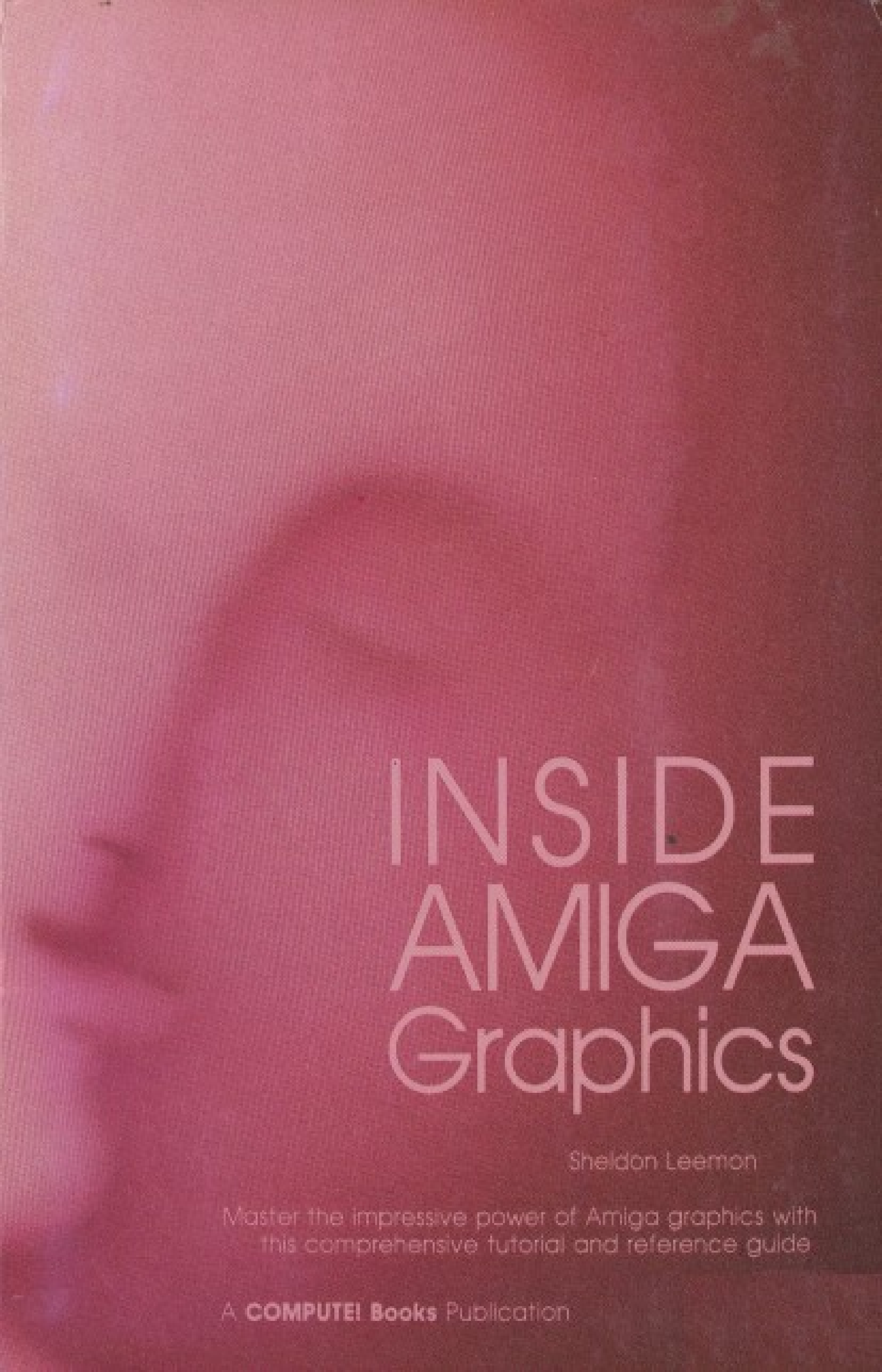


INSIDE AMIGA Graphics

Sheldon Leemon

Master the impressive power of Amiga graphics with this comprehensive tutorial and reference guide

A **COMPUTE! Books** Publication

INSIDE AMIGA Graphics

Sheldon Leemon

COMPUTE!™ Publications, Inc. abc
Part of ABC Consumer Magazines, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-87455-040-8

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is part of ABC Consumer Magazines, Inc., one of the ABC Publishing Companies, and is not associated with any manufacturer of personal computers. Amiga and AmigaDOS are trademarks of Commodore-Amiga, Inc.

Contents

Foreword

Among personal computers, Amiga graphics are unparalleled. Its custom graphics chips, powerful 68000 microprocessor, and true multitasking capability drive this immensely powerful graphics machine.

Learning to utilize the powerful features included in the Amiga's hardware and software, though, can be complex, even frustrating. *Inside Amiga Graphics,* written for the intermediate and advanced programmer, details how you can take advantage of the Amiga's power.

Whether you program in BASIC, C, or machine language, you'll find here the information you need to begin exploiting the advanced power of the Amiga's coprocessor, the *copper;* its high-speed memory-mover chip, the *blitter;* and its software library functions.

Inside Amiga Graphics covers everything from setting up your own display screens and windows to drawing and filling shapes. Scores of program examples written in C and Amiga BASIC make it easy to understand even the most advanced graphics techniques. You'll see how to use the functions in the Graphics and Intuition libraries to create and open custom screens; draw lines and fill shapes; move sprites around the screen; change text fonts; and much more. You can even use the stand-alone routines in your own programs.

The author of *Inside Amiga Graphics,* Sheldon Leemon, is the author of the bestselling *Mapping the Commodore 64* and has coauthored three other COMPUTE! books, including the popular *COMPUTE!'s AmigaDOS Reference Guide.*

Inside Amiga Graphics is for the intermediate to advanced BASIC, C, or machine language programmer. Here in one volume is everything you need to exploit the advanced graphics features of this latest and greatest computer from Commodore.

Introduction

The Amiga is the first personal computer to fully exploit the advanced features of the 68000 microprocessor. The graphics capabilities of the Amiga surpass any microcomputer in its class.

Unlike most other microcomputers which use the microprocessor each time the display changes, the Amiga uses a separate coprocessor, called the *copper,* to control every aspect of the screen display. The copper uses its own program, the *copper list,* and operates entirely independently of the main 68000 microprocessor. In the course of executing its program, the copper can actually write to the graphics hardware registers. Thus, it can change every feature of the display on a line-by-line basis (and in some cases, even a pixel-by-pixel basis).

The display features that the copper controls include a choice of two horizontal and two vertical display resolutions, color resolutions ranging from 2 to 4096 colors (available onscreen at the same time), and a selection of 4096 colors which the computer can display.

Most other microcomputers require display data to be stored in a specific location. Not so for the Amiga. Any portion of the first 512K of memory can be displayed in any order. This capability even includes the ability to display two distinct graphics planes at once, one superimposed upon the other.

Sprites, Too

The Amiga also supports hardware sprites. Sprites are animation objects which are displayed by a mechanism entirely separate from that of the rest of the graphics. They are easily moved, and their shapes may be changed without affecting any other part of the display.

The Amiga has eight hardware sprites, each of which can be up to 16 dots wide, as tall as the screen (or taller), and of three different colors (plus transparent). In addition, pairs of sprites can be "attached" so that they form a single sprite that is of normal size, but can display up to 15 colors (plus transparent).

Although there are only eight sprites, the hardware lets you change their shapes and horizontal positions as you move down the display. Therefore, it is possible to have a single sprite appear as many different objects in one display.

Moving Graphics Data in Memory

In addition to the many and varied features of the display hardware, the Amiga contains a high-speed memory-mover chip known as the *blitter* (short for bit-block image transferrer). The blitter is used to move graphics data around in memory (and, consequently, around the display) almost instantly.

With most computers, in order to move a graphics image, you must use the microprocessor to read the display memory, perform complicated calculations to determine the memory location of each part of the image, perform more calculations to combine the image data with that of the background, and store the new combined image data in memory. With the Amiga, all you have to do is specify the size and location of the object to be moved, and the location of the destination, and let the blitter do the rest.

The Amiga's blitter can not only move a bit-image quickly, but it can also perform sophisticated manipulation of up to three data sources at once. This means that it can combine images in various ways, such as inverting and erasing images. The blitter also performs hardware line drawing and area filling, thus permitting it to draw solid or patterned lines at any angle when it is given the line's origin, direction, and length. And it can fill an enclosed area with color, either during a copy operation or by itself.

Software Support

The Amiga provides a wealth of software support that enhances the capabilities of its powerful hardware. This software support consists of several levels of graphics support routines, ranging from the simplest drawing routines to the Layers library, a system for maintaining multiple displays onscreen at once, to Intuition, a user interface that among other things is able to manage a number of overlapping window displays. The operating system Kernel even contains an entire software animation system, complete with provisions for sequenced drawing and synchronized motion of related objects, collision detection, and more.

Programming Choices

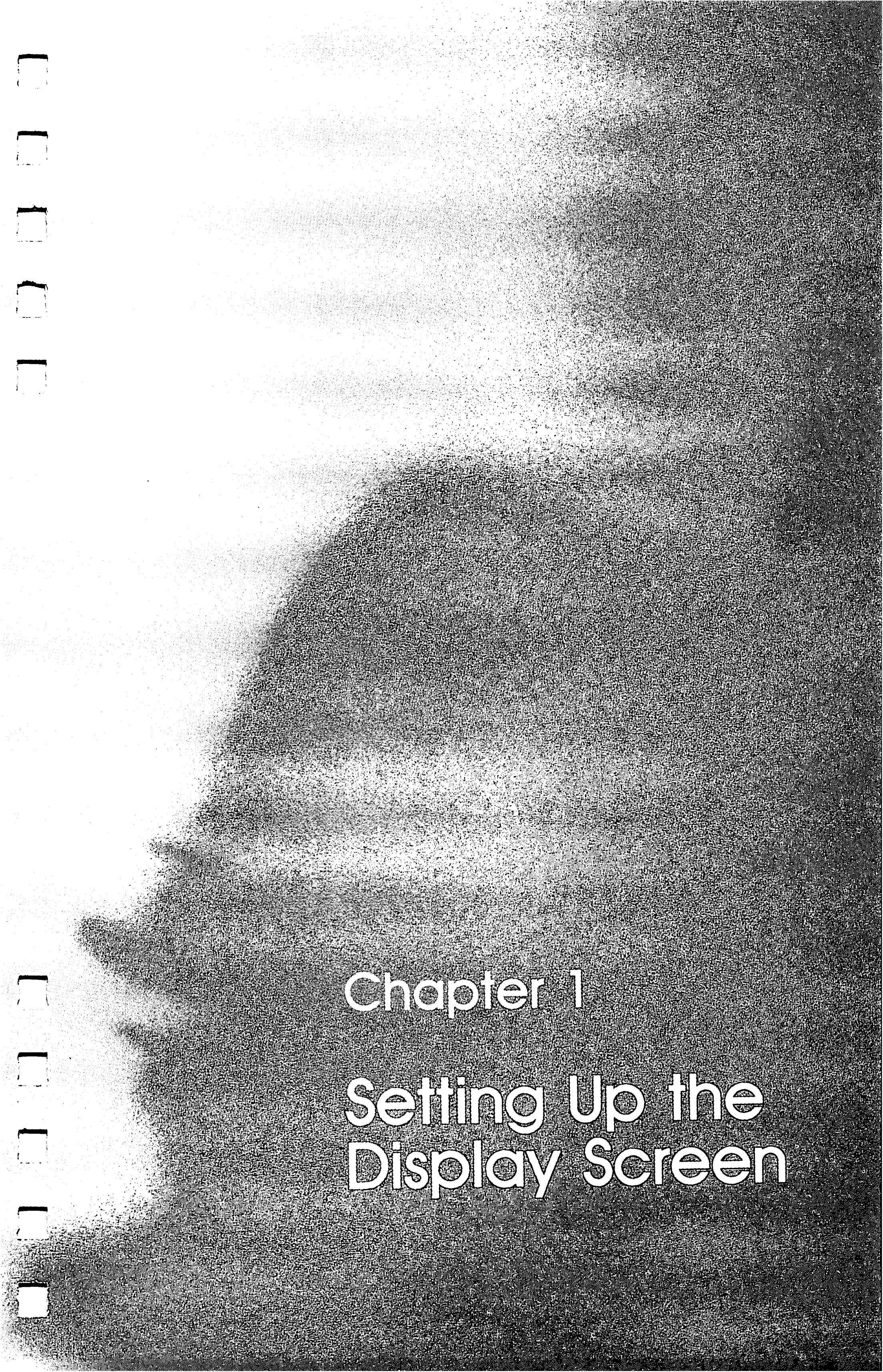
Considering the length of the capsule description of the Amiga's graphics capabilities furnished above, it should be apparent that the subject of Amiga graphics is a broad one indeed. For one thing, the machine can be programmed at every level described above. It is possible to ignore the operating system entirely and program the hardware graphics chips directly. Or, you might decide to ignore the Intuition windowing system and use the most basic graphics routines supplied by the operating system.

Though either of these methods affords an extremely high level of control over the system's graphics, they all require an intimate knowledge of the hardware and operating system software. Moreover, using the lower level graphics features of the machine undermines one of its greatest strengths, its ability to run several programs at once. By their very nature, programs that directly access the display hardware or use the lowest level operating system routines take over all of the Amiga's graphics resources. One of the primary functions of Intuition's windowing system is to provide a common ground in which programs that know nothing of each other's existence can peacefully share the graphics resources of the machine.

Therefore, for the most part, this book deals with programming Amiga graphics at the highest level, that which is compatible with Intuition. Examples are written mainly in the C programming language and in Amiga BASIC. C, although not as widely known as BASIC, is fast becoming the language of choice for producing commercial application programs on microcomputers. The Amiga operating system was designed to interface with C as its primary programming language.

While Amiga BASIC does not offer the execution speed of C, it nonetheless provides convenient access to the most important graphics features of the Amiga. It is extremely valuable as a graphics learning tool for two reasons. First, its own commands closely parallel the features of the operating system routines. If you're familiar with the former, it's easier to learn about the latter. Second, Amiga BASIC also provides a relatively simple method for calling operating system graphics routines directly from BASIC. This means that virtually any of the graphics routines that can be called from C can be executed from BASIC, even though BASIC may not be ideally suited for setting up the data structures that the operating system routines require.

Because of the power and flexibility offered, dealing with the full range of the Amiga's graphics capabilities can be somewhat complex. In ordinary use, however, you may discover that Amiga graphics are actually easier to program than those of other microcomputer systems. This book is intended to show you both sides of the Amiga. The basics of graphic display will be covered in detail, allowing you to discover how simple and convenient these capabilities can be to use. Along the way, we hope to touch on enough of the more esoteric side of the Amiga to satisfy the more adventurous reader as well.



Chapter 1

Setting Up the Display Screen

Setting Up the Display Screen

The Amiga uses a technique called *raster graphics* to produce its display. The picture you see on the screen is composed of a number of horizontal lines, each made up of many tiny dots of color. These lines are created by an electronic video beam, or *raster.*

When producing a display, the electronic beam starts at the top left corner of the screen and moves from left to right, lighting up points on the line as it goes. These dots of color that it lights are known as *pixels* (short for picture elements); they comprise the smallest unit of the display that you can control.

As the beam gets to the right edge of the horizontal line (or *scan line,* as it is called), it shuts off while it moves back to the left edge and down a line. This time period is called the *horizontal blanking interval.* The beam then starts scanning the next line from left to right, repeating the cycle.

When the beam has finished scanning the last line at the bottom of the screen, it is turned off while it moves back up to the top left corner. This period of time is called the *vertical blanking interval,* or *vblank.*

It takes 1/60 second to scan the entire screen from top to bottom. This means that each pixel is drawn on the screen 60 times per second.

The three most fundamental characteristics of the display are the horizontal resolution (the number of dots drawn per line), the vertical resolution (the number of lines per frame), and the color resolution (the number of possible colors for each dot). We will examine the variations that the Amiga provides for each of these display characteristics below.

Horizontal Resolution

Two levels of horizontal resolution are available on the Amiga, high resolution and normal, or low, resolution. In high-resolution mode, the display can have a maximum of 752 dots of color in each line. Such a long line will probably not be displayed completely on most monitors, however. For that reason, the user interface supports a standard line length of 640 dots across in high resolution. In low-resolution mode the standard display width is only 320 pixels across, each dot being twice as wide as in high-resolution mode.

There is no special *text only* display mode on the Amiga. Text is drawn on the graphics screen. Therefore, the horizontal resolution has a dramatic effect on the size in which this text appears on the screen. In high-resolution mode, a maximum of 80 characters can fit on a single line of text when using the system default Topaz 8 font, and 64 characters when using the Topaz 9 font. In low-resolution mode, half as many characters can fit on a line as in high-resolution mode.

Vertical Resolution

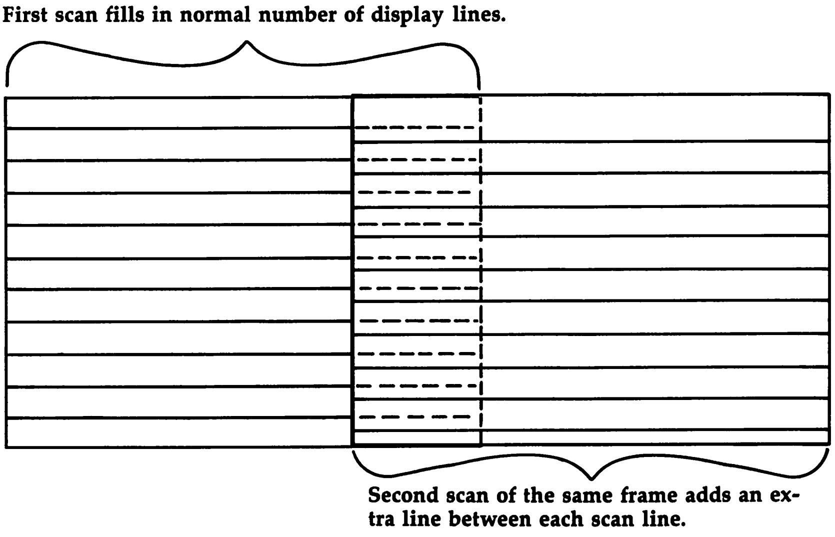
There are two levels of vertical resolution. Here again, the absolute maximum display height is somewhat larger than the one typically used by the system. The standard (noninterlaced) mode can provide up to 242 lines vertically. Typically, however, the user interface provides a display height of 200 lines in noninterlaced mode.

Interlaced mode provides 400 lines of vertical resolution by means of a hardware trick. As explained above, the display is formed by a beam of light that starts at the top of the screen and scans one line at a time from right to left until it gets to the bottom of the screen. Sixty complete scans occur every second, and each scan provides a complete picture. In interlaced mode, each scan provides only half the picture. After every even frame is drawn, the beam of light is moved down half a line so that the lines of the odd frame are interlaced between the lines left by the even scans. This allows twice the vertical resolution in the same amount of space (see Figure 1-1).

While interlaced mode doubles the vertical resolution, it cuts the *refresh rate* (the number of complete pictures formed per second) in half, since it now takes two passes to update the entire display. One result is that in interlaced mode, the display tends to flicker, or vibrate. The amount of vibration depends to a large extent on the type of monitor used.

When the electron beam strikes the face of the display tube, it lights up a dot onscreen, but that dot remains lit only for a short period of time. So, while the picture may appear to be solid if it is redrawn 60 times per second, it may seem less so if redrawn only 30 times per second. There are special types of monitors whose picture tubes contain high-persistence phosphors that stay lit for a longer period of time. But these monitors are not really suited for general use with the Amiga since their picture does not change quickly enough. On high-persistence monitors, moving objects (such as the mouse pointer) appear to leave trails of light behind them as they move.

Figure 1-1. Interlacing Mode


First scan fills in normal number of display lines.
Second scan of the same frame adds an extra line between each scan line.

