Use `defrasterfont()` to define a single-byte raster font. Font 0 is the default raster font, which you cannot redefine. It is a Helvetica-like font with fixed-pitch characters. If the viewport is set to the whole screen, approximately 142 of the default characters fit on a line (1 character occupies 9 pixels). If baselines are 16 pixels apart, 64 lines of the default characters fit on the screen.

The ANSI C specification for `defrasterfont()` is:

```
void defrasterfont(short n, short ht, short nc,
Fontchar [chars],short nr,unsigned short raster[])
```

where:

n is an index into a font table. Font 0 is the default font; it cannot be redefined.

ht is the maximum height of the font characters in pixels.

nc is the number of elements in the *chars* array. The first 32 entries of *chars* are usually undefined because they correspond to the ASCII control characters.

chars is an array of character descriptions of type *Fontchar*, which is defined in the header file *gl.h*. The description includes the *height* and *width* of the character in pixels, the *offsets* from the

character origin to the lower-left corner of the bounding box, an *offset* into the array of *rasters*, and the amount to add to *x* of the current character position after drawing the character.

nr is the number of 16-bit integers in *raster*.

raster is a one-dimensional array of *nr* bitmask bytes, ordered from left to right, then bottom to top. Mask bits are left-justified in the character's bounding box.

The following code fragment contains the defining parameters for the character in Figure 3-3 and shows the data in location 724 of the *chars* array:

```
defrasterfont(n, ht, nc, chars, nr, rasters);
chars ['g'] = {724, 8, 9, 0, -2, 9 }
short rasterarry [] =  {...
                       ...
                       0x7E00, 0xC300, 0x0300, 0x0300,
                       0x7F00, 0xC300, 0xC300, 0xC300,
                       0x7E00,
                       ...
                       }
```

Using Multiple Byte Character Data

Extensive character sets such as Asian ideograms require more raster data and a larger index space than is provided by `defrasterfont()`. Use `deflfont()` for multibyte character sets. The `deflfont()` subroutine is similar to `defrasterfont()`, except that:

- Larger amounts of raster data are supported because the number of raster elements is a long integer instead of a short integer.
- Character movement in the *y*-direction can be specified, allowing vertical displacement of characters.
- Very large bitmaps of characters can be supported, because height and width are short integers, instead of single-byte unsigned chars.
- Characters can be defined with an index beyond the single-byte, 256-character limit.
- Additional characters can be registered with an existing raster font after its initial definition. That is, if a character is currently defined at a specific offset within the raster array, it can be replaced with a new / different character. The same set of operations applied with `defrasterfont()`

results in the removal of the entire font set; thus, a single character can be modified with far less overhead using `deflfont()`.

The ANSI C specification for `deflfont()` is:

```
void deflfont(short n,long nc,Lfontchar chars, long nr, unsigned short raster[])
```

where:

n — is the value to use as the identifier for this raster font. The default font is a fixed-pitch ASCII font with a height of 15, width of 9, character values 0 through 127 defined, and is specified by a font identifier of 0. Font 0 cannot be redefined.

nc — is the number of elements in the *chars* array.

chars — is an array of character description structures of type Lfontchar. One structure is required for each character in the font.

The Lfontchar structure is defined in <*gl/gl.h*> as:

```
typedef struct {
       unsigned long value;
       unsigned long offset;
       short w, h;
       short xoff, yoff;
       short xmove, ymove;
} Lfontchar;
```

value — is the integer value corresponding to this character. When value is encountered by `charstr()` or `lcharstr()` this character is drawn.

offset — is the element number of the raster at which the bitmap data for this character begins. Element numbers start at zero.

w — is the number of columns in the bitmap that contain set bits (character width).

h — is the number of rows in the bitmap of the character (including ascender and descender).

xoff — is the offset in bitmap columns between the start of the character's bitmap and the start of the character.

yoff is the number rows between the character's baseline and the bottom of the bitmap. For characters with descenders this value is a negative number. For characters that rest entirely on the baseline, this value is zero.

xmove is the pixel spacing for the character. This signed value is added to the x-coordinate of the current raster position after the character is drawn.

ymove is the pixel spacing for the character. This signed value is added to the y-coordinate of the current raster position after the character is drawn.

nr is the number of 16-bit integers in raster.

raster is a one-dimensional array containing all the bitmap data for the characters in the font. The bitmap data for each character is a set of consecutive, 16-bit integers, comprising the bit mask for the character from left to right, bottom to top. For characters of width greater than 16, the rows of a bitmap span more than one array element; however, each new row in the character bitmap must start with its own array element.