The colors being displayed also affect the amount of flicker an interlaced display produces. An interlaced screen showing black text on a white background is almost unviewable. The best results are generally obtained when using color combinations that have as little contrast in brightness levels as possible. It also helps considerably to turn down the contrast adjustment on your monitor until the flickering fades. Even if you stick to optimum color combinations and monitor settings, however, interlaced mode generally will not produce very good results on most monitors.

Color Resolution

The Amiga offers flexibility in its choice of color resolution, which is the number of colors that can be displayed onscreen at any one time. The color resolution is determined by the number of blocks of display memory, known as *bit planes,* that are allocated to the display. This is also sometimes referred to as the display depth. In order to explain how display memory is used to make up the screen display, we must first briefly review binary arithmetic.

Binary. In the binary (base 2) numbering system, there are only two digits, zero and one. It may seem difficult to count very high with only two digits to work with, but it works the same way as in the decimal system. After you've counted from zero to one, you have to add another digit to the left. In the decimal system, each column to the left increases in value by a power of 10 (ten's place, hundred's place, and so forth). In the binary system, each column to the left increases in value by a power of 2. So a one in the second digit to the left has a value of 2, the next digit to the left has a value of 4, and so on. For instance, with two binary digits you can count from 0 to 3:

00 = 0
01 = 1
10 = 2 (2+0)
11 = 3 (2+1)

Any decimal number can be represented by a series of zeros and ones. Converting numbers to binary is very useful when working with computers, because the zero and one can represent two opposite logic states like true and false. Or, to bring the subject back to graphics, they can portray the two states of a pixel, display on or display off. And in practice, that is how the display pattern is related to the numbers stored in display memory. Each dot that is turned on corresponds to a binary digit (or bit) that is set to one, and each dot that is turned off corresponds to a bit that is reset to zero.

Bit planes. A bit plane is the area of memory that stores information concerning which color is to be shown at each dot position of the screen display. It is organized so that the first memory location contains information about the leftmost eight dots on the top line, the second location contains the display pattern for the next eight dots, and so on.

In low-resolution mode, there are 320 pixels per line, so it takes 40 bytes of memory, each holding eight bits, to represent the display for one line. And since there are 200 lines in noninterlaced mode, it takes 8000 bytes of memory (200 × 40) to make up one complete bit plane. In high-resolution or interlaced mode, there are twice as many pixels; thus, it takes twice as much memory (16,000 bytes) to hold the display data. And if both high-resolution and interlaced modes are used at once, there are four times as many dots, and four times as much memory (32,000 bytes) is needed to hold the display data.

A single bit plane can hold the information for a two-color display. But if you want to represent more than two colors (background and foreground), you need more than one bit to represent a single dot on the screen. On the Amiga, the maximum number of colors available is increased by adding more bit planes. When using more than one bit plane, the digits from corresponding spots on the different planes are grouped together as one number. For example, if there are two bit planes, the first digit from plane 0 is grouped together with the first digit of plane 1 to form a two-digit binary number. As we have seen from the illustration above, two binary digits can be combined in four different ways to represent four numbers from 0 to 3. So, using two bit planes allows a maximum of four color combinations. Each additional bit plane increases the number of colors that can be displayed by a factor of two (see Figure 1-2).

Hardware color registers. The numbers which are formed from the bits from the various bit planes do not correspond to actual colors. These numbers correspond instead to special memory locations known as *hardware color registers.* The color registers may be thought of as a set of 32 pens, each of which may be filled with colored "ink" in any of the 4096 shades that can be displayed on the Amiga.

Register 0 always holds what is normally thought of as the background color; any dot position whose display memory holds the number 0 displays this color. When you wish to use another color to draw a line or a point, you put the number of a color register into the bit planes of display memory (or, as is more likely, you have the operating system drawing routines do it for you). The color whose number is currently contained in that color register is the color that appears onscreen.

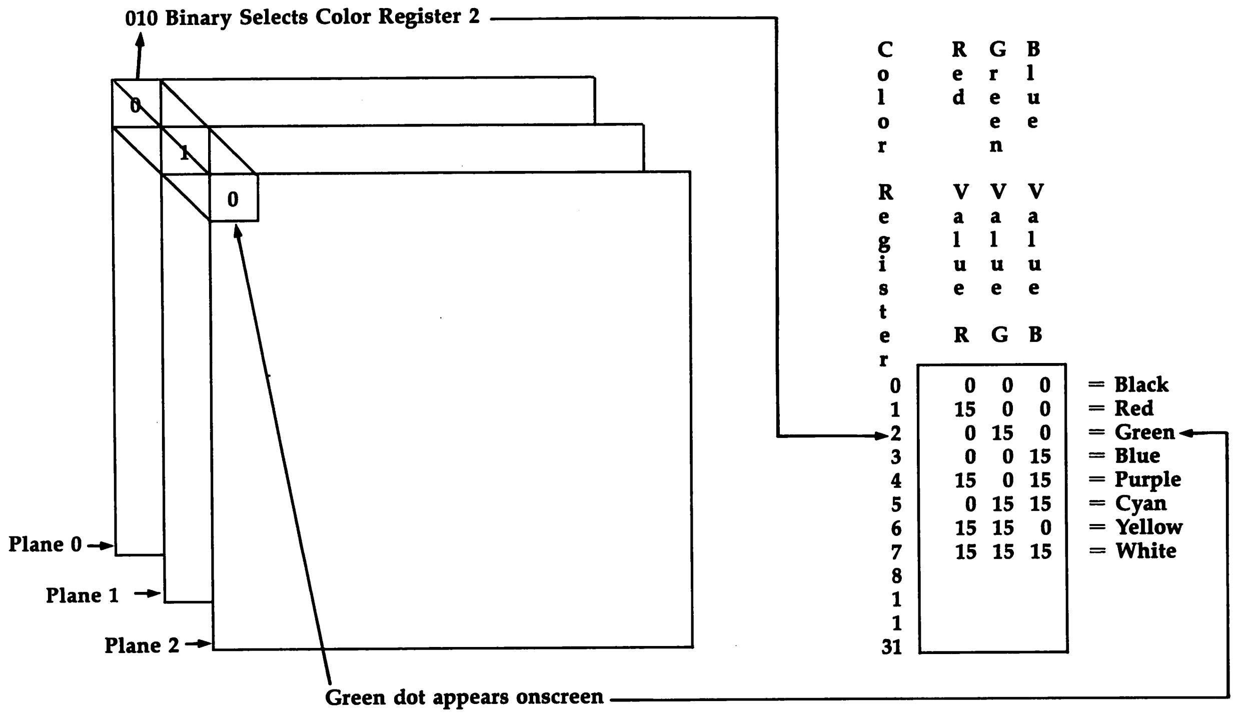
Unlike ink, however, the color of a dot drawn onscreen can change after you have drawn it. When the display memory for a screen dot holds the number of a particular pen, that dot displays whatever color is in the pen at any given moment, not the color that was in the pen at the time the dot was drawn. This means that if you use pen 1 (hardware register 1) to draw a line, and that pen contains the color red, the line will be red. But if you change the color in pen 1 to green after you've drawn the line, the line you drew and everything else onscreen that was drawn with pen 1 will instantly become green. Figure 1-2 shows the correspondence between bit planes, color registers, and colors.

Memory Usage

The maximum number of bit planes that can be used (and the number of colors available) depends on the horizontal resolution of the screen. In high-resolution mode, up to four bit planes can be used for a total of 16 colors on the screen at once. Normally, in low-resolution mode, up to five bit planes may be used for a maximum of 32 colors. There are certain special graphics modes that can be used in conjunction with low-resolution mode which require six bit planes. These modes will be discussed separately later.

In determining how many bit planes to use, there are a number of tradeoffs to consider. Each bit plane consumes its share of valuable RAM. As stated above, display memory requirements range from 8000 bytes per plane in low-resolution mode to 32,000 per plane for high-resolution, interlaced mode.

Figure 1-2. Relationship of Bit-Plane Values to Color


010 Binary Selects Color Register 2
0
1
0
Plane 0
Plane 1
Plane 2
Green dot appears onscreen

Color Register	Red Value R	Green Value G	Blue Value B	
0	0	0	0	= Black
1	15	0	0	= Red
2	0	15	0	= Green
3	0	0	15	= Blue
4	15	0	15	= Purple
5	0	15	15	= Cyan
6	15	15	0	= Yellow
7	15	15	15	= White
8				
1				
1				
31				

Therefore, a 640 × 400 display that has four bit planes (allowing up to 16 colors) uses almost 128K of memory for the display alone. In a 256K system, such a display would consume virtually all free RAM.

Besides using a lot of memory, high-resolution displays that use a lot of bit planes can slow down the microprocessor as well. A good illustration of this can be heard when you use the built-in speech synthesis while a high-resolution screen with four bit planes is being displayed. The voice sounds very scratchy and rough. The job of updating the display takes so much time that the display chips must preempt some of the time in which the processor has access to user memory. You can generally avoid such conflicts by keeping to displays that require no more memory than a four-color high-resolution (noninterlaced) screen.

Both of these concerns affect the capabilities of the Amiga as a multitasking machine. Obviously, it is going to be much more difficult to run other programs along with yours if yours leaves no free memory or burdens the processor unduly. This is not to say that you should never use the more memory-consumptive display modes. Rather, you should keep in mind the degree to which they will hamper other applications and use them more sparingly than if you expected to have the entire machine at the disposal of your application.

Table 1-1. Graphics Memory Requirements

	Bytes Required				
Bit	Noninterlacing		Interlacing		Number of
Planes	Lo-res	Hi-res	Lo-res	Hi-res	Colors
1	8000	16,000	16,000	32,000	2
2	16,000	32,000	32,000	64,000	4
3	24,000	48,000	48,000	96,000	8
4	32,000	64,000	64,000	128,000	16
5	40,000	NA	80,000	NA	32

Viewports and Screens

The Amiga's display coprocessor, the copper, allows a change of all the characteristics of the display on a line-by-line basis. This means that segments of differing horizontal, vertical, and color resolution may appear on the screen at once.

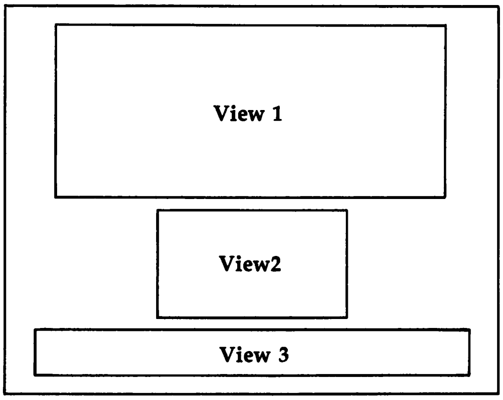
The operating system makes the changes to the display mode when the electron beam has reached the right edge of the screen and is turned off while it moves back to the left edge. This means that segments of differing display modes are confined to horizontal stripes that extend across the complete width of the screen.

While it is technically possible to change display modes in the middle of a horizontal line, making the change while the display is being drawn can result in unpredictable and unsightly visual effects and is therefore impractical. In effect, you may not have an area of high-resolution display and an area of low-resolution side by side.

View and Viewports

In a multitasking environment, it's impossible for each program to have direct control over the copper and, therefore, over the entire display. The operating system provides a method by which each application can decide how it wants its display to look without having to take over the whole display. In this scheme, the overall display is known as the *view.*

Figure 1-3. Possible Division of Display into Views

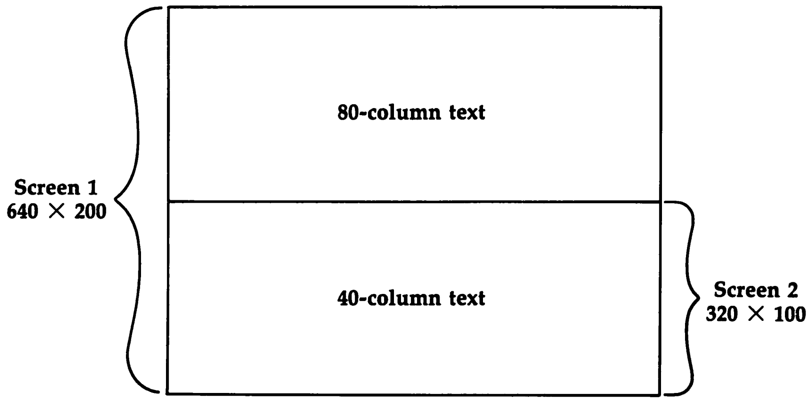


The view can be divided into one or more *viewports.* Each viewport defines its own display resolutions, colors, and special features. Since the display can be changed only at the end of a horizontal line, each viewport must form a complete horizontal segment. These segments are stacked one on top of the other to form the view. At least one blank scan line is used to separate one viewport from another.

Screens

Intuition, the Amiga's user interface, implements viewports through the use of data structures known as *screens.* A screen has a few more limitations than a viewport. A screen must be as wide as the standard display, either 640 pixels (high resolution) or 320 pixels (low resolution). Though it can be any number of lines tall, if it is shorter than the display, it must sit at the bottom of the display, not at the top or middle. There can be no gap between the bottom of the screen and the bottom of the display. If you wish to stack multiple displays onscreen, you must have overlapping screens, with the tallest screen in back and the shortest in front. Intuition maintains two blank scan lines at the place where two screens meet.

Figure 1-4. Division of Display into Intuition Screens



Each screen must extend down to the bottom of the display and be as wide as the display width.

Screens come with two of the system gadgets attached, the drag bar and the depth arrangers. The drag bar allows the user to move the whole screen up and down by clicking on the bar, then holding down the left mouse button and dragging the mouse. The depth arrangers allow the user to bring the screen to the front of the display or send it to the back by clicking on the light or dark squares. In addition, a title may appear in the bar at the top of the screen.

The display screen that appears when you turn on the computer is known as the Workbench screen. This is the screen used by both the Workbench and the Command Line Interpreter (CLI). It is a high-resolution, noninterlaced display, which is two bit planes deep, providing a maximum of four screen colors. The actual color selections that appear on this screen are those set by the Preferences program. If none has been set, the Workbench screen uses the system default colors of blue, white, black, and orange.

Application programs are free to use the Workbench screen. The windowing system described below was designed to allow several overlapping windows, each potentially belonging to a different program, to coexist on one screen.

For instance, when you start Amiga BASIC, the BASIC interpreter does not open its own screen. Rather, the output and list windows are drawn on the Workbench screen. Unless you specify otherwise, all of the output from a BASIC program is displayed in a window which shares the display characteristics of the Workbench screen.

There are several advantages to using the Workbench screen for your programs. It's convenient to use because you don't have to do anything to set it up—it's already there. Using another screen means that you have to allocate memory for that screen in addition to the display memory used for the Workbench screen (which will be there in any case unless the application is able, under special circumstances, to close it). It allows easy access to the Workbench or CLI—you just use the depth arrangement buttons in the corner of the window to send your window behind the Workbench. Finally, it presents a reasonably good tradeoff of system resources. It has high resolution for 80-column text and two bit planes for a touch of color, but not so much color as to hog most of the system's memory.

Opening Custom Screens

Despite the versatility of the Workbench screen, there will be times when you'll want to custom-tailor the display characteristics to suit your needs. This means opening a custom screen. From C or machine language, you use the Intuition library routine OpenScreen to set up a new screen (we'll discuss the BASIC commands later on). This call takes the form

```
Screen = OpenScreen(NewScreen);
 (d0)                (a0)
```

This means that when you call the OpenScreen routine, you must furnish the address of a data structure known as a NewScreen structure, and the routine, if successful, opens the screen and returns the address of the Screen data structure in the d0 register. If it is not successful, it returns the number 0 instead.

Before you can call this routine, however, there are two preparatory steps that you must take. First, you must set up the data structures required by the OpenScreen routine. And next, you must use the OpenLibrary command to prepare the Intuition library for use, if it has not already been opened.

The first step is the more involved. The OpenScreen routine requires a pointer to a block of data known as a NewScreen structure. This structure contains 14 different pieces of information about the screen that you want created. Here is the C language definition of the NewScreen data structure:

```
struct          NewScreen
    {
    SHORT          LeftEdge, TopEdge
    SHORT          Width, Height, Depth;
    UBYTE          DetailPen, BlockPen;
    USHORT         ViewModes, Type;
    struct         TextAttr    *Font;
    UBYTE          *DefaultTitle;
    struct         Gadget    *Gadgets;
    struct         Bitmap    *CustomBitMap;
    };
```

The explanation of these variables is as follows:

LeftEdge and TopEdge. These describe the top corner of the screen. In the current version of Intuition, a screen must be as wide as the display, so LeftEdge should always be set to zero. TopEdge specifies the scan line where you want the screen to start. For purposes of describing the location of a particular scan line, we say that the top line of the display is line 0, and line numbers increase as we move down to the bottom line, whose number is line 199 or 399, depending on whether the display is noninterlaced or interlaced.

Width, Height, and Depth. The width should be set to the full display width, 640 for high resolution or 320 for low resolution. Since all screens must go down to the bottom of the display, Height should be set to the display height minus TopEdge. For example, if TopEdge is set to 50, Height should be set to 150 for a noninterlaced display. Depth specifies the number of bit planes, from 1 to 6. The number of planes determines the number of possible colors, as explained above.

DetailPen. DetailPen specifies the color register to be used for details, such as the text characters that appear in the title bar.

BlockPen. BlockPen specifies the color register to be used for filled areas, such as the title bar background.

ViewModes. This flag lets you set the various display modes which follow:

HIRES. If this flag is set, horizontal resolution is 640 pixels across. Otherwise, the horizontal resolution is 320 pixels.

INTERLACE. If selected, the vertical resolution is 400 lines instead of the default 200 lines.

SPRITES. Turns on sprite DMA and allocates color map memory for sprite color registers so that sprites may be included in the display. Even if you omit this flag, it may be possible to use sprites in the display, since the mouse pointer is a sprite, and therefore sprite DMA must always be on in order to display the pointer.

DUALPF. This is used to set up a special mode in which there are two overlapping display fields (called playfields).

HAM. This flag enables the special Hold and Modify display mode, which can be invoked only from a low-resolution screen that is six bit planes deep. This mode will be discussed more thoroughly in the "Advanced Topics" chapter.

EXTRA_HALFBRITE. This flag enables the special Halfbrite display mode, which can be invoked only from a low-resolution screen that is six bit planes deep. This mode will also be discussed more thoroughly in the "Advanced Topics" chapter.

Type. This should be set to CUSTOMSCREEN. If you wish to set up your own custom bitmap so that you control where the display memory for this screen is, you should add the CUSTOMBITMAP flag here also as well as supplying the address of the bitmap in the CustomBitMap field described below.

Font. A pointer to the Intuition data structure for the text font that should be used as the default in this screen. The format of the TextAttr structure is discussed in the chapter on text. If you wish to use the default system font, you may set this value to zero.

DefaultTitle. This is a pointer to the address of a string of ASCII text characters, ending with an ASCII 0. This text is displayed in the screen's title bar. If you don't want a title, set this value to zero.

Gadgets. This value points to the address of the first gadget in a linked list of your own custom screen gadgets (the drag bar and depth arrangers appear regardless of this setting). If there are no custom gadgets, set this to zero.

CustomBitMap. If you wish to specify the display memory used for this screen, this value should point to a BitMap structure that describes this display memory area. If you wish Intuition to allocate the display memory, set this to zero. Custom bitmaps are discussed in the "Advanced Topics" chapter. But rest assured that in almost every case it is sufficient to use the display memory that Intuition allocates.

Once the data structure is set up, the next step is to open the Intuition library, if it hasn't been opened already. The Amiga was designed so that no system routine has to start at a fixed memory location. Rather, each group of operating system functions is set up in the format known as a library. Each routine within a library starts at a fixed offset from the beginning of the library, but in order to find the address of the library itself, you must call the Exec library function OpenLibrary. Exec is the only library whose address is stored in a fixed place in memory. Its address is always in memory location 4, also known as AbsExecBase.

The OpenLibrary call takes this form:

```
library_base_address = OpenLibrary("name.library",version);
        (d0)                                (a1)         (d0)
```

You must pass the OpenLibrary routine a pointer to the name of the library and the library version number. The name must be a string of lowercase ASCII characters that end with an ASCII 0. For example, the Intuition library would be referred to as "intuition.library". The version number corresponds to the version of the operating system that you require. The program will run if the version of Kickstart used is the one specified or any later version. The current internal version number can be found by using the Version menu item on the Special menu of the Workbench. If it does not matter what version is used, use the number 0, which allows the OpenLibrary routine to succeed no matter what version of Kickstart is used.

If the OpenLibrary call is successful, it returns the base address of the library in register d0. If it is not successful, it returns a zero in that register.

Machine language programs explicitly use the pointer returned by OpenLibrary for the purpose of making indirect calls through the library base vector. Thus, calling a library routine is a two-step process. The OpenLibrary call is used to find the base address of the library. That address is saved and used to make indirect calls to the library routines that start at fixed offsets from the library base.

With the Amiga *Macro Assembler*, you can use the symbolic names for these offsets provided by the Amiga.lib library file. These symbolic names use the name of the routine preceded by the characters _LVO (Library Vector Offset). So, after you have used OpenLibrary to get with the base address for the Intuition library and moved that address from register d0 to register a6, you could call OpenScreen like this:

```
JSR _LVOOpenScreen(A6)
```

You would of course have to first set up the NewScreen structure and store a pointer to it in register a0. If the OpenScreen call is successful, the pointer to the Screen data structure is returned in the d0 register.

A Machine Language Example

The sample machine language program, Program 1-1, demonstrates the process of opening a new screen. It opens a low-resolution screen that covers the full display, waits a few seconds, and closes it.

A C Language Example

C programs must also use the OpenLibrary routine before accessing functions found in the library. But with C, it is not necessary to explicitly use the library base pointer to call system library routines. You may just call the library routines as you would any external function, and the C compiler takes care of the process of jumping through the correct offset from the library base. Here is a typical example of opening a library from C:

```
IntuitionBase = (struct IntuitionBase *)
     OpenLibrary("intuition.library",31)
```

The cast (struct IntuitionBase *) is used to let the *Lattice C Compiler* know that the value returned is of the proper type. Once the library is open, you could use the following C code to open a new screen:

```
Screen = (struct Screen *)OpenScreen (NewScreen);
 (d0)                                  (a0)
```

If successful, the call opens the new screen and returns a pointer to the Screen data structure. That data structure contains the information from the NewScreen data, plus a lot more. It is possible to examine this information to learn things about the screen. For example, structure member Screen.TopEdge contains the current vertical coordinate of the top of the screen. This can be used to learn where a user has dragged the screen. Full details about the information stored in the Screen structure appear in the "Intuition/Intuition.h" include file.

Program 1-1. Opening a New Screen, ML Example

```
   XREF    _AbsExecBase
   XREF    _LVOOpenLibrary
   XREF    _LVOCloseLibrary
   XREF    _LVOOpenScreen
   XREF    _LVOCloseScreen

**** open the Intuition Library ****

   movea.l  #IntuitionName,al     ;ask for 'intuition.library'
   move.l   #31,dØ                ;version 31 or later (not used in v1.Ø)
   movea.l  _AbsExecBase,a6       ;get pointer to Exec library
   jsr      _LVOOpenLibrary(a6)   ;jump through offset to OpenLibrary
   move.l   dØ,IntuitionLibrary
      ;save pointer to Intuition library base address
   beq      Abort                 ;if pointer not found, abort

**** open our Screen  *****

   movea.l  #NewCustScreen,aØ     ;pointer to NewWindow structure in aØ
   movea.l  IntuitionLibrary,a6   ;pointer to Intuition library base in a6
   jsr      _LVOOpenScreen(a6)    ;jump through offset to OpenWindow
   move.l   dØ,CustScr            ;save pointer to Window structure
   beq      Abort                 ;if no pointer returned, abort

***** wait a few seconds  ****

   move.l   #$fffff,dØ

loop:
   subi.l   #1,dØ
   bne      loop
```

```
**** close the Screen ****

   movea.l  CustScr,aØ             ;set Screen pointer parameter for call
   movea.l  IntuitionLibrary,a6    ;set pointer to Intuition Library
   jsr      _LVOCloseScreen(a6)    ;and close the window

   movea.l  IntuitionLibrary,al    ;set pointer to intuition library
   movea.l  _AbsExecBase,a6        ;get pointer to Exec library
   jsr      _LVOCloseLibrary(a6)   ;jump through offset to CloseLibrary

**** quit immediately if library won't open ****

Abort:
   clr.l    dØ                     ;return code in dØ
   rts

**** here's our data ****

   SECTION data,DATA

IntuitionName:
   dc.b   'intuition.library',Ø

STitle:
   dc.b   'Custom Low-Res Screen',Ø      ;text of window title

**** the NewScreen structure ****

NewCustScreen:
   dc.w   Ø             ;LeftEdge
   dc.w   Ø             ;TopEdge
   dc.w   32Ø           ;Width
   dc.w   2ØØ           ;Height
```

```
    dc.w   3            ;Depth
    dc.b   3            ;DetailPen
    dc.b   1            ;BlockPen
    dc.w   Ø            ;special display modes
    dc.w   $Øf          ;Screen type--CUSTOMSCREEN
    dc.l   Ø            ;pointer to custom font structure
    dc.l   STitle       ;Title -- ptr to Screen title text
    dc.l   Ø            ;ptr to Screen gadgets
    dc.l   Ø            ;BitMap -- ptr to custom BitMap

    SECTION mem,BSS

IntuitionLibrary:
    ds.l   1           ;place to store Intuition library base address

CustScr:
    ds.l   1           ;place to keep pointer to Window structure

    END
```

Program 1-2 is a C program example that opens a new screen. First, if either the OpenLibrary or OpenScreen routine fails, it returns a value of zero. Your program should check for this event and abort if either call does not work.

Second, the example programs use the CloseLibrary function before they end. OpenLibrary is used not only to find the pointer to the base address of a system library, but is also used to load nonresident libraries from disk. Such nonresident libraries take up valuable RAM, so when you're through using them, it is advantageous to notify the system that the library code is no longer needed and that the memory space it occupies may be reused. Of course, resident libraries like the Intuition and Graphics libraries will not be unloaded if they are no longer in use, since they are in Writeable Control Store (WCS) rather than RAM. Therefore, it is not strictly necessary to close them when you are finished using them. Nonetheless, it's a good habit to close libraries you will no longer use. Finally, the CloseScreen routine is used to close the screen when the program is finished. This is necessary in order to clear it from the display and to free the memory used by the screen.

BASIC Custom Screens

Amiga BASIC also lets you set up a new screen with the SCREEN statement.

SCREEN screen_number, width, height, depth, mode

The BASIC statement gives you considerably fewer choices to make than does the Intuition library routine. The first value you must supply, screen_number, is a number from 1 to 4 which is used to identify the screen for the purpose of closing the screen when you are done with it and for opening windows (see Chapter 2 for more information on windows).

The width and height values correspond to those passed to OpenScreen in the NewScreen data structure. Width should always be set to 320 for a low-resolution screen or 640 for high-resolution, just as you do when using the operating system routines. Setting a lesser width confuses the display. Height is a different matter. Intuition does allow you to set up a screen that is shorter than the full display height. But because of a flaw in the way that the first version of Amiga BASIC sets up short screens, you must take some additional steps if you want to create a noninterlaced screen that is fewer than 200 lines tall or an interlaced screen of fewer than 400 lines. These steps are detailed a little later on.

The depth value is a number from 1 to 5 that specifies the number of bit planes to use. This, in turn, determines the number of different colors available at any one time. As previously discussed, each additional plane doubles the number of colors available. The number of colors available for each depth value is as follows:

Depth	Number of Colors
1	2
2	4
3	8
4	16
5	32

In high-resolution mode, the maximum number of bit planes that can be used is only four. Low-resolution screens can use five bit planes.

The final value to be specified in the SCREEN statement is the mode. There are four different display modes available from Amiga BASIC. Here are the meanings of each of the allowable mode values:

1	Low resolution, noninterlaced
2	High resolution, noninterlaced
3	Low resolution, interlaced
4	High resolution, interlaced

For example, to set up a low-resolution, noninterlaced screen that displays up to eight colors simultaneously, you would use the statement

```
SCREEN 1,320,200,3,1
```

When you open a screen with the SCREEN statement, BASIC allocates display memory for it in the same way that the Intuition library routine OpenLibrary does. When you are

Program 1-2. Opening a New Screen, C Example

```
/* Include the definitions that we need */

#include <exec/types.h>
#include <intuition/intuition.h>

/* Structures needed for libraries */

struct IntuitionBase *IntuitionBase;

/* Structures required for graphics */

struct Screen *CustScr;


/* ******* Pre-initialized Intuition Structures ******** */

   struct NewScreen NewCustScr =
      {
      Ø,Ø,           /* LeftEdge (always=Ø),TopEdge */
      32Ø,2ØØ,3,        /* Width, Height, Depth */
      3,1,           /* DetailPen and BlockPen */
      NULL,            /* special display modes */
      CUSTOMSCREEN,       /* Screen Type */
      NULL,           /* Pointer to Custom font struct*/
      "Low-Res Screen",   /* Pointer to title text */
      NULL,           /* Pointer to Screen Gadgets */
      NULL,           /* Pointer to CustomBitMap */
      };

/************** Program Begins Here ***************** */
```

```
main()
{

/* Open the Intuition library
 * Get pointer to WCS routines, and if = Ø,
 * library isn't available.
 */

   IntuitionBase = (struct IntuitionBase *)
         OpenLibrary("intuition.library",LIBRARY_VERSION);
   if (IntuitionBase == NULL) exit(FALSE);


/* Open Screen.  If it's address = Ø, it wasn't opened. */

   if ((CustScr =
 (struct Screen *)OpenScreen(&NewCustScr)) == NULL)
      exit(FALSE);

Delay(3ØØ);

   CloseScreen(CustScr);
    CloseLibrary(IntuitionBase);

}
```

finished with the screen, you should free up that display memory with the SCREEN CLOSE statement, which uses this syntax:

SCREEN CLOSE screen_number

In the example above, you could close the screen with

SCREEN CLOSE 1

Creating a Short Screen in BASIC

As stated above, Amiga BASIC does not set up short screens correctly. Since a screen that is less than full height must always sit at the bottom of the display, the TopEdge setting of the NewScreen structure should be equal to the full height of the display minus the height of the screen.

BASIC apparently always sets the TopEdge value to zero, however, so that the screen starts at the top of the display. Therefore, when you set the height for a value less than the full display height, you create a screen that ends before the bottom of this display. As a result, the display becomes confused, and random "garbage" fills the bottom of the display. When you drag the screen down by using the drag bar, however, everything is sorted out, and the screen takes its proper position at the bottom of the display.

Therefore, after you create a short screen using Amiga BASIC, you must move the screen down to its proper position under program control (since you can't very well depend on users to realize that they have to pull the drag bar to straighten out the display). BASIC has no direct instruction to perform this task, but the Intuition library does. And BASIC does provide a way for you to call such routines.

The way that you call operating system library functions from BASIC is by using the LIBRARY statement

LIBRARY *"name.library"*

where *name.library* represents the name of the library, in the same format used by the OpenLibrary routine (lowercase only).

BASIC uses this information to call OpenLibrary, which provides the library base address. In order to use the library base address to call the library routines, BASIC still needs a way of finding out the address offsets of these routines and the registers used to pass the data values they need. For this purpose, it uses a special data file that *must be located in the current disk directory when the program is run.* The format of this file is described in detail in Appendix F of the *Amiga BASIC* manual. Its name must be the same as that of the library, with the characters *.bmap* (for binary map) appended to it.

Files named graphics.bmap and dos.bmap are provided in the BasicDemos directory of the BASIC disk; they allow BASIC to access the Graphics and DOS libraries.

To use other libraries, such as the Intuition library, you must create a .bmap file for that library. This can be accomplished in one of two ways. First, you can obtain what is known as an .fd file. These are text files that are included with the Amiga *Macro Assembler* (and possibly the *Lattice C Compiler* as well). They describe the offsets and data registers for each routine in the library and can be converted to .bmap form by the "ConvertFD" program which is found in the BasicDemos directory of the BASIC distribution disk. The other method is to use the BASIC file commands to create a .bmap file from the program that uses it. This is the approach that we'll take for purposes of demonstrating library routines. Of course, if you already have these .bmap files, you can omit this portion of the demo programs.

Once you have opened the library with the LIBRARY statement, you may use any routine described in the .bmap file via the CALL statement. The syntax for this statement is

CALL Routine_name (value1,value2,...)

It is possible also to use the alternative form:

Routine_name value1,value2,...

For the purposes of this statement, the name of the routine is used as the name of a variable that holds the actual address of the routine. BASIC computes this address from the library base address and the address offset given in the .bmap file, and then stores it in the variable Routine_name.

Since most code addresses are 24 bits long, the variable Routine_name must be of the long-integer or double-precision type. If it is not, a *Type mismatch* error occurs when you try to call the routine. You can make sure that the variable is of the correct type by using the DEFLNG or DEFDBL statement, or by adding the correct symbol to the end of the name (Routine_name& or Routine_name#).

The values in parentheses (value1, value2, and so on) are the data values used by the routine. You must make sure that these values also are of the correct type. Usually, it is safe to make all variables involved in a library call into long integers by adding the ampersand (&) to the variable name.

The Intuition library routine used to move a screen is named, appropriately enough, MoveScreen. The format for this routine is

```
MoveScreen (Screen, DeltaX, DeltaY);
             (a0)    (d0)    (d1)
```

where Screen is a pointer to the Screen data structure, and DeltaX and DeltaY are the increments by which you wish to move the screen. DeltaX is only provided for upward compatibility, since currently a screen must fill the entire horizontal display and therefore cannot be moved horizontally. This value should be set to zero for purposes of clarity (though it is currently ignored). The DeltaY value should be set for the number of scan lines that you wish to move the screen, either downward (a positive number) or upward (a negative number).

BASIC does not provide a direct way to learn the address of the Screen data structure. It does, however, provide a way to discover the address of a Window data structure. Among the items provided in this data structure is the address of the window's screen. The BASIC function WINDOW(7) returns the address of the Window data structure. This address of the Screen data structure for the screen on which the window resides can be found at address WINDOW(7)+54. Therefore, you can find the screen data address by opening a window on the screen and taking the value at PEEKL(WINDOW(7)+54).

Though this method works, it does not represent the best in programming practices. Normally, it is not a good idea to access data structures using absolute offsets, because these structures could possibly change in future versions of the computer or future versions of the operating system and render your program incompatible. Therefore, while we demonstrate this technique as one way of correctly generating short screens using the first version of Amiga BASIC, we caution you to use such methods sparingly and only in cases where better techniques do not exist.

Once the address of the Screen data structure is known, use the MoveScreen routine to move a screen 100 scan lines down the display with the statement

CALL MoveScreen&(Screen&,0,100)

Program 1-3 shows the whole process involved in setting up a short screen from BASIC.

Program 1-3. Setting Up a Short Screen from BASIC

```
DEFLNG a-z
GOSUB Initlib  'initialize Intuition library


SCREEN 1,320,100,3,1      'open short screen
WINDOW 2,"Short Screen Window",(0,15)-(297,86),,1
PALETTE 0,0,0,0           'change screen colors
PALETTE 2,0,.3,.6
s = PEEKL(WINDOW(7)+46)   'find Screen address
CALL MoveScreen(s,0,100)  'and move it down

PRINT
PRINT "Hello, out there"

x = INT (TIMER)+5    'wait a few seconds
WHILE (TIMER <x)
WEND

WINDOW CLOSE 2       'close window,
SCREEN CLOSE 1       'Screen
LIBRARY CLOSE        'and all libraries

END

Initlib:
  CHDIR "ram:"     'put bmap file in RAM:

  'Create text of .bmap file
  fd$="MoveScreen"+CHR$(0)
  fd$=fd$+CHR$(255)+CHR$(94)+CHR$(9)+CHR$(1)
  fd$=fd$+CHR$(2)+CHR$(0)

  'print it to the file
  OPEN "intuition.bmap" FOR OUTPUT AS 1
  PRINT#1,fd$;
  CLOSE 1
  'open the library
  LIBRARY "intuition.library"
  CHDIR "df0:"

RETURN
```

Manipulating Screens

As seen above, it is possible to move a screen up and down under program control. The Intuition library also contains routines that allow you to control the depth arrangement of your screens from a program. These two library functions are

```
ScreenToBack(Screen);
                (a0)
```

and

```
ScreenToFront(Screen);
                (a0)
```

They operate exactly as if the user had clicked on one of the depth arrangement gadget boxes in the top right corner of the screen.

The Workbench screen is a special case. Since it is meant to be available to several different programs at once, you can't count on knowing the address of its Screen structure, as if it were a screen that your application had opened. To arrange the depth of the Workbench screen, you use the Intuition library functions:

```
results = WBenchToFront( );
 (d0)
```

and

```
results = WBenchToBack( );
 (d0)
```

These functions do not require you to pass them the address of a Screen data structure. They return a false (zero) value if the Workbench screen is closed and a true (nonzero) value if it is open and can be moved.

Opening and closing the Workbench screen is a special case also. The Workbench screen opens when you start up the computer and normally stays open all the time. If your program needs more memory, it may be able to gain some by closing the Workbench screen, however. The Intuition library function that performs this task is

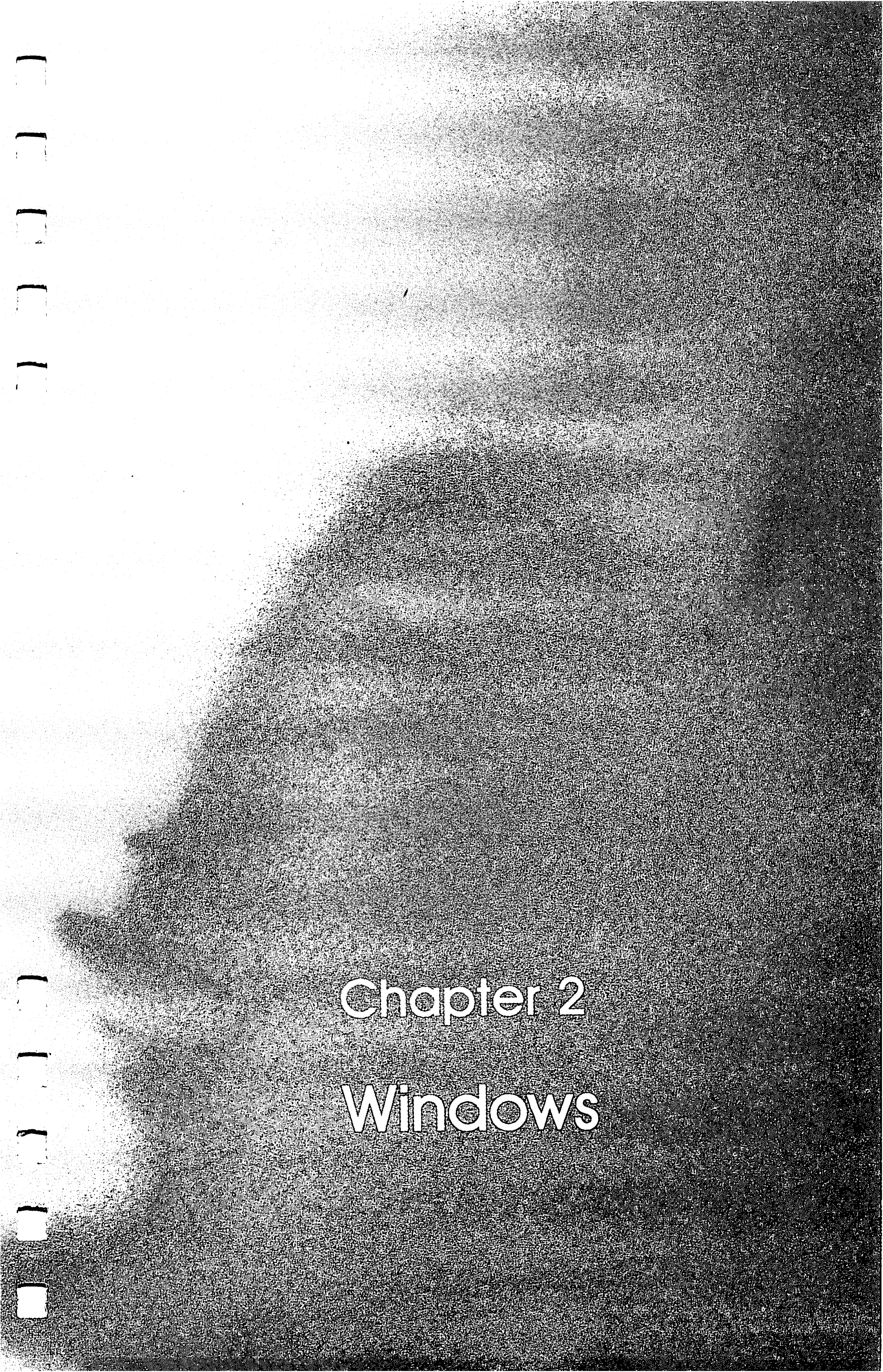
```
results = CloseWorkBench( );
  (d0)
```

This routine will not succeed in closing the Workbench screen if any application programs have opened windows on the screen and are using it. In such a case, the function returns a value of false (zero). But if the only program using the Workbench screen is the Workbench itself, it will close its window and free up the memory used by its display. A true (nonzero) value will then be returned.