The number of 16-bit integers per row in a character's bitmap is (w+15)/16. The total number of 16-bit integers in a character's raster definition is h*((w+15)/16).

Bit 15 of each element is left-most when displayed. Bits that are 1 are drawn; bits that are 0 are masked.

Examples of how to use this interface are included in *4Dgifts/examples/intl*.

3.2.2 Selecting the Font

Use `font()` to select the font that the system uses for drawing a text string. Its argument is the font number assigned to the font built by `defrasterfont()` or `deflfont()`. This font remains the current font until you call `font()` to select another font.

The next sample program, *font.c*, defines a font with three characters: a lowercase j, an arrow, and the Greek letter sigma. The j is assigned to the ASCII value of j, and the arrow and sigma are assigned to ASCII values 1 and 2 (written \001 and \002 in the C code). Two sample strings are then written out, the first of which contains only characters that are defined, while the second

contains undefined characters. When characters are not defined, no error occurs, but nothing is drawn for them.

```
/*
 * Define a font with three characters -- a lower-case j,
 * an arrow, and a Greek sigma. Use ASCII values 1 and 2
 * (‘\001’ and ‘\002’) for the arrow and sigma. Use the
 * ASCII value of j (= ‘\152’) for the j character.
 */
#include <gl/gl.h>

#define EXAMPLEFONT 1
#define efont_ht    16
#define efont_nc    127
#define efont_nr(   (sizeof efont_bits)/sizeof(unsigned short))

#define ASSIGN(fontch, of, wi, he, xof, yof, wid) \
    fontch.offset = of; \
    fontch.w = wi; \
    fontch.h = he; \
    fontch.xoff = xof; \
    fontch.yoff = yof; \
    fontch.width = wid

Fontchar efont_chars[efont_nc];
unsigned short efont_bits[] = {
    /* lower-case j */
    0x7000, 0xd800, 0x8c00, 0x0c00, 0x0c00, 0x0c00, 0x0c00,
    0x0c00, 0x0c00, 0x1c00, 0x0000, 0x0000, 0x0c00, 0x0c00,

    /* arrow */
    0x0200, 0x0300, 0x0380, 0xafc0, 0xafe0, 0xaff0, 0xafe0,
    0xafc0, 0x0380, 0x0300, 0x0200,

    /* sigma */
    0xffc0, 0xc0c0, 0x6000, 0x3000, 0x1800, 0x0c00, 0x0600,
    0x0c00, 0x1800, 0x3000, 0x6000, 0xc180, 0xff80,
};

main()
{
    ASSIGN(efont_chars[‘j’], 0, 6, 14, 0, -2, 8);
    ASSIGN(efont_chars[‘\001’], 14, 12, 11, 0, 0, 14);
    ASSIGN(efont_chars[‘\002’], 25, 10, 13, 0, 0, 12);

    prefsize(400, 400);
    winopen("font");
```

```
    color(BLACK);
    clear();
    defrasterfont(EXAMPLEFONT, efont_ht, efont_nc,
                  efont_chars, efont_nr, efont_bits);
    font(EXAMPLEFONT);
    color(RED);
    cmov2i(100, 100);
    charstr("j\001\002\001jj\002");
    cmov2i(100, 84);
    charstr("ajb\001c\002d");
    sleep(10);
    gexit();
    return 0;
}
```

3.2.3 Querying the System for Font Information

The following subroutines return information about the current font: its index, the height of its characters, and the maximum descender for its characters.

Use `getfont()` to query for the index of the current raster font:

```
long getfont(void)
```

Use `getheight()` to query for the maximum height, in pixels, of a character in the current raster font, including ascenders (present in tall characters, such as the letters t and h) and descenders (present in such characters as the letters y and p, which descend below the baseline):

```
long getheight(void)
```

Use `getdescender()` to query for the longest descender in the current font. It returns the number of pixels that the longest descender extends below the baseline:

```
long getdescender(void)
```