If your program has closed the Workbench screen, you should try to open it again when the program finishes. To do this, use the function

```
results = OpenWorkBench( );
  (d0)
```

This function returns a false (zero) value if the Workbench screen cannot be opened, or a true (nonzero) value if the Workbench can be opened or already was open when the function was called.



Chapter 2
Windows

Windows

An Intuition screen completely defines a graphics display area, and it's possible to draw directly onto the screen. If a program writes directly to the screen, however, there is no easy way to separate its output from that of another.

One of the major functions of Intuition is to allow many tasks or programs, each with its own display area, to share a single display. For this reason, Intuition allows you to divide the display area of a screen into several overlapping windows. This system allows each task or program to function as if it has the display area all to itself, even if in reality all are sharing the display bitmap of the same screen.

The Layers Library

The graphics foundation of the Intuition windowing system is a group of operating system routines known as the *Layers library.* The Layers library provides routines for organizing the display into a number of rectangles and making it appear that some rectangles are in front of others, though in fact, all of them share the same display space. This library provides the means of restoring the display when one rectangle is moved or uncovered.

These routines also perform what is known as *clipping.* This means that if you're drawing within a certain rectangle, all drawing will be confined to within the boundaries of that rectangle. When the drawing reaches the borders of the rectangle, graphics output stops so that you don't overwrite the area belonging to an adjacent rectangle or even some memory unrelated to the display.

To the foundation provided by the Layers library, Intuition adds a number of other features to make up its windowing system. Many of these features relate more to the area of input/output (I/O) than they do to graphics. These include the system of gadgets and pull-down menus, graphic devices which allow the user to communicate with a program.

The windowing system also provides the means by which a program can obtain information about the position of the mouse pointer and which keys the user presses on the keyboard. For the most part, we will pass over these features and stress those aspects of Intuition windows that relate to graphics on the Amiga. You should note, however, that the existence of these vital I/O functions provides another reason for using windows for your program graphics, rather than drawing directly on the screen display area.

Opening a Window

The process of opening an Intuition window is very similar to that of opening a screen. The library routine that is used takes this form:

```
Window = OpenWindow(NewWindow);
 (d0)                   (a0)
```

The routine requires as input a pointer to a data structure known as a NewWindow structure. If the call sucessfully opens the window, it returns a pointer to the Window data structure of the new window. This data structure contains all the information provided by the NewWindow structure and more. If unsuccessful, the OpenWindow function returns a zero value.

Like the NewScreen structure used by the OpenScreen routine, the NewWindow data structure describes a wide variety of attributes of the new window. The definition of this data structure in C looks like this:

```
struct NewWindow
  {
  SHORT LeftEdge, TopEdge, Width, Height;
  UBYTE DetailPen, BlockPen;
  USHORT IDCMPFlags;
  ULONG Flags;
  struct Gadget *FirstGadget;
  struct Image *CheckMark;
  UBYTE *Title;
  struct Screen *Screen;
  struct BitMap *Bitmap;
  SHORT MinWidth, MinHeight, MaxWidth, MaxHeight;
  USHORT Type;
  }
```

As you can see, a lot of information is needed to create a new window. That's because there are many variations on the kinds of windows that can be created. We'll discuss these variations below, as we explain the function of each member of the NewWindow data structure.

LeftEdge, TopEdge. These describe the initial position of the top left corner of your window. The required values specify in pixels how far that corner is from the top left corner of the screen. The coordinates for this corner are (0,0). Their vertical component increases as you go down the display, and the horizontal component increases as you move across the display to the right.

Width, Height. These values describe the initial size of the window in pixels. The Width value should be less than or equal to the width of the screen (640 or 320 pixels) minus the LeftEdge value. The Height value should be less than or equal to the height of the screen (200 or 400 lines) minus the TopEdge value.

These size values describe the total size of the window. Keep in mind, however, that not all of this area will be available for your graphics. Unless otherwise specified, each window comes with a border drawn around it. At the minimum, this border is a double line that occupies several pixels. If the border contains one or more gadgets, like the depth arrangers, drag bar, or close box in the top border and sizing box in the right border, the border can be considerably wider. The size of each border can be found in the Window data structure variables BorderLeft, BorderRight, BorderTop, and BorderBottom. Remember that the area available for drawing should be reduced by the values found in these variables.

DetailPen, BlockPen. These values contain the pen numbers used to draw different parts of the window. The DetailPen value is the number of the pen used to draw details like the text in the title bar, certain gadgets, and the inner border line around the window. The BlockPen value is the number of the pen used to draw filled blocks like the title bar and the outer border line that surrounds the window. Either or both of these values can be set to −1, in which case the pen used will be the same one contained in the DetailPen and/or BlockPen variables in the Screen data structure.

IDCMPFlags. This variable can contain a number of flags that specify the conditions under which Intuition will send your program messages about I/O functions.

Flags. The Flags variable contains a lot of information about just what kind of window will be created. This information is in the form of flags, numbers which have a special meaning to Intuition. Some of these flags are mutually exclusive, but most can be added together in a number of different combinations. These flags affect many different aspects of the window's appearance and performance. The explanations of these flags are grouped together below by function.

Refresh method. One of the most important aspects of the window that is controlled by the Flags variable is the method used to refresh its display. When one window is moved on top of another, the display information for the window that is covered up is no longer saved in the display bitmap area. Some provision must be made for saving that information elsewhere so that the display can be restored if the window is later uncovered. The same is true when the sizing gadget is used to make a window a different size. Information is lost when a window is made smaller and must be restored when the window is made larger again.

The Layers library provides Intuition with three different schemes for refreshing the display. These methods vary in the amount of memory used, the amount of work that the program must do to refresh the display, and how quickly the refresh is accomplished. Each is associated with one of the refresh flags that can be stored in the Flags variable. You must set one and only one of these flags when you open a window.

The first method is **SIMPLE_REFRESH**. Simple refreshing requires the least memory of the three methods, but it does the least for you. A SIMPLE_REFRESH window is drawn in the screen's display memory and uses no additional memory buffers. When you choose this refresh method, Intuition preserves the display when the user merely moves the window around the screen with the drag bar. It does not bother to save the portion of a window that is obscured, however, when the window is sized or another window is moved on top of it. Therefore, the program itself must redraw the display whenever it gets a message from Intuition, either via the Intuition Direct Communication Message Port (IDCMP) or the console device, telling it that the window has been uncovered or sized larger.

Although SIMPLE_REFRESH uses less memory than the other methods, it tends to be a bit slower. Intuition does, however, provide a set of functions that can help speed up the refresh routines that you provide. Whenever a window is moved or sized, Intuition keeps track of what part of the old display was damaged by the move. The function BeginRefresh( ) clips the display so that no matter how much drawing your program orders the operating system to do, it will only perform the portions of the drawing commands that act on the area that was damaged. This prevents it from redrawing sections that do not need to be refreshed. When the program is finished with the refresh, it should call the function EndRefresh( ), which terminates the clipping.

The next method is known as **SMART_REFRESH.** Like the SIMPLE_REFRESH window, the SMART_REFRESH window uses the screen's bitmap for its display. When part of a SMART_REFRESH window is obscured, either by another window or by resizing, Intuition saves the part that was covered up in an extra memory buffer. This method requires more memory than SIMPLE_REFRESH since it stores both the portion of the window that is displayed and that which is covered.

In return, however, a SMART_REFRESH window takes care of most of the refresh process by itself and generally refreshes the window more quickly than does SIMPLE_REFRESH. Since Intuition saves the display information for the part of the window that is covered up, it is able to restore that part of the display when the window is uncovered again. The same is true when a window is sized down and then enlarged. The only case in which Intuition will inform the program that a SMART_REFRESH window needs to be refreshed is when the window is made larger than its original size. When that happens, you may use the BeginRefresh( ) and EndRefresh( ) routines to confine the refresh to the area that was just uncovered. Note that if you do not attach a sizing gadget to the SMART_REFRESH window, the program will never have to refresh the display.

The final refresh scheme requires a special window type known as a **SUPER_BITMAP** window. This type of window uses part of the screen's display memory for its graphics, but also has it own complete bitmap storage area. This bitmap area may be larger than that required by the current window display. The SUPER_BITMAP window uses the most memory of all, since information for the entire display area is saved in RAM in addition to the portion of the screen memory used for the display. Because the entire bitmap for this window is always saved in memory, however, Intuition can refresh its display for you automatically.

In addition to the extra memory requirements, the SUPER_BITMAP window requires a little more work to set up. RAM must be allocated for the bitmap area, and the bitmap must be initialized and linked into the window. Because of these extra requirements, the subject of SUPER_BITMAP windows will be left for more thorough treatment in the "Advanced Topics" chapter.

The last flag concerning window refresh is called **NOCAREREFRESH.** This flag should be set only when you do not intend to perform any window refreshing, regardless of the circumstances. It tells Intuition never to send any messages to this window concerning window refresh events.

Borders. Unless you specify otherwise, Intuition automatically draws a double line around each window to make it easier for the user to distinguish where one window ends and another begins. The border area need not be confined to the thickness of these lines, however. If there are gadgets in the border, such as the system close box, drag bar, depth arrangers, or sizing box, the border area will automatically be extended to accommodate these gadgets.

As explained above, these extended borders are drawn within the area specified for the window, and they occupy part of the usable area of the window. There are two flags that can be used to change this state of affairs. The first, **BORDERLESS,** creates a window that has no border lines drawn around it and no extended border area automatically placed around the gadgets. In fact, the only things that denote the edges of such a window at all are the border gadgets, or the text of the window title, if any of these are used. The lack of borders gives you a little more room to draw on, which is useful for applications that need the entire width of the screen—for 80-column text, for example. Of course, since border lines are normally used to separate one window from the next, it could be confusing to put a number of small, borderless windows on the screen at once. That's why it makes more sense to make a borderless window fill the entire display. You can also use the BACKDROP flag, described below, to keep the borderless window in the background.

The other flag, **GIMMEZEROZERO**, creates a window where the border area is drawn in an entirely separate layer from the rest of the window. Normally, the border area of a window is drawn in the same layer as the rest of the window. This means that the border lines and the gadgets take up some of the window's drawing room. You cannot start drawing at the edge of the window because that's where the borders are, and if you do start there, you may draw over gadgets that are located in the border. With a Gimmezerozero window, there is never any possibility that you will draw over the border lines or gadgets. Your drawing area consists only of the part of the window that lies inside the border area. For purposes of drawing, the top left corner of the window—the (0,0) coordinate—lies in an area that is safe to use, rather than in the border area as with other windows. Although a Gimmezerozero window frees you from worrying about drawing over the border area, it does use more RAM than a regular window. It also slows down such operations as moving and sizing windows, since each window is in effect made up of two subwindows.

Use of the **BACKDROP** flag produces a window that always stays in the background. It opens behind every other window that is already open on the screen; it cannot be moved, depth arranged, or sized. In fact, you cannot attach the sizing box, the drag bar, or the depth arrangement gadgets to this kind of window. The close box is the only system gadget that may be attached to a Backdrop window. Non-Backdrop windows always stay in front of a Backdrop window; using the depth arrangement gadgets on a normal window never sends it behind a Backdrop window.

The other distinctive feature of a Backdrop window is that it does not necessarily cover the screen's title bar as other windows do. By default, the screen's title bar goes in front of a Backdrop window that is opened at the top of the screen. You can, however, change this with the Intuition library function ShowTitle, which takes the form

```
ShowTitle(Screen, ShowIt);
              (a0)     (d0)
```

where Screen is a pointer to the address of the Screen data structure, and ShowIt is a Boolean value true (1) or false (0). A call to this function with a true value for the ShowIt variable causes the title bar to be shown in front of Backdrop windows, while a false value hides the title bar behind any window.

It is particularly useful to add Backdrop features to a Borderless window. If you create such a window full-size, with no system gadgets or title, and hide the screen title bar, you will have the entire drawing surface of the display at your disposal, just like that of any ordinary microcomputer display. The difference, however, is that you can still open auxiliary windows on top of such a display.

Gadgets, activation, and other flags. The remainder of the flags are not strictly concerned with display aspects of the window. The first group allows you to attach any of the system gadgets that you wish. These flags are **WINDOWCLOSE** (close box), **WINDOWDEPTH** (depth arrangers), **WINDOW-DRAG** (drag bar), and **WINDOWSIZING** (size box). Two additional flags allow you to position the size box either in the right border (**SIZEBRIGHT**), which is its default position, or in the bottom border (**SIZEBBOTTOM**).

Another set of flags deals with window activation. The active window is the one which is ready to accept input from the user. Generally, it is the user who decides which window will become active, by clicking the mouse button while the pointer rests in that window. You can tell the active window from inactive ones because the title bar of the active window is drawn in solid colors, while the title bars of inactive windows are ghosted (covered with a pattern of dots, making them lighter in appearance).

If you include the flag **ACTIVATE**, upon opening, the new window becomes the active one. Two other flags allow your window to receive a message from Intuition telling it when the user makes it active or inactive.

The flag **ACTIVEWINDOW** tells Intuition that you want your program to receive a message each time the window becomes active, and **INACTIVEWINDOW** requests a message each time the window becomes inactive.

The last two flags are also used for I/O functions. **REPORTMOUSE** requests that a message be sent to your program each time the mouse moves. **RMBTRAP** is used to let your program know about right mouse button clicks, rather than having them perform menu functions without informing your program of their occurrence as is usually the case.

FirstGadget. This NewWindow variable points to the address of the first in a linked list of Gadget data structures which describe your own custom gadgets. If you don't have any custom gadgets, you may set this value to null (zero).

CheckMark. This is a pointer to the address of an Image data structure that describes the shape of the image to be used as a checkmark to show when a menu item has been selected. If you want to use the default checkmark, set this value to null (zero).

Title. This is a pointer to the address of the window title text. This text consists of a string of ASCII characters, ending in an ASCII 0. As much as possible of this text is displayed in the window's title bar, the width of the window being the determining factor. The text is drawn using the screen's default text font. The letters themselves are drawn in the color of the DetailPen, while the background for the letters is drawn in the color of the BlockPen.

If you use a value of null (zero) for the Title field, no title will appear in the title bar. In fact, there will not even be a title bar at the top of the window unless you attach one of the gadgets that go in the top border of the window. These gadgets include the close box, the drag bar, and the depth arrangers.

Screen. If you are using a custom screen, this value should be set to the address of the Screen data structure which was returned by the OpenScreen function. If you are using the Workbench screen, then this field is ignored.

Bitmap. If you chose SUPER_BITMAP for your method of screen refresh, this field should be set to the address of the custom BitMap data structure that you have set up. If you have specified one of the other refresh types, this field will be ignored.

MinWidth, MinHeight, MaxWidth, MaxHeight. These four variables are used to set the minimum and maximum sizes to which the user may change your window by using the sizing gadget. If you do not attach the sizing gadget to this window by setting the flag value WINDOWSIZING, then these variables are ignored by Intuition. If you include the sizing gadget and set any of these values to zero, it means that you wish to use the same setting as that in the Width or Height variables discussed above. Thus, if you set both MinWidth and MinHeight to zero, it means that you don't want to let the window get any smaller than its initial size.

If you wish to change these limits after the window has been opened, you can do so with a call to the WindowLimits function, of the form

```
status =
 (d0)
WindowLimits(Window,MinWidth,MinHeight,MaxWidth,MaxHeight);
                     (a0)      (d0)      (d1)       (d2)      (d3)
```

where Window is the pointer to the Window data structure that was returned by the OpenWindow call, and the other variables are as discussed above. A zero used for any of the size values means that the previous limit should not be changed. A value of true (1) will be returned if all the values are within range, and false (0) if one of the minimums is greater than the current size or if one of the maximums is less than the current size.

Type. This variable is used to specify the type of screen to which this window is attached. The two types that you can choose are WBENCHSCREEN for the Workbench screen and CUSTOMSCREEN for your own custom screen. If you are using a custom screen, you must open the screen before opening the window, and you must place the Screen data structure address returned by the OpenScreen call into the NewWindow variable Screen.

Program 2-1 demonstrates opening a simple window on the Workbench Screen in C. If you are using the *Lattice C Compiler,* call the file "Window.c", and compile and link it using the MakeSimple script. Since most of the demonstration programs in this book require you to open a new window, we'll be using this program as an include file with later demonstration programs that use the Workbench screen. When used for this purpose, the comment marks (/* */) should be removed from the line

```
/* Demo( ); */
```

because Demo is the name of the function that makes up our sample programs.

Program 2-2 is the machine language version of Program 2-1. Call Program 2-2 "Window.asm" and compile it from the CLI with the command

```
:c/assem window.asm -o window.o :include -c W200000
```

This assumes that the assembler is in the *c* directory of your disk, the source code file is in the current directory, and the include files are in the include directory. If the compile is successful, link the window.o file to the library file Amiga.lib with the command

```
:c/alink window.o to window library :lib/amiga.lib
```

again assuming that the linker is in the *c* directory and the library files are in the *lib* directory.

Most of the sample programs in this book will be written in C, and not machine language. Nonetheless, if you compare the C and machine language versions of this program, it should be clear that the two are very similar. If you are a machine language programmer who is at all familiar with C, it should not be too dificult for you to make the necessary translation.

Program 2-1. Opening a Window In C

```
/* Include the definitions we need */

#include <exec/types.h>
#include <intuition/intuition.h>

/* Structures needed for libraries */

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

/* Structures required for graphics */

struct Screen *CustScr;
struct Window *Wdw;
struct ViewPort *WVP;

/***************** Program Constants ****************/

#define Rp   Wdw->RPort      /* shorten this up */

#define BLUP     Ø
#define WHTP     1
#define BLKP     2
#define ORNP     3

/* ******* Pre-initialized NewWindow Structure ******** */

struct NewWindow NewWdw =
   {
   Ø,Ø,          /* Left Edge, Top Edge */
   64Ø,2ØØ,       /* Width, Height */
   BLUP,WHTP,     /* Block Pen, Detail Pen */
```

```
    CLOSEWINDOW,    /* IDCMP Flags */
    SMART_REFRESH | ACTIVATE
     | BORDERLESS | WINDOWCLOSE,    /* Flags */
    NULL,       /* Pointer to First Gadget */
    NULL,       /* Pointer to Check Mark image */
    NULL,       /* Title */
    NULL,        /* Pointer to Screen structure, dummy */
    NULL,       /* Pointer to custom Bit Map */
    Ø,Ø,          /* Minimum Width, Height */
    Ø,Ø,          /* Maximum Width, Height */
    WBENCHSCREEN    /* Type of Screen it resides on */
    };

/* *************** Program Begins Here ********** */

main()
{

/* Open the Intuition and Graphics libraries.
 * Get pointer to WCS routines,
 * and if = Ø, libraries aren't available.
 */

    IntuitionBase = (struct IntuitionBase *)
          OpenLibrary("intuition.library",LIBRARY_VERSION);
    if (IntuitionBase == NULL) exit(FALSE);

    GfxBase = (struct GfxBase *)
          OpenLibrary("graphics.library", LIBRARY_VERSION);
    if (GfxBase == NULL) exit(FALSE);

/* Open the Window.  If Wdw = Ø, it wasn't opened. */

    if (( Wdw = (struct Window *)OpenWindow(&NewWdw)) == NULL)
       exit(FALSE);
```

```
demo();

Wait(1<<Wdw->UserPort->mp_SigBit);
/* wait till close box clicked */

   CloseWindow(Wdw);
   CloseLibrary(GfxBase);
   CloseLibrary(IntuitionBase);

}
```

Program 2-2. Opening a Window In Machine Language

```
   INCLUDE   "exec/types.i"
   INCLUDE   "intuition/intuition.i"

* external references to library routines

   XREF   _AbsExecBase
   XREF   _LVOOpenLibrary
   XREF   _LVOCloseLibrary
   XREF   _LVOWait
   XREF   _LVOOpenWindow
   XREF   _LVOCloseWindow

************** open the Intuition Library *************

   movea.l  #IntuitionName,a1   ;request 'intuition.library'
   move.l   #LIBRARY_VERSION,d0
   movea.l  _AbsExecBase,a6     ;get pointer to Exec library
   jsr      _LVOOpenLibrary(a6) ;jsr thru OpenLibrary offset
   move.l   d0,IntuitionBase    ;save ptr to Intuition lib
   beq      Abort               ;if ptr not found, abort
```

```
************* open our simple window **************

   movea.l  #NewWdw,aØ          ;pointer to NewWindow struct
   movea.l  IntuitionBase,a6    ;pointer to Intuition library
   jsr      _LVOOpenWindow(a6) ;jsr thru OpenWindow offset
   move.l   dØ,Wdw              ;save ptr to Window structure
   beq      Abort               ;if no ptr returned, abort

**************** wait for mouse click ************

   movea.l  Wdw,aØ              ;get pointer to Window
   movea.l  wd_UserPort(aØ),aØ ;get pointer to IDCMP port

* find which of the task's signal bits is set
* when Intuition sends us a message
   move.b   MP_SIGBIT(aØ),d1
   moveq.l  #1,dØ             ;convert bit number to mask
   lsl.l    d1,dØ             ;by shifting so many times

   movea.l  _AbsExecBase,a6  ;set pointer to Exec library
   jsr      _LVOWait(a6)     ;sleep til we get a message

************ close the window and library ***********

   movea.l  Wdw,aØ                 ;set Window pointer
   movea.l  IntuitionBase,a6       ;set ptr to Intuition Lib
   jsr      _LVOCloseWindow(a6)    ;and close the window
   movea.l  IntuitionBase,a1       ;set ptr to Intuition
   movea.l  _AbsExecBase,a6        ;use Exec library
   jsr      _LVOCloseLibrary(a6)   ;and Close the Library

******* quit immediately if library won't open *******

Abort:
   clr.l    dØ                ;return code in dØ
   rts
```

```
************** here's our data **************

   SECTION data,DATA

IntuitionName:
   dc.b   'intuition.library',Ø

WFlags   EQU   WINDOWCLOSE!SMART_REFRESH!ACTIVATE!BORDERLESS

NewWdw:
   dc.w   Ø              ;LeftEdge
   dc.w   Ø              ;TopEdge
   dc.w   64Ø            ;Width
   dc.w   2ØØ            ;Height
   dc.b   Ø              ;DetailPen
   dc.b   1              ;BlockPen
   dc.l   CLOSEWINDOW    ;IDCMPFlags -- Intuition messages
   dc.l   WFlags         ;Flags -- Gadgets, Refresh mode, etc.
   dc.l   Ø              ;FirstGadget -- ptr to user gadgets
   dc.l   Ø              ;CheckMark --ptr to custom ckmark
   dc.l   Ø              ;Title -- ptr to window title text
   dc.l   Ø              ;Screen -- ptr to custom Screen
   dc.l   Ø              ;BitMap -- ptr to custom BitMap
   dc.w   Ø              ;MinWidth -- to size window
   dc.w   Ø              ;MinHeight
   dc.w   Ø              ;MaxWidth
   dc.w   Ø              ;MaxHeight
   dc.w   WBENCHSCREEN ;Type -- use Workbench Screen

   SECTION mem,BSS

IntuitionBase:
   ds.l   1
* place to store Intuition library base address
```

```
Wdw:
   ds.l    1
* place to keep pointer to Window structure

   END
```

Program 2-3. Opening a Custom Screen in C

```
/* Include the definitions we need */

#include <exec/types.h>
#include <intuition/intuition.h>

/* Structures needed for libraries */

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

/* Structures required for graphics */

struct Screen *CustScr;
struct Window *Wdw;
struct ViewPort *WVP;

/***************** Program Constants ****************************/

#define Rp   Wdw->RPort      /* shorten this up */

#define WHITE   ØxFFF
#define RED      ØxFØØ
#define GREEN   ØxØFØ
```

```
#define BLUE   ØxØØF
#define CYAN   ØxØFF
#define PURPLE     ØxFØF
#define YELLOW   ØxFFØ
#define BLACK   ØxØØØ

/* ************ Color Map Data ************* */

   static USHORT colormap [8] =
      {
      BLACK,    /* background color */
#define BGRP      Ø
      WHITE,       /* color of window-close box */
#define WHTP      1
      BLUE,   /* color of menu title */
#define BLUP      2
      RED,   /* color of window-close dot */
#define REDP      3
      GREEN,
#define GRNP      4
      YELLOW,
#define YELP      5
      PURPLE,
#define PURP      6
      CYAN
#define CYNP      7
      };

/* ********** Pre-initialized Text Structure ******* */


struct TextAttr StdFont =
   {
   "topaz.font",              /* Font Name */
   TOPAZ_EIGHTY,                /* Font Height */
   FS_NORMAL,                  /* Style */
   FPF_ROMFONT,                /* Preferences */
   };

/* * Pre-initialized NewScreen and NewWindow Structures * */

   struct NewScreen NewCustScr =
      {
      Ø,Ø,          /* LeftEdge (always=Ø),TopEdge */
      32Ø,2ØØ,3,       /* Width, Height, Depth */
      PURP,YELP,       /* DetailPen and BlockPen */
      SPRITES,        /* special display modes */
      CUSTOMSCREEN,       /* Screen Type */
      &StdFont,       /* Pointer to Custom font*/
      NULL,          /* Pointer to title text */
      NULL,          /* Pointer to Screen Gadgets */
      NULL,          /* Pointer to CustomBitMap */
      };


struct NewWindow NewWdw =
   {
   Ø,Ø,          /* Left Edge, Top Edge */
   32Ø,2ØØ,       /* Width, Height */
   BLUP,WHTP,     /* Block Pen, Detail Pen */
   CLOSEWINDOW,    /* IDCMP Flags */
   SMART_REFRESH | ACTIVATE
    | BORDERLESS | WINDOWCLOSE,    /* Flags */
   NULL,       /* Pointer to First Gadget */
   NULL,       /* Pointer to Check Mark image */
   NULL,       /* Title */
   NULL,        /* Pointer to Screen structure, dummy */
   NULL,       /* Pointer to custom Bit Map */
```

```
    Ø,Ø,          /* Minimum Width, Height */
    Ø,Ø,          /* Maximum Width, Height */
    CUSTOMSCREEN   /* Type of Screen it resides on */
    };

/* **************** Program Begins Here *************** */

main()
{

/* Open the Intuition and Graphics libraries.
 * Get pointer to WCS routines,
 * and if = Ø, libraries aren't available.
 */

    IntuitionBase = (struct IntuitionBase *)
          OpenLibrary("intuition.library", LIBRARY_VERSION);
    if (IntuitionBase == NULL) exit(FALSE);

    GfxBase = (struct GfxBase *)
          OpenLibrary("graphics.library", LIBRARY_VERSION);
    if (GfxBase == NULL) exit(FALSE);

/* Open the Screen and Windows.  If they = Ø, they weren't opened. */

    if ((NewWdw.Screen = CustScr =
 (struct Screen *)OpenScreen(&NewCustScr)) == NULL)
       exit(FALSE);

    if (( Wdw = (struct Window *)OpenWindow(&NewWdw)) == NULL)
       exit(FALSE);

 /* find the viewport and load color map*/
```

```
WVP = (struct ViewPort *)ViewPortAddress(Wdw);
LoadRGB4(WVP,&colormap,8);    /*load our new colors */

demo();

Wait(1<<Wdw->UserPort->mp_SigBit);

   CloseWindow(Wdw);
   CloseScreen(CustScr);
   CloseLibrary(GfxBase);
   CloseLibrary(IntuitionBase);
}
```

Program 2-4. Opening a Custom Screen In Machine Language

```
   INCLUDE   "exec/types.i"
   INCLUDE   "intuition/intuition.i"

* External references to library routines

   XREF    _AbsExecBase
   XREF    _LVOOpenLibrary
   XREF    _LVOCloseLibrary
   XREF    _LVOWait
   XREF    _LVOOpenWindow
   XREF    _LVOCloseWindow
   XREF    _LVOOpenScreen
   XREF    _LVOCloseScreen
```

```
************* open the Intuition Library **********

   movea.l  #IntuitionName,a1    ;request'intuition.library'
   move.l   #LIBRARY_VERSION,dØ
   movea.l  _AbsExecBase,a6      ;get pointer to Exec library
   jsr      _LVOOpenLibrary(a6) ;jsr thru OpenLibrary vector
   move.l   dØ,IntuitionBase     ;save ptr to Intuition lib
   beq      Abort                ;if pointer not found, abort

***************** open our Screen ****************

   movea.l  #NewCustScreen,aØ     ;pointer to NewScreen
   movea.l  IntuitionBase,a6      ;ptr to Intuition library
   jsr      _LVOOpenScreen(a6)    ;jsr thru OpenWindow vec
   move.l   dØ,CustScr            ;save pointer to Screen
   beq      Abort                 ;if no ptr returned, abort

************** open our simple window **************

   movea.l  #NewWdw,aØ           ;pointer to NewWindow
   move.l   CustScr,nw_Screen(aØ) ;link Screen to NewWindow
   movea.l  IntuitionBase,a6     ;pointer to Intuition lib
   jsr      _LVOOpenWindow(a6)   ;jsr thru OpenWindow vector
   move.l   dØ,Wdw               ;save pointer to Window
   beq      Abort                ;if no ptr returned, abort

*************** wait for mouse click ***************

   movea.l  Wdw,aØ             ;get pointer to Window
   movea.l  wd_UserPort(aØ),aØ ;get ptr to IDCMP port

* find which of the task's signal bits is set
* when Intuition sends us a message
   move.b   MP_SIGBIT(aØ),d1
   moveq.l  #1,dØ              ;convert bit number to mask
   lsl.l    d1,dØ              ;by shifting so many times
   movea.l   _AbsExecBase,a6     ;set pointer to Exec library
   jsr       _LVOWait(a6)        ;Wait til we get a message

*********** close the window and library ***********
   movea.l   Wdw,aØ               ;set Window pointer
   movea.l   IntuitionBase,a6     ;set ptr to Intuition Lib
   jsr       _LVOCloseWindow(a6)  ;and close the window
   movea.l   CustScr,aØ           ;set Screen pointer
   jsr       _LVOCloseScreen(a6)  ;and close the Screen
   movea.l   IntuitionBase,a1     ;set ptr to Intuition
   movea.l   _AbsExecBase,a6      ;use Exec library
   jsr       _LVOCloseLibrary(a6) ;Close the Intuition lib

***** quit immediately if something won't open ****

Abort:
   clr.l     dØ                 ;return code in dØ
   rts

**************** here's our data ***************

   SECTION data,DATA

IntuitionName:
   dc.b   'intuition.library',Ø

WFlags  EQU   WINDOWCLOSE!SMART_REFRESH!ACTIVATE!BORDERLESS

************ the NewScreen structure ************

NewCustScreen:
   dc.w   Ø             ;LeftEdge
   dc.w   Ø             ;TopEdge
   dc.w   32Ø           ;Width
   dc.w   200            ;Height
   dc.w   3              ;Depth
   dc.b   Ø              ;DetailPen
   dc.b   1              ;BlockPen
   dc.w   Ø              ;special display modes
   dc.w   CUSTOMSCREEN   ;Screen type--CUSTOMSCREEN
   dc.l   Ø              ;pointer to custom font structure
   dc.l   Ø              ;Title -- ptr to Screen title text
   dc.l   Ø              ;ptr to Screen gadgets
   dc.l   Ø              ;BitMap -- ptr to custom BitMap

************ the NewWindow structure ***********

NewWdw:
   dc.w   Ø             ;LeftEdge
   dc.w   Ø             ;TopEdge
   dc.w   32Ø           ;Width
   dc.w   2ØØ           ;Height
   dc.b   Ø             ;DetailPen
   dc.b   1             ;BlockPen
   dc.l   CLOSEWINDOW   ;IDCMPFlags -- Intuition messages
   dc.l   WFlags        ;Flags -- Gadgets, Refresh mode, etc.
   dc.l   Ø             ;FirstGadget -- ptr to user gadget
   dc.l   Ø             ;CheckMark --ptr to custom ckmark
   dc.l   Ø             ;Title -- ptr to window title text
   dc.l   Ø             ;Screen -- ptr to custom Screen
   dc.l   Ø             ;BitMap -- ptr to custom BitMap
   dc.w   Ø             ;MinWidth --to size window
   dc.w   Ø             ;MinHeight
   dc.w   Ø             ;MaxWidth
   dc.w   Ø             ;MaxHeight
   dc.w   CUSTOMSCREEN ;Type -- use Custom Screen
   SECTION mem,BSS

IntuitionBase:
   ds.l   1
* place to store Intuition library base address

Wdw:
   ds.l   1
* place to keep pointer to Window structure

CustScr:
   ds.l   1
* place to keep pointer to Screen structure

   END
```

Opening a Custom Screen

Some of the sample programs in this book will require the use of a custom screen. Program 2-3 is the C language version of a program demonstrating how to open a custom screen and then open a window on that screen.

This program should be called Window1.c, and should be compiled and linked just as Window.c was. This program will also be INCLUDEd in some of our sample programs later on, and the same caution about removing the comments from the line that calls the Demo( ) function applies.

For the benefit of machine language programmers, Program 2-4 is a similar program which does not open the Graphics library or change the color map.

Manipulating Windows

The Intuition library provides several functions for manipulating windows after they've been opened. These allow you to move the window, resize it, and change its depth arrangement. To move a window, use the the routine MoveWindow. A typical call to this routine takes the form

```
MoveWindow(Window,DeltaX,DeltaY);
           (a0)   (d0)   (d1)
```

where Window is a pointer to the Window data structure returned by the call to OpenWindow. DeltaX and DeltaY are signed values that specify how far the window is to be moved horizontally and vertically. A positive DeltaX value means that the window is to be moved that many pixels to the right, while a negative value means that it is to be moved to the left. A positive DeltaY value means that the window is to be moved that many lines toward the bottom of the display, while a negative DeltaY means that it is to be moved toward the top.

It's your responsibility to make sure that it is possible to move the window to the location specified by MoveWindow. If the window is already at the right edge of the screen, for example, and you try to move it farther to the right, you'll probably crash the system. You can prevent this by checking the window's current position before moving it. The boundaries of the window can be computed from the values found in the TopEdge, LeftEdge, Width, and Height fields of the Window data structure.

You can also change the size of the window under program control, using the SizeWindow procedure. A call to this function looks like

```
SizeWindow(Window,DeltaX,DeltaY);
           (a0)   (d0)   (d1)
```

where Window is a pointer to the Window data structure, and DeltaX and DeltaY specify the movement of the right and bottom borders of the window. A positive DeltaX signifies that the right edge of the window will expand to the right, while a negative value means that it contracts to the left. A positive DeltaY means that the bottom edge of the window expands downward, while a negative DeltaY means that it shrinks upward. As with MoveWindow, this procedure performs no error checking, so it is up to your program to make sure that the window does not, for example, expand off the edge of the screen.

Two other Intuition library functions allow your program to change the depth arrangement of your window. The procedure

```
WindowToFront(Window);
              (a0)
```

brings your window to the front of the display, as if the user had clicked on the UpFront depth arrrangement gadget, while the procedure

```
WindowToBack(Window);
             (a0)
```

sends your window to the back of the display.

Closing a Window

When you are through using a window, you must call the CloseWindow function to erase the window and free up the memory it has been using. The format for this routine is

```
CloseWindow(Window);
            (a0)
```

where Window is the pointer to the Window data structure that was returned by the OpenWindow routine.

BASIC Windows

Unlike C and machine language, Amiga BASIC does not allow you to draw on a screen until you have opened a window on it. The BASIC interpreter uses the Workbench screen and opens two windows on it, the output window, which is used for program output, and the list window, which is used only to display and edit the program listing.

If you have not opened any other windows, graphic output goes by default to the BASIC output window, the one that appears on the left side of the screen with the word BASIC in its title bar when you start up the BASIC interpreter.

But it's quite possible for a BASIC program to open and close its own graphics windows, either on the default Workbench screen or on a custom screen. Graphics output can be directed to any open window, by making it the one that is currently active under program control.

To open a new window, you use the WINDOW statement. This statement requires you to supply some of the information that goes into a NewWindow data structure, but limits your choices. The syntax is

WINDOW window_num [, [title] [,[size] [,attributes] [,screen_num]]]

The window ID. The first value, window_num, is an identification number which other statements use to refer to this window. For example, you use this window number with the WINDOW CLOSE statement to specify which window to close and with the WINDOW OUTPUT statement to specify the window to which graphics output should be directed.

You may use any number from 1 upward for your window_num. The number 1, however, is reserved for the output window that BASIC uses. While you can close this window and reopen it like other windows, it still has a special significance, since it is the only window in which the user can type immediate-mode BASIC commands. Moreover, a program does not have absolute control over this window, since its comings and goings are affected by the Show Output item on the BASIC menu bar. If there is no output window currently open, BASIC tends to be fussy about the syntax used to open one. Since default values (such as that for the size of the window) do not always apply to the default output window, you may have to specify values that are listed here as optional.

Be careful when using the default output window for your program's output. Remember that the default window has a size gadget that lets the user change the size of the window at any time, and the user is likely to change the size of that window to allow it to share space with the list window. Therefore, if it is important that your window be a certain minimum size (as it almost always is), either open a second window or reopen window 1 to the requisite size.

The title is an optional string expression that will be displayed in the window's title bar. If you omit this expression, there will be no title (and there may not be a title bar, either, depending on the value chosen for attributes). Window 1, the default output window, is an exception. It displays the name of the program or the word BASIC in its title bar if you fail to specify your own title.

Sizing and positioning BASIC windows. Another optional value that you may specify is the size and position of the window. If you omit this value, your new window will cover the entire display screen. The exception to this is when you open an existing window or one that your program had opened earlier and closed. In that case, the window defaults to its previous size.

The way that you specify the size and position of the window is to describe the coordinates of the top left and bottom right corners of the window. The format for this description is

(left, top)-(right, bottom)

As we said earlier, the display is 640 pixels (dots) wide in high-resolution mode and 320 pixels wide in low-resolution mode. When describing the left and right coordinates for the screen, we say that horizontal position 0 is at the left edge of the screen, and 319 or 639 is at the right edge, depending on whether low- or high-resolution mode is used. The height of the display is 200 lines noninterlaced or 400 lines interlaced. In describing the top and bottom coordinates, we say that line 0 is at the top of the display, and line 199 or line 399 at the bottom, depending on whether or not the display is interlaced.

This would lead you to believe that the correct description for a full-size window would be (0,0)–(639,199) for a high-resolution, noninterlaced display. But as mentioned earlier in this chapter, you must also take into account the space required for the border line that is drawn around the window and for gadgets like the title bar and the sizing gadget.

Since the WINDOW statement does not allow you to create a borderless window, at the very least each window will have a a double border line drawn around it. Because of this, the highest line number that you can specify as the bottom line of the window is 186 for a noninterlaced screen and 386 for an interlaced screen. The highest value that you can specify for the right side of the window is 631 for a high-resolution screen and 311 for a low-resolution screen. If you attach a sizing gadget to your window as described below, this gadget is drawn in the right border of the window and further reduces the possible width of the window. A window that contains a sizing gadget can have a maximum horizontal value of 617 on a high-resolution screen or 297 on a low-resolution screen. To summarize, the table gives the proper descriptions for the largest possible windows:

(0,0)–(631,186)	high resolution	noninterlaced	no sizing gadget
(0,0)–(617,186)	high resolution	noninterlaced	with sizing gadget
(0,0)–(631,386)	high resolution	interlaced	no sizing gadget
(0,0)–(617,386)	high resolution	interlaced	with sizing gadget
(0,0)–(311,186)	low resolution	noninterlaced	no sizing gadget
(0,0)–(297,186)	low resolution	noninterlaced	with sizing gadget
(0,0)–(311,386)	low resolution	interlaced	no sizing gadget
(0,0)–(297,386)	low resolution	interlaced	with sizing gadget

The attributes value is a subset of the Flags field used in the NewWindow data structure which we described above. It is used to specify which of the standard window gadgets will be attached to the window that you are opening and the screen refresh method to be used.

The available system gadgets include the sizing gadget, the drag bar, the depth arrangement boxes, and the close box.

The sizing gadget appears in the lower right-hand corner of the window and allows the user to change the size of the window.

The drag bar appears in the title bar at the top of the window and allows the user to move the window around on the screen if the window is smaller than the screen.

The depth arrangement boxes appear in the upper right corner of the window; they can be used to send the window to the back of the screen (the dark box) or to bring it to the front of the screen (the light-colored box).

The close box is located in the upper left corner of the window and allows the user to close the window entirely by clicking on the box.

Refresh window. The attribute value also lets you choose from two of the available screen refresh methods. If you so desire, it lets you make the new window a SMART_REFRESH window. As explained above, this means that the contents of the window will be redrawn after the window has been covered by another window or has been resized. While this can be very convenient, it may be costly in terms of memory usage, since BASIC must reserve enough memory to save the part of the image that is covered up. If you do not specify that you want a SMART_REFRESH window, you will get a SIMPLE_REFRESH window instead.

The following table lists the numbers that can be used for the attribute value and their meanings:

Attribute Value	Attribute
1	Sizing gadget
2	Drag bar gadget
4	Depth arrangement gadget
8	Close gadget
16	Smart refresh

Any or all of these values may be used. To attach more than one gadget to your window, add the attribute value of each together. For example, use an attribute value of 3 to indicate that you wish the window to have both a sizing gadget (1) and a drag bar (2). Any number from 0 to 31 is a valid attribute value. If you use 0 as the attribute value, you will get a plain window with a border around it (and a title bar, if you have given the window a title). If you do not specify a value here, the default value 31 (which provides all of the gadgets and smart screen refresh) is used.

Screen number. The last option value for the WINDOW statement is screen_num, the number of the screen upon which you wish the window to be drawn. If you do not specify a value here, the default Workbench screen (whose screen_num is −1) will be used. If you want to attach the window to a custom screen you have opened, use the screen number that you specified as the first value of the SCREEN command when you opened that screen.

When you open a window with the WINDOW statement, not only is a new window created, but two other things happen as well. The first is that this window is brought to the front of the screen and becomes the active window (the window whose title bar is shown in solid lines). The second is that this window becomes the current output window. This means that from then on, the output from any graphics or text commands will be directed to this window until the program redirects output to another window. It is possible to make an existing window the current output window without bringing it to the front of the screen by using the WINDOW OUTPUT statement. The syntax of this statement is WINDOW OUTPUT window_num, where window_num is the window number assigned as the first value of the WINDOW statement that created the window.

It's also possible to bring an existing window to the front of the display and make it the current window with the WINDOW statement. The syntax for this form is WINDOW window_num, with no other values specified. If the attribute value for that window is 15 or higher, the contents of the window will be restored when the window is brought forward.

BASIC's WINDOW Function

In addition to the WINDOW statement, BASIC also provides a WINDOW function that can be used to check which window is active, which is the current output window, the size and depth of the current output window, and the location where the next text character will be drawn. In effect, it provides much of the information that a C program could learn by checking the Window data structure. The syntax for this function is

value = Window*(n)*

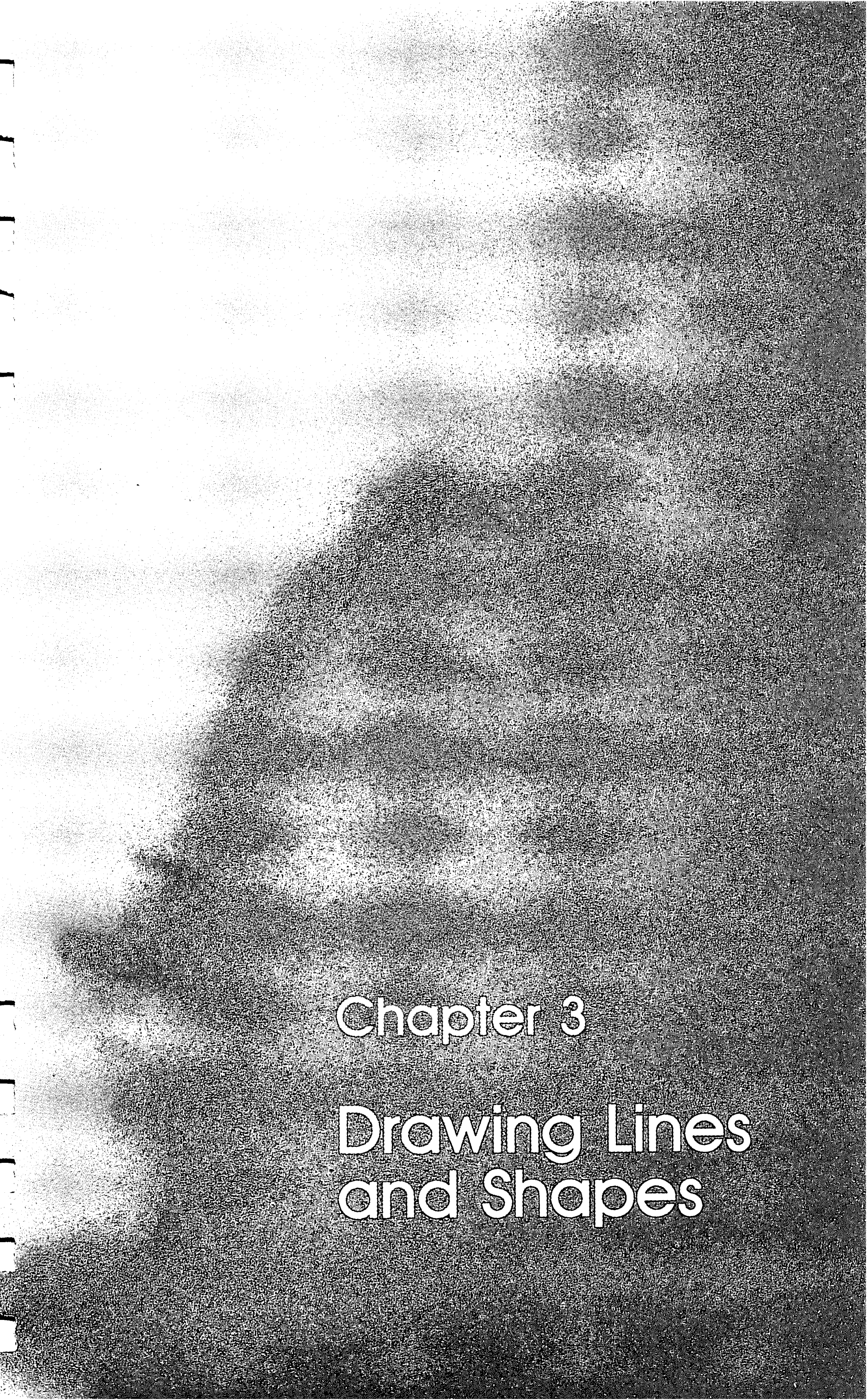
where *n* is a number from 0 to 8. The following table shows what information is returned for each value of *n*.

Value of *n*	Information Returned
0	Window_num of the active window
1	Window_num of the current output window
2	Width of the current output window
3	Height of the current output window
4	The *x* coordinate where the next text character will be drawn in the current output window
5	The *y* coordinate where the next text character will be drawn in the current output window
6	The maximum pen number for the current output window
7	The address of the Intuition Window data structure for the current output window
8	The address of the RastPort data structure for the current output window

These last two bits of information are particularly useful in calling Intuition and Graphics library functions from BASIC, since these usually require a pointer to either the Window or RastPort data structures. In addition, many other useful pieces of information can be gained by PEEKing at these structures directly.

Closing a Window from BASIC

To close a window, use the WINDOW CLOSE statement. The syntax for this statement is WINDOW CLOSE window_num. When you use this statement to close the current output window, the visible window that was previously the current output window becomes current once more. Note that this is different from what happens when the user of a program closes the window by clicking on the close gadget. In that case, the closed window remains the current window, and graphics output goes nowhere at all. A program can check to see whether the current output window has been closed by using the WINDOW function. If WINDOW(7) = 0, there is no Window data structure, which means that the current window has been closed and the program should make another window current.



Chapter 3

Drawing Lines and Shapes

Drawing Lines and Shapes

Whenever Intuition opens a window, it prepares a drawing surface as well. It keeps the information it needs for graphics rendering in a data structure known as a *rastport*. The rastport for each window includes the following types of information:

- Information used for clipping so that drawing takes place only within the area of the window that is currently exposed.
- The pens used in drawing foreground and background areas. These pens are known as the APen (foreground) and BPen (background). A third pen, OPen (the area outline pen) is sometimes used to draw an outline around a filled shape.

  To specify the color with which you wish to draw, you set the APen (and sometimes the BPen) to the number of the pen that contains that color. For example, the default colors for the Workbench screen are blue (pen 0), white (pen 1), black (pen 2), and orange (pen 3). Therefore, to draw an orange line, you would first set the foreground pen value to 3, the number of the orange pen. The actual mechanics of setting the colors that will be drawn by each pen are discussed in a separate section below.
- Pattern information, for drawing patterned lines and patterned area fills.
- Information about temporary storage areas and other information required for filling entire areas of the display.
- The current drawing position of the drawing pen.
- Information about animated shapes.
- The current drawing mode.
- Information about the text font and font styles that are being used.

Most of the drawing routines in the Graphics library refer to the window's rastport. C language programmers can use the RPort field in the Window data structure returned by OpenWindow to locate the window's rastport. If the declaration for the Window data structure initialized by OpenWindow is

struct Window *Wdw

then the window's rastport may be referred to as

Wdw->RPort

BASIC programmers can also use the rastport to call Graphics library functions. The BASIC function WINDOW(8) returns a pointer to the rastport of the current output window.

Color Selection

As stated in Chapter 1, the maximum number of colors that can be displayed at one time on a given screen depends on the number of bit planes of display memory allocated for that screen. One bit plane allows two colors, two bit planes four colors, three bit planes eight colors, and so forth.

We have also seen that if there are three bit planes, the display memory for each dot position on the display screen contains three binary digits, which may hold a number from 0 to 7. This number held in display memory does not refer directly to a color, using a code where 0 is black, 1 is white, and so forth. Instead, the number at the screen dot position refers to a color register.

The color registers may be thought of as a set of 32 pens, each of which may filled with colored ink in any of the 4096 shades that can be displayed on the Amiga. Register 0 always holds what is normally thought of as the background color; any dot position whose display memory holds the number 0 will display this color. When you wish to use another color to draw a line or a point, you specify the pen (color register) that is to be used in drawing it. Whatever color ink it currently contains is the color that the pen will draw.

Unlike ink, however, the color of a dot drawn onscreen can change after you have drawn it. When the display memory for a screen dot holds the number of a particular pen, that dot displays whatever color is in the pen at any given moment, not the color that was in the pen at the time the dot was drawn. This means that if you use pen 1 to draw a line, and that pen contains the color red, the line will be red. But if you change the color in pen 1 to green after you've drawn the line, the line you drew and everything else on the screen that was drawn with pen 1 will instantly become green.

The two factors which determine what color will be drawn on the screen, therefore, are the pen you're using for the drawing and the color used by that pen. You choose a pen to draw with by assigning a pen number to the Amiga's drawing pens. There are two primary drawing pens, the foreground pen (APen) and the background pen (BPen). The foreground pen is used when drawing single points or solid lines. When drawing dotted lines or text, the background pen is used in addition to the foreground pen.

The Graphics library routines used to set the drawing pens are SetAPen and SetBPen. The formats for calls to these routines are

```
SetAPen (RastPort, Pen);
           (a1)    (d0)
```

and

```
SetBPen (RastPort, Pen);
           (a1)    (d0)
```

where Pen is the pen number (color register) to be used for the drawing pen.

In BASIC, the COLOR statement is used to set the color register for each of the drawing pens. The syntax is

COLOR [foreground_pen_number] [, background_pen_number]

where foreground_pen_number and background_pen_number are the numbers of the pens (color registers) used by the foreground and background pens, respectively. If your program does not use any COLOR statements, the foreground pen defaults to pen 1 and the background pen to pen 0.

Color Registers

In addition to determining which color register will be used for drawing, we must also determine the color that the register contains. Colors are chosen by mixing various levels of the colors red, blue, and green. Each color register holds one of 16 color levels for each of these colors, which means that there are 4096 (16 × 16 × 16) possible colors to choose from.

The Graphics library routine that you use to set a color register for a particular screen is SetRBG4 (set four bits each of red, green, and blue values).

```
SetRGB4 (ViewPort, pen, red, green, blue);
          (a0)     (d0) (d1)  (d2)   (d3)
```

Note that this function requires as part of its input the address of the ViewPort data structure. As was explained in Chapter 1, the ViewPort describes a horizontal slice of the display that has particular display characteristics, which result from the way it sets the graphics hardware registers. It is related to the screen. The fact that this function uses the ViewPort should remind you that the same pen colors are used by all of the windows in a screen.

To find the address of the ViewPort associated with a particular window, use the Intuition library function:

```
ViewPort = ViewPortAdress(Window);
  (d0)                     (a0)
```

The other values required by this function are the number of the pen (color register) that you are changing and the red, green, and blue color values. These values represent color intensity from 0 (darkest) to 15 (brightest).

It is also possible to load several color registers at once with the routine LoadRGB4. This routine takes the form

```
LoadRGB4 (ViewPort, Colormap, Pens);
           (a0)      (a1)     (d0)
```

where Colormap is a pointer to a table of 16-bit color values, stored in the format 0x0RGB, where the high four bits (nybble) are always zero, and the next three nybbles each hold a 4-bit value for red, green, and blue intensity. The Pens value specifies the number of registers to load from the table. Colors are always assigned in order, starting with pen 0, then pen 1, and so on.

In Amiga BASIC, you set the color for each pen with the PALETTE statement. The syntax for this statement is

```
PALETTE pen_number, red_value, green_value, blue_value
```

The value pen_number specifies the color register whose color you wish to change. The values red_value, green_value, and blue_value are the levels of each of these three primary colors you wish to use. These are expressed as fractions ranging from 0 (the lowest level, using none of that color) to 1 (the highest level of that color). Although in theory, the 16 levels could be represented by dividing by 15, in practice, Amiga BASIC does not convert the fractions to color levels quite evenly. Table 3-1 gives the values that we will be using with the PALETTE statement and the range of values that can be used to produce the same color level.

Table 3-1. PALETTE Values

Color Level	Palette Value	Range of Acceptable Values
0	0	0.00 to 0.03
1	0.05	0.04 to 0.09
2	0.1	0.10 to 0.15
3	0.2	0.16 to 0.21
4	0.25	0.22 to 0.28
5	0.3	0.29 to 0.34
6	0.4	0.35 to 0.40
7	0.45	0.41 to 0.46
8	0.5	0.47 to 0.53
9	0.55	0.54 to 0.59
10	0.6	0.60 to 0.65
11	0.7	0.66 to 0.71
12	0.75	0.72 to 0.78
13	0.8	0.79 to 0.84
14	0.9	0.85 to 0.90
15	1	0.91 to 1.00

Since there are 4096 possible combinations, it is impossible to describe each available combination or explain exactly how to find a particular shade. In general, however, the higher the color level, the brighter the color, and the lower the level, the darker the color. Whether the color displayed by a register tends toward the red, green, or blue depends on which value has the highest brightness level. If all three values are equal, the color is a shade of gray.

Thus, PALETTE 0,0,0,0 or SetRGB4 (Vp,0,0,0,0) sets pen 0 to black, while PALETTE 0,1,1,1 or SetRGB4 (Vp,0,15,15,15) sets it to white. You may lighten a shade by increasing the value of the two other colors in equal proportions. PALETTE 0,1,0,0 or SetRGB4 (Vp,0,15,0,0) sets pen 0 to a bright red, while PALETTE 0,1,.3,.3 or SetRGB4 (Vp,0,15,5,5) lightens it to a rose color. To darken the original red color, you could try PALETTE 0,.5,0,0 or SetRGB4 (Vp,0,8,0,0).

When you are unsure of what colors to mix, it may help to start with the nearest primary color mixture and experiment from there. These are the primary color mixtures:

Black	(PALETTE) 0,0,0	(SetRGB4) 0,0,0
Blue	(PALETTE) 0,0,1	(SetRGB4) 0,0,15
Green	(PALETTE) 0,1,0	(SetRGB4) 0,15,0
Cyan	(PALETTE) 0,1,1	(SetRGB4) 0,15,15
Red	(PALETTE) 1,0,0	(SetRGB4) 15,0,0
Purple	(PALETTE) 1,0,1	(SetRGB4) 15,0,15
Yellow	(PALETTE) 1,1,0	(SetRGB4) 15,15,0
White	(PALETTE) 1,1,1	(SetRGB4) 15,15,15

If you do not specify a color change for a particular color register, the default color will be used. The default values for each of the 32 pens are listed in Table 3-2. The value given is that used for the SetRGB4 function, with the BASIC PALETTE value in parentheses.

Keep in mind that the same color palettes are used by every window in a screen. When you change the pen colors with the PALETTE statement, you affect the color of every window that appears in the same screen as the current output window. The change is limited to that screen, however, and windows in other screens will not be affected.

Locating Color Information

Sometimes it is useful for your program to be able to find out the actual colors that your window is using. This information can be learned indirectly from the ViewPort data structure. As we have seen above, you find the ViewPort address by using the Intuition function ViewPortAddress. One of the members of the ViewPort struct is a pointer to another data structure called the ColorMap. In turn, the ColorMap structure contains

Table 3-2. Default Pen Colors

Pen	Red	Green	Blue	Color
0	0 (0)	5 (0.3)	10 (0.6)	Dark blue
1	15 (1)	15 (1)	15 (1)	White
2	0 (0)	0 (0)	2 (0.1)	Black
3	15 (1)	8 (0.5)	0 (0)	Orange
4	0 (0)	0 (0)	15 (1)	Blue
5	15 (1)	0 (0)	15 (1)	Purple
6	0 (0)	15 (1)	15 (1)	Cyan
7	15 (1)	15 (1)	15 (1)	White
8	6 (0.4)	2 (0.1)	0 (0)	Dark brown
9	14 (0.9)	5 (0.3)	0 (0)	Red-orange
10	9 (0.55)	15 (1)	1 (0.05)	Lime green
11	14 (0.9)	11 (0.7)	0 (0)	Gold
12	5 (0.3)	5 (0.3)	15 (1)	Blue
13	9 (0.55)	2 (0.1)	15 (1)	Violet
14	0 (0)	15 (1)	8 (0.5)	Blue-green
15	12 (0.75)	12 (0.75)	12 (0.75)	Gray 12
16	0 (0)	0 (0)	0 (0)	Black
17	13 (0.8)	2 (0.1)	2 (0.1)	Red
18	0 (0)	0 (0)	0 (0)	Black
19	15 (1)	12 (0.75)	10 (0.6)	Tan
20	4 (0.25)	4 (0.25)	4 (0.25)	Gray 4 (dark)
21	5 (0.3)	5 (0.3)	5 (0.3)	Gray 5
22	6 (0.4)	6 (0.4)	6 (0.4)	Gray 6
23	7 (0.45)	7 (0.45)	7 (0.45)	Gray 7
24	8 (0.5)	8 (0.5)	8 (0.5)	Gray 8
25	9 (0.55)	9 (0.55)	9 (0.55)	Gray 9 (medium))
26	10 (0.6)	10 (0.6)	10 (0.6)	Gray 10
27	11 (0.7)	11 (0.7)	11 (0.7)	Gray 11
28	12 (0.75)	12 (0.75)	12 (0.75)	Gray 12
29	13 (0.8)	13 (0.8)	13 (0.8)	Gray 13
30	14 (0.9)	14 (0.9)	14 (0.9)	Gray 14 (light)
31	15 (1)	15 (1)	15 (1)	White

a pointer to a table of colors called ColorTable. So, to get the address of this table, you use these statements:

```
ColorMap = ViewPort->ColorMap;
ColorTable = ColorMap->ColorTable;
```

Once you find the address of the table, you must know how to interpret the numbers it contains. The color values for each pen (color register) are stored in a 16-bit word. The first word gives the colors for pen 0, the second for pen 1, and so on. In these 16-bit words, the first four bits are zeros, the next four represent the red value, the next four the green, and the final four bits represent the blue color level. So, the hex value 0x0f82 represents a red value of 15, a green value of 8, and a blue value of 2.

In order to find the colors for a window from BASIC, first find the address of the window's ViewPort. To do this, use the library routine ViewPortAddress. Until now, we have only used passed values to a library routine. To get a value back, we must use the DECLARE FUNCTION command. The syntax for this command is

```
DECLARE FUNCTION FunctionName( ) LIBRARY
```

In this case, we would use the statement

```
DECLARE FUNCTION ViewPortAddress&( ) LIBRARY
```

Of course, we must also open the Intuition library with the statement

```
LIBRARY "intuition.library"
```

We must also have an intuition.bmap file in our current directory to let BASIC know the proper offset for the ViewPortAddress function. As with previous example programs, we will include a subroutine in the program below to create such a file in case the user does not have one. Once this is done we can find the ViewPort address with the statement

```
ViewPort& = ViewPortAddress&(WINDOW(7))
```

From the C language definition of the ViewPort structure, we can determine that the address of the ColorMap structure appears at an offset of four bytes from the beginning of the ViewPort. Thus,

```
ColorMap& = PEEKL(ViewPort&+4)
```

From the definition of the ColorMap structure, we may also tell that the address of the ColorTable is four bytes from the beginning of that structure, so

```
ColorTable& = PEEKL(ColorMap&+4)
```

If we put these together, we get the statement

ColorTable& = PEEKL(PEEKL(ViewPort&+4)+4)

Once we have found the address of the ColorTable, we may use the PEEK function to look at the color settings for the individual color registers. Program 3-1 shows how to find the red, green, and blue values for each color register. It uses an array to translate these values from their normal range 0–15 to the fractional values used by the PALETTE statement. It prints a table that shows the PALETTE red, green, and blue values for each pen.

Program 3-1. Finding Color Values from BASIC

```
DECLARE FUNCTION ViewPortAddress&() LIBRARY
GOSUB Init

VPA& = ViewPortAddress&(WINDOW(7))
ColorTable& = PEEKL(PEEKL(VPA&+4)+4)

FOR Pen = Ø TO 31
  Red = PEEK(ColorTable&+2*Pen)
  Bluegreen = PEEK(ColorTable&+2*Pen+1)
  Green=Bluegreen\16
  blue = Bluegreen MOD 16
  PRINT "Pen";Pen;
  PRINT Colvals(Red),
  PRINT Colvals(Green),
  PRINT Colvals(blue)
NEXT

LIBRARY CLOSE
END

Init:

DIM Colvals(15)
FOR X=Ø TO 15
READ a: Colvals(X)=a
NEXT X

DATA Ø,.Ø5,.1,.2,.25,.3,.4,.45,.5
DATA .55,.6,.7,.75,.8,.9,1

Initlib:
  CHDIR "ram:"     'put bmap file in RAM:

  'Create text of .bmap file
  fd$="ViewPortAddress"+CHR$(Ø)
  fd$=fd$+CHR$(254)+CHR$(212)+CHR$(9)+CHR$(Ø)
  'print it to the file
  OPEN "intuition.bmap" FOR OUTPUT AS 1
  PRINT#1,fd$;
  CLOSE 1

  'open the library
  LIBRARY "intuition.library"
  CHDIR "df0:"

RETURN
```

Drawing Points

The simplest of the drawing commands is that used to set the color of a single point of the display. The Graphics library routine used to accomplish this is WritePixel. The format for this function is

```
result = WritePixel(RastPort,  X,  Y);
 (d0)                    (a1)  (d0) (d1)
```

where RastPort is a pointer to the RastPort data structure of the window, and *x* and *y* specify the horizontal and vertical coordinates of the point to be drawn. These *x* and *y* coordinates are relative to the top left corner of the window. Their values, therefore, should be smaller than those of the window's boundaries. You may check the size of the window by looking at the Width and Height values in the Window data structure.

If the function is successful, it draws the point specified in the color of the current foreground pen (APen), and returns a value of 0. If the routine could not draw the point because it lay outside the area of the rastport, the function returns a value of −1.

In Amiga BASIC the statements that color a single dot on the screen are PSET and PRESET. The two statements are identical, except PSET uses the foreground pen as its default drawing pen and PRESET uses the background pen. The syntax is

PSET [STEP] (x,y) [,pen]
PRESET [STEP] (x,y) [,pen]

There are two ways of indicating the position at which you want the dot drawn. The first is to use absolute horizontal and vertical coordinates. The horizontal coordinates range from 0 at the left edge of the screen to a maximum of 631 or 311 at the right edge of the screen for a full-size window, depending on whether your screen is high-resolution or low-resolution. If you have a sizing gadget in the right border of the window, the maximum is cut to 617 or 297. The vertical coordinates range from 0 at the top of the screen to 186 or 386 at the bottom, depending on whether the screen is noninterlaced or interlaced. If there is no title bar, the coordinate at the bottom of the screen is 195 (noninterlaced) or 395 (interlaced). If your window is smaller, of course, you should use values that are less than the width and height of the current output window. You can find these values by using the WINDOW(2) function to return the window's width and the WINDOW(3) function to return its height. The *x* and *y* coordinates that you specify are relative to the top left corner of the window, regardless of where the window is positioned on the screen.

To put a white dot (the default color of the default foreground pen) midway down the left edge of the standard output window on the Workbench Screen, you would use

```
PSET (0,98)
```

To erase that dot (by drawing over it with the background pen), you could use

```
PRESET (0,98)
```

Relative Coordinates

The other way to specify the point at which to draw the dot is to indicate that you wish to use relative coordinates by including the keyword STEP. Relative coordinates specify a position relative to the last dot drawn. If none has been drawn yet, the position is relative to the middle of the output window (including the borders). For a full-size low-resolution, noninterlaced window, for example, this position would be (160,100). A positive horizontal coordinate indicates that the dot will be positioned to the right of the last one, while a negative coordinate moves the dot to the left. A positive vertical coordinate means that this dot is drawn lower than the last one, and a negative vertical coordinate means it is drawn closer to the top. For instance, if the last dot drawn was at (100,50), this statement would draw the next dot at (90,70):

```
PSET STEP (−10,20)
```

If the last dot drawn was at position (150,90), this statement would draw a dot at (110,80):

```
PSET STEP (−40,−10)
```

Relative coordinates are extremely useful when you wish to draw the same image in different places, or when you aren't quite sure where the image will be drawn.

Let's say, for example, that you are drawing an image in a window that has a sizing gadget. If the user leaves the window alone, the right edge may be at position 600. But if he or she shrinks the window, its right edge may be only at position 400. You can find the right edge with the WINDOW function and set the first point accordingly. By using relative coordinates for the rest of the drawing statements, all of them will then be positioned properly, regardless of where the right edge of the window is. The other advantage of using relative coordinates is that they can make it easier to change your program. If you later decide that you want to move an image over a few pixels, it is much easier just to change the starting point than to change the coordinates for every point.

Pen Color

Both the PSET and PRESET statements take an optional pen value. That value, if specified, selects the pen to be used in drawing the dot. If none is specified, the PSET statement uses the color register associated with the foreground pen, and PRESET uses the color register associated with the background pen. The foreground and background pen values default to color registers 1 and 0, respectively. You can change these assignments at any time, however, by using the COLOR statement (see below). Note that when you specify the pen to use, PSET and PRESET can be used interchangeably; the only difference between them is the default pen that each uses.

Which Pen?

Sometimes it is useful for a program to be able to tell what pen was used to color a particular location in a window. The operating system provides a routine called ReadPixel which does just that. A call to this function is of the form

```
Pen = ReadPixel (RastPort,  X,  Y);
(d0)                 (a1)   (d0) (d1)
```

The value RastPort is the address of the window's RastPort structure. The *x* and *y* values stand for the horizontal and vertical coordinates of the point that you wish to read. If the point lies within the area of the rastport, the value returned will be the number of the pen with which the dot is colored. If the point is out of range of the rastport, a −1 is returned.

In Amiga BASIC, the POINT function returns the same information. The syntax of this function is

```
Pen = POINT(x,y)
```

where *x* and *y* specify the horizontal and vertical coordinates of the point to be read. Like ReadPixel, this function returns the pen number used to color the point if it lies within the area of the window. If the point lies outside the current output window boundaries, the function returns a value of −1.

```
PSET (100,50),3          'Draw at 100,50 with pen 3
Pen& = POINT(100,50)     'Read dot at 100,50 into Pen&
PRINT Pen&               'Should be 3, for pen 3
```

This program draws a dot at (100,50) with pen 3, then reads the pen value at (100,50) into the variable Pen&. The value of Pen& is printed to confirm that it has read the pen number correctly.

Drawing Lines and Shapes

Drawing single points is the least of the Amiga's abilities. Amiga BASIC and the operating system also contain commands that allow you to draw lines and entire geometric shapes such as rectangles, squares, circles, ellipses, and polygons at once.

The Graphics library routine that is used to draw lines is called Draw. A call to this routine looks like

```
Draw (RastPort,  X,  Y);
          (a1)  (d0) (d1)
```

where *x* and *y* are the coordinates for the endpoint of the line. The starting point for Draw depends on the current position of the drawing pen. This position is also sometimes referred to as the pixel cursor. Whenever you use one of the drawing pens to do any drawing, the position of the pen stays at the last dot that was drawn. For example, if you use WritePixel to color the dot at position (200,100), the drawing pen is left at that spot after the dot is drawn.

It is possible to move the drawing pen without drawing anything. The Graphics library routine Move is used to pick up the drawing pen and move it to a new location. A call to this routine takes the form

```
Move (RastPort,  X,  Y);
          (a1)  (d0) (d1)
```

Therefore, to specify both the starting and ending points of the line, you must use a call to Move followed by one to Draw. For example, to draw a line from position (10,15) to position (100,150), you would use the sequence

```
Move (RastPort, 10, 15);
Draw (RastPort, 100, 150);
```

Drawing Lines and Rectangles from BASIC

In Amiga BASIC, the LINE statement is the one you use to draw lines or rectangles:

```
LINE [[STEP] (x1,y1)]-[STEP] (x2,y2), [pen_number] [,b [f]]
```

With the LINE statement, you specify two pairs of coordinates, one for the starting point and one for the ending point. These coordinates can be absolute coordinates, relative coordinates, or a combination of absolute and relative. For example, the statements

```
LINE (30,50) - STEP (40,40)
LINE STEP (0,0) - STEP (−40,40)
```

first draw a line from (30,50) to (70,90) and then draw a line from that point to (30,130).

The value pen_number can be used to indicate the pen to use for drawing the line. If no pen is specified, the current foreground pen is used as the primary drawing pen.

Besides drawing lines, the LINE statement can also be used to draw rectangles. By adding the letter *b* after the pen number (or a comma used as a place holder instead of the pen number), you can indicate that you want a box to be drawn. When this option is used, the first pair of coordinates specifies the top left corner of the box, while the second pair determines where the lower right corner will be placed. For example, this statement draws a box from (100,50) to (150,50) to (150,100) to (100,100) to (100,50), using the foreground pen color:

```
LINE (100,50) - STEP (50,50),,b
```

If you use the letters *bd* instead of just *b*, the box is filled in with either the foreground pen color or the color of the pen that you've selected. For more information, see the section about filled shapes, below.

Lines and Points, Program Examples

To consolidate what we've presented so far, here are a couple of short example programs. Programs 3-2 and 3-4 are written in C, and Programs 3-3 and 3-5 are in BASIC.

Program 3-2 uses the WritePixel, Move, and Draw routines to draw three lines in different colors. It uses a window that sits on the 640 × 200 Workbench screen. Program 3-3 is a BASIC version and uses PSET and LINE.

Program 3-4 draws the same three lines, only this time a window opened on a low-resolution screen that has its own custom color palette. It also uses the ReadPixel routine to read each dot in a rectangular area that contains parts of the three lines and then resets each point to a new color. When the drawing is done in Program 3-5, the program waits for the user to click the mouse button and then closes the new window and screen.

Program 3-2. Drawing Lines and Points In C

```
#include <window.c>

demo()
{
int y;

   SetAPen (Rp,1);
   Move(Rp,50,50);
   Draw(Rp,150,100);

   SetAPen (Rp,3);
   Draw(Rp,50,150);

   SetAPen (Rp,2);
   for (y=50;y<151;y++)
   WritePixel(Rp,50,y);

}

/* end of Draw.c */
```

Program 3-3. Drawing Lines and Points In BASIC

```
LINE (50,50)-STEP (100,50)
'draw first with foreground pen 1
LINE STEP (0,0)-STEP(-100,50),3
'second line drawn with pen 3
FOR y = 50 TO 150
  PSET (50,y),2
  'third line with pen 2
NEXT
END
```

Program 3-4. Drawing Lines and Points on a Low-Resolution Screen

```
#include <windowl.c>

demo()
{
int x,y,Pen;

   SetAPen (Rp,1);
   Move(Rp,50,50);
   Draw(Rp,150,100);

   SetAPen (Rp,3);
   Draw(Rp,50,150);
   SetAPen (Rp,2);
   for (y=50;y<151;y++)
   WritePixel(Rp,50,y);

   for (x=48;x<101;x++)
   {
      for (y=50;y<151;y++)
         {
         Pen = ReadPixel(Rp,x,y);
         SetAPen(Rp,3-Pen);
         WritePixel(Rp,x,y);
         }
     }
}

/* end of Drawl.c */
```

Program 3-5. Drawing Lines and Points in BASIC on a Low-Resolution Screen

```
SCREEN 1,320,200,2,1 '320x200 low-res, 4 color Screen
WINDOW 2,,,0,1       'Full-screen window, no gadgets

PALETTE 0,1,1,1     'White background
PALETTE 1,1,0,0     'red
PALETTE 2,0,1,0     'green
PALETTE 3,0,0,1     'blue

'draw 1st with foreground pen (1)
LINE (50,50) - STEP (100,50)
'second line drawn with pen 3
LINE STEP (0,0) - STEP (-100,50),3
FOR y = 50 TO 150
  PSET (50,y),2     'third line with pen 2
NEXT

FOR y = 50 TO 150
  FOR x = 49 TO 100
    Pen = POINT(x,y)    'read pen for each point
    PSET (x,y), 3-Pen   'and complement
  NEXT x
NEXT y

WaitForClick:  IF NOT MOUSE(0) THEN WaitForClick

WINDOW CLOSE 2      'close the window
SCREEN CLOSE 1      'and the Screen
END
```

Drawing Polygons

One of the Graphics library routines, PolyDraw, can be helpful in drawing shapes composed of a number of connected lines. You give this routine a list of points on the screen as input, and PolyDraw draws a line from the current pen position to the first point on the list, then from the first point on the list to the next point, and the next, until lines have been drawn to all specified points. To call this routine, you use the form

```
PolyDraw(RastPort, Coordinate_pairs, Array_address);
            (a1)          (d0)              (a0)
```

where Array_address is a pointer to the beginning address of an array of *(x,y)* coordinate pairs. This array holds the *x* and *y* coordinates for a number of points on the screen, stored in the format of one 16-bit word for the *x* coordinate followed by another 16-bit word for the *y* coordinate. Such an array might look like this:

```
WORD points_array [ ] =
{
    180,50,
    210,80,
    10,120,
    180,150,
    100,150,
    70,120,
    70,80,
    100,50,
    10,10
};
```

The variable Coordinate_pairs holds the number of *(x,y)* pairs in the array. Since it takes two words of data to describe each point, the number in Coordinate_pairs should be half as large as the total number of words in the array. In the above example, 18 words are used in the array to describe nine points, so 9 is the appropriate number to use for Coordinate_pairs. A PolyDraw using this array would look like this:

```
PolyDraw (RastPort, 8, &points_array[0]);
```

To use the PolyDraw routine from BASIC, you must first open the Graphics library with the statement

```
LIBRARY "graphics.library"
```

(**Note:** When this LIBRARY statement is used, BASIC gets information about the location of the system graphics routines from a file called graphics.bmap. This file is included on the Amiga BASIC disk, in the BasicDemos directory, and must be present in the current disk directory when the program containing the LIBRARY statement is run.)

Once the library is open, you can call PolyDraw with the BASIC statement

CALL PolyDraw& (RP&, Coordinate_pairs, Array_address)

where RP& is the rastport address of the window, which can be found with the WINDOW(8) function. The other values specify the points to be drawn.

To use PolyDraw, you must first set up an array of short integers. This array must hold the coordinates of each point which is to be connected by a line. For instance, if you wanted to use PolyLine to draw a line from the current pen position to (100,100), then to (120,70), then to (90,50), you could set up an array called POINTS%( ), where POINTS%(0)=100, POINTS%(1)=100, POINTS%(2)=120, POINTS%(3)=70, POINTS%(4)=90, and POINTS%(5)=50. You would then call PolyDraw with the statement

CALL PolyDraw& (RP&,3,VARPTR(POINTS%(0))

The number 3 indicates that there are three pairs of coordinates. It is important to remember that the proper figure for the Coordinate_pairs value is not the size of the array, but the number of coordinate pairs (half the size of the array). The second value to pass is the address of the array, which can be found by using the VARPTR function.

PolyDraw uses the current location of the drawing pen as its starting point. This location depends on where the last point was drawn; if none was drawn, it defaults to (0,0). Rather than leaving things to chance, you will probably want to move the pixel cursor to the correct starting location before calling PolyDraw. This can be accomplished with a call to Move, a Graphics library routine that was described above. The proper way of calling this routine from BASIC (once the Graphics library has been opened, of course) is

CALL Move& (RP&, x&,y&)

where RP& is the address of the window RastPort (WINDOW(8)), and x& and y& are the horizontal and vertical coordinates at which the pixel cursor is set.

Although Move does not seem to have any effect on relative coordinates used with LINE and PSET (BASIC appears to keep track of its own internal pen position), it has a definite effect on the positioning of BASIC text. Preceding a PRINT statement with a call to Move allows you to position text at precise coordinates, rather than at a particular character position.

Drawing Octagons Using PolyDraw

Program 3-6 shows how to draw an eight-sided figure using PolyDraw from BASIC. It uses a custom low-resolution screen and shows how to use the COLOR statement with CLS to clear the window to a particular color. Program 3-7 is a similar program written in C.

Program 3-6. Using PolyDraw from BASIC

```
LIBRARY "graphics.library"

SCREEN 1,320,200,4,1
'320X200 low-res, 16 color screen
WINDOW 2,,,0,1
'Full-screen window, no gadgets
Rp&=WINDOW(8)
'Window's RastPort address
COLOR 9,1
'foreground to red, back to white
CLS
'clear screen to white

DIM points%(16)
FOR p=0 TO 15
  READ d
  points%(p)=d
  'put coordinate pairs in array
NEXT
  DATA 180,50,   210,80,  210,120,  180,150
  DATA 100,150,  70,120,  70,80,    100,50

CALL Move& (Rp&,100,50)
'move pixel cursor
CALL PolyDraw& (Rp&,8,VARPTR(points%(0)))
'draw polygon
```

```
WaitForClick: IF NOT MOUSE(Ø) THEN WaitForClick

WINDOW CLOSE 2
SCREEN CLOSE 1

END
```

Program 3-7. Using PolyDraw from C

```
#include <windowl.c>

demo()
{
static UWORD Points [] =
   {
    18Ø,5Ø,
    21Ø,8Ø,
    21Ø,12Ø,
    18Ø,15Ø,
    1ØØ,15Ø,
    7Ø,12Ø,
    7Ø,8Ø,
    1ØØ,5Ø
    };

   SetAPen (Rp,1);
   Move(Rp,1ØØ,5Ø);
   PolyDraw(Rp,8,&Points);


}

/* end of Polydraw.c */
```

Circles

In addition to the line drawing functions provided by the Graphics library, BASIC provides a CIRCLE statement, which can be used to draw circles, ellipses, and arcs. The syntax is

CIRCLE [STEP] (x,y),radius [,pen_number [,start_angle, end_angle [,aspect_ratio]]]

The only values that are required are the coordinates of the center point and the radius of the circle. The center coordinates may be expressed as an absolute—such as (50,50), which indicates a point 50 dots from the left edge and 50 dots from the top—or relative to the point where the last dot was drawn—for instance, STEP (50,50), 50 dots to the right and 50 dots below the last position drawn.

After the circle is drawn, the pixel cursor remains at the center point of the circle, even though no dot is drawn there. This means that you can draw concentric circles by specifying a center point of STEP (0,0) for each circle, as the following program demonstrates:

```
CIRCLE (100,100), 20
FOR R=27 TO 100 STEP 7
  CIRCLE STEP (0,0), R
NEXT
```

The radius value is the radius of the circle, expressed in pixels. Note that the vertical radius will probably not be the same as the horizontal radius. Since there are 640 pixels across and only 200 lines vertically on the Workbench screen, a circle that was 100 pixels wide and 100 pixels high would be tall and skinny, not round. Therefore, the CIRCLE statement automatically scales down the vertical radius to make the circle appear round. You can change this scaling with the aspect_ratio value, discussed below.

The pen_number value designates the pen that you want used to draw to the circle. If you do not specify a pen number, the foreground pen is used as a default.

The start_angle and end_angle values allow you to draw only a portion of the circle or ellipse. The values designate the starting and ending angles of the arc, expressed in radians. Since there are 2*pi radians in a circle, the permissible values range from 0 to 2*pi. The point described by a value of 0 is the rightmost point on the circle, that which we would normally think of being at 90 degrees. As the value increases, you move around the circle counterclockwise. The value for the point at the top of the circle is pi/2, that for the left of the circle is pi, and that for the bottom of the circle is pi*3/2. To convert degrees to radians, use the formula

radians = degrees/180 * pi

The starting angle may be smaller than the ending angle, but in either case, the arc will be drawn counterclockwise.

This means that the statement

CIRCLE (100,100), 70,, 0, 3.14*3/2

draws three-quarters of a circle (starting at the right and moving counterclockwise to the bottom). If you reverse the starting and ending points, however, the statement

CIRCLE (100,100), 70,, 3.14*3/2, 0

draws only a quarter circle (starting at the bottom and moving counterclockwise to the right side).

If either the start_angle or end_angle value is negative, the value will be treated as if it were positive, but that position on the arc will be connected by a line to the center of the circle. For example, this statement produces a wedge, with both ends of the arc connected to the center point.

CIRCLE (100,100), 70,, - 3.14*7/4, -.01

Reversing the start and end points gives you a circle with a wedge cut out of it.

The aspect_ratio value describes the scaling used to make the circle appear round instead of elliptical. Since the standard Workbench screen is 640 pixels wide, but only 200 lines tall, a circle which is as many dots tall as it is wide will be tall and skinny instead of round. So the width is multiplied by the aspect_ratio value to determine the height. The default value for a high-resolution, noninterlaced screen is .44, which means that vertical radius will be 44 percent as large as the horizontal radius. For a low-resolution, noninterlaced screen, the default is .88, twice as large, because the screen is half as many dots wide. An aspect_ratio value of less than the default creates an ellipse that is short and fat, while a value that is greater than the default creates an ellipse that is tall and skinny. If the aspect_ratio value is greater than 1, the horizontal radius may be shortened to preserve the ratio of height to width. This means that the ellipse created by the statement

CIRCLE (100,100), 70 ,,,, 10

will be narrower than the circle drawn by the statement

CIRCLE (100,100), 70

Patterned Lines

Until this point, the lines that we have been drawing have all been solid. The Amiga graphics hardware, however, is capable of drawing dotted lines as well. The pattern for line drawing can be up to 16 dots wide. It is stored as a 16-bit number in the rastport variable LinePtrn.

Although there is no direct Graphics library routine that sets the line pattern, the include file Graphics/Gfxmacros.h contains a macro routine that can be used. This macro routine directly manipulates the RastPort data structure. To use it, include the GRAPHICS/GFXMACROS.h file in your program, and invoke it with a statement like

```
SetDrPt (Window, Pattern);
```

where Window points to the address of the Window data structure, and Pattern is a 16-bit number that represents the line pattern.

We have discussed in previous chapters the way in which binary numbers relate to the patterns of color that appear on the computer display. For example, the number 65,535 in base two looks like this:

```
1111111111111111
```

Imagine this as a pattern for line drawing, where every one represents a position where a dot will be drawn with the foreground pen, and every zero represents a position where a dot will either be drawn with the color of the background pen or left alone, depending on the drawing mode selected. You can see that this pattern produces a solid line drawn with the foreground pen. The number 43,690, on the other hand, looks like this in binary format:

```
1010101010101010
```

This represents a pattern where one dot of foreground color alternates with one dot of background color, in other words, a dotted line.

Counting in binary is difficult for most of us, because of the long string of digits needed to represent relatively small numbers. The hexadecimal, or base 16, number system is somewhat easier to use when figuring out line patterns, since each digit corresponds to four dots. The following table shows the correspondence between dot patterns and hexadecimal digits:

Hex Digit	Dot Pattern
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111
8	1000
9	1001
A	1010
B	1011
C	1100
D	1101
E	1110
F	1111

By breaking the 16-dot mask down into 4-dot groups, we can figure out the patterns a little more easily. For instance, if we want a pattern where three dots of foreground color alternate with one of background, we choose the pattern that corresponds to E hexadecimal and repeat it four times. The macro statement to set such a pattern is

SetDrPt (Window, 0xEEEE);

The line pattern is used with all operating system and BASIC instructions that use the hardware line-drawing capabilities. This includes the BASIC LINE statement and PolyLine Graphics library routine.

The line pattern also affects the lines that are used to connect the starting and ending arcs to the center when negative values are used for start_angle and/or end_angle in the CIRCLE statement. Normally, you would not be aware of this, since the pattern is initialized to −1, which is the signed binary equivalent of 16 binary one bits, representing a solid line drawn in the foreground color. If you change the line pattern, however, all lines will be drawn with that pattern until you change it again. (Note—when you change colors with SetAPen or SetBPen, it sets the line drawer to restart the pattern—try this with a 12-dot segment of 15-dot pattern.)

Program 3-8 is a short C language program that demonstrates the use of the line pattern with the Graphics library line drawing commands.

From BASIC, you set the line pattern with the PATTERN statement. The syntax is

PATTERN [line_pattern] [,area_pattern]

The value which determines how lines are drawn is line_pattern. The other value, area_pattern, is used for pattern fills; it will be discussed in the section "Filled Shapes" that appears below. The line_pattern value is an integer expression that describes a mask that is 16 dots wide. For example, the BASIC equivalent of the SetDrMd example given above is

PATTERN &HEEEE

Program 3-9 is a BASIC program; note the affect PATTERN has on the line drawing. When you want to switch back to drawing a solid line, use PATTERN −1.

Program 3-8. Line Patterns in C

```
#include <windowl.c>
#include <graphics/gfxmacros.h>

demo()
{
int line;
static UWORD Points [] =
   {
    180,50,
    210,80,
    210,120,
    180,150,
    100,150,
    70,120,
    70,80,
    100,50
    };

   SetAPen(Rp,1);
   SetBPen(Rp,2);
   SetDrPt(Rp,0xF0F0);
```

```
    Move(Rp,100,50);
    PolyDraw(Rp,8,&Points);

    for (line=2; line<7; line++)
   {
    SetAPen(Rp,line);
    SetBPen(Rp,line+1);
    Move(Rp,100,line*20+20);
    Draw(Rp,180,line*20+20);
    }


}
```

Program 3-9. Line Patterns from BASIC

```
LIBRARY "graphics.library"

'Make output window full size
WINDOW 1,,(0,0)-(617,186),31,-1

Rp&=WINDOW(8)    'Window's RastPort address

DIM points%(16)
FOR p=0 TO 15
  READ d
  points%(p)=d
NEXT
  DATA 180,50,   210,80,  210,120,  180,150
  DATA 100,150,  70,120,  70,80,    100,50

PATTERN &HFF00                'even stripes
COLOR ,2                      'white and black
CALL Move& (Rp&,100,50)
'draw polygon
CALL PolyDraw& (Rp&,8,VARPTR(points%(0)))


PATTERN &HFFF0               'mostly foreground
'draw orange and black box
LINE (250,10) - STEP (350,170),3,b


PATTERN &HAAAA               'dotted pattern
COLOR ,3                     'white and orange
LINE (70,10)-(210,40)        'draw dotted line

PATTERN &HF0F0               'smaller stripes
COLOR ,0                     'default colors
CIRCLE (425,95), 100,1, -4.71,-3.14
'circle with wedge removed

END
```

Drawing Modes

The Amiga graphics drawing routine can operate in two primary drawing modes, which are mutually exclusive. So far, we have been doing all of our drawing in the default drawing mode, but that doesn't matter much, since the two modes have the same effect when drawing solid lines. But with the introduction of patterned lines (and text, as we shall see later), the subject of drawing modes becomes a good deal more relevant.

The default drawing mode that we have been using is known as the JAM2 mode, because in some instances this mode "jams" both the color of the foreground pen and the color of the background pen into display memory simultaneously. BASIC always selects JAM2 mode, so if you want to select one of the other modes described below, you must do so by using the operating system Graphics library routine for changing drawing modes.

In BASIC, the default color for the background pen is the same as that of the display background, so it is sometimes difficult to tell that two colors are being drawn at once. But if you enter the BASIC statement

COLOR ,3

in immediate mode, it is quite easy to see that both the text characters and the background are drawn at once (in this case, white letters on an orange background).

Another case in which the effects of JAM2 mode are evident is patterned line drawing. In JAM2 mode, all of the bits of the line pattern that are set to 1 are drawn with the foreground pen, and all of the bits of the line pattern that are set to 0 are drawn in the background pen. When drawing solid lines, the JAM2 mode does not quite live up to its name, since only the foreground pen color is used.

The other major drawing mode on the Amiga is known as JAM1 mode. As you might have guessed, in JAM1 mode, only one pen, the foreground pen, is used for drawing. This means that the area that would normally be drawn in with the background pen in JAM2 mode is left undisturbed by JAM1 mode. Using JAM1 mode, it is possible to superimpose text on a graphics image without blotting out a rectangular area of that image. Patterned lines turn out differently in JAM1 mode than in JAM2 mode, since only the bits of the pattern that contain ones will be colored, leaving the areas represented by zero bits as they are. Solid lines are drawn the same as in JAM2 mode, however, since only the foreground pen is used in either case.

In addition to these two drawing modes, there are two modes that modify their effect. The first is known as COMPLEMENT mode. In this mode, neither the foreground nor background pen is used. Instead, the color of each screen dot where a pen was supposed to draw is complemented. To complement the color of a pixel, you invert the bits of its pen number, changing all the ones to zeros and all the zeros to ones first.

For those who do not think in binary, another way of looking at the process is that you take the highest possible pen number, subtract the pen number of the current color, and you're left with the pen number of the new color. For instance, if the window that you're drawing on is attached to a screen that has three bit planes, there are eight drawing pens. These pens are numbered from 0 to 7. If you want to find the complement of pen 2, subtract 2 from 7, which leaves 5. If you are using four bit planes, the highest pen number is 15, so the complement of pen 2 would be pen 13.

COMPLEMENT mode is useful only with JAM1 mode. If you add COMPLEMENT mode to JAM2 mode, dots that are represented by zero bits are complemented along with the dots represeneted by one bits. The result is that the entire area is complemented, and you end up with solid lines instead of patterned lines and solid rectangles instead of text. When COMPLEMENT mode is used with JAM1 mode, however, only the area represented by one bits is complemented.

The other mode that can be used to modify JAM1 and JAM2 is called INVERSVID. Used mostly for text, INVERSVID reverses the roles of the foreground pen and background pen. If you use the INVERSVID mode along with JAM1 to draw text, the background area surrounding the letters will be colored in with the foreground pen, while the area where the characters themselves would normally be drawn is left untouched. If COMPLEMENT mode is added to the combination of INVERSVID and JAM1, the area that is represented by zero bits is the one that is complemented. When used with JAM2 mode, INVERSVID merely reverses the colors of foreground and background pens.

The drawing modes can be set with the operating system routine SetDrMd (short for Set Draw Mode). A call to this routine takes the form

```
SetDrMd (RastPort, Mode);
           (a1)      (d0)
```

From BASIC, the syntax to use is

CALL SetDrMd& (RP&,Mode&)

where RP& is the address of the window's rastport—WINDOW(8)—and Mode& is a value equal to the combination of values which represent the various modes desired. While C programmers can use the names of the modes, which have been defined in the graphics include files, BASIC programmers can use the following table to find the correct value for the modes desired.

Mode	Value
JAM1	0
JAM2	1
COMPLEMENT	2
INVERSVID	4

You may add these values to form any combination. To select a combination of JAM1, COMPLEMENT, and INVERSVID modes, for example, you would use the number 6 (0+2+4).

Program 3-10 is a C program and should help you visualize the effects of the various drawing modes. The Text routine was used to add text to better illustrate the effects of these modes. This function is explained more fully in Chapter 4. Program 3-11 is a similar program written in BASIC.

Program 3-10. Drawing Modes, C Example

```
#include <windowl.c>
#include <graphics/gfxmacros.h>

demo()
{

   SetAPen (Rp,4);
   RectFill(Rp,88,1Ø,247,11Ø);

   SetAPen (Rp,2);
   SetBPen (Rp,3);
   SetDrPt (Rp,ØxFFØØ);

   Move(Rp,2Ø,2Ø);
   SetDrMd (Rp,JAM1);
   Text(Rp, "This is JAM1 mode ",18);
   Draw(Rp,3ØØ,2Ø);

   Move(Rp,2Ø,3Ø);
   SetDrMd (Rp,JAM1+INVERSVID);
   Text(Rp,"This is INVERSE  ",17);
   SetDrMd (Rp,JAM1);
   Draw(Rp,3ØØ,3Ø);

   Move(Rp,2Ø,55);
   SetDrMd (Rp,JAM2);
   Text(Rp, "This is JAM2 mode ",18);
   Draw(Rp,3ØØ,55);

   Move(Rp,2Ø,65);
   SetDrMd (Rp,JAM2+INVERSVID);
   Text(Rp,"This is INVERSE ",16);
   Draw(Rp,3ØØ,65);

   Move(Rp,2Ø,9Ø);
   SetDrMd(Rp,COMPLEMENT);
   Text(Rp,"This is COMPLEMENT mode ",24);
   Draw(Rp,3ØØ,9Ø);

   Move(Rp,2Ø,1ØØ);
   SetDrMd (Rp,COMPLEMENT+INVERSVID);
   Text(Rp,"This is INVERSE ",16);
   Draw(Rp,3ØØ,1ØØ);

}
```

Program 3-11. Drawing Modes, BASIC Example

```
LIBRARY "graphics.library"
DEFLNG a-z  'all long integers
```

```
SCREEN 1,320,200,3,1 '320*200 low-res
'8-color screen
WINDOW 2,,,0,1  'full screen window
'no gadgets

PALETTE 0,1,1,1  'White background
PALETTE 1,1,0,0  'red--foreground pen
PALETTE 2,0,1,0  'green--background pen
PALETTE 3,0,0,1  'blue
PALETTE 4,1,1,0  'yellow--complement
PALETTE 7,0,0,0  'black--complement

Mode$(0)="JAM1 mode"
Mode$(1)="INVERSID JAM1"
Mode$(2)="JAM2 mode"
Mode$(3)="INVERSID JAM2"
Mode$(4)="COMPLEMENT mode"
Mode$(5)="INVERSID COMP"


LINE (92,17)- STEP (170,120),3,bf
'draw blue box for contrast
PATTERN &HFF00 'striped pattern
COLOR 1,2  'set colors to red and green

FOR Row = 0 TO 5  'for 6 lines
   y=((Row\2)+1)*5+(Row MOD 2 = 0)
   LOCATE y,4   'position for print
   Mode = (Row\2) - 4*(Row MOD 2 = 1)
   CALL SetDrMd& (WINDOW(8),Mode)
   'set drawing mode
   PRINT "This is ";Mode$(Row)
   'print text in this mode
   LINE (219,y*8-4)- STEP (85,0)
   'draw line in this mode
NEXT


WaitForClick:IF NOT MOUSE(0) THEN WaitForClick
WINDOW CLOSE 2
SCREEN CLOSE 1
END
```

Filled Shapes

Not only can the Amiga graphics routines draw lines and shapes, but they can fill them with color as well. We have already seen one example of filled shapes in the BASIC LINE statement. As you may remember, if you add the letters *bf* to the end of this statement, a filled box is drawn. For example, this statement draws a box that is 200 pixels × 100 pixels, using the color of the foreground pen:

LINE (20,10) - STEP (200,100),,bf

The operating system routine which performs the equivalent function is called RectFill. The blitter assists this operation, drawing the filled rectangle almost instantly. The syntax for this routine is

```
RectFill (RastPort, X1, Y1, X2, Y2);
            (a1)    (d0) (d1) (d2) (d3)
```

where the point X1,Y1 is the location of the upper left corner of the filled box, and the point X2,Y2 is the location of the lower right corner. In the default case, the rectangle will be solidly filled with the color of the foreground pen (APen), but this may be altered by changing the area pattern and drawing mode, as we shall see later.

The Graphics library also provides routines that allow you to create nonrectangular filled shapes. These area fill routines, as they are called, let you create a list of points to be connected by lines, like the list used by the PolyFill routine. Then, they use the list both to draw the figure and fill it in, either with the color of the foreground pen, or using a fill pattern, as we shall see later.

The area fill routines (and the flood fill routine described below) require that you set up some temporary work space before you use them. First, you must declare a data structure called an AreaInfo structure for use by the rastport and a buffer area to be used by the AreaInfo structure, like this:

```
struct AreaInfo AInfo;
WORD Buffer [200];
```

Then, you must initialize the AreaInfo structure by calling the InitArea routine:

InitArea (&AInfo, &Buffer, Max_points);

where &AInfo is a pointer to your AreaInfo structure, and &Buffer is a pointer to the buffer area. The Max_points variable specifies the maximum number of points that can be described for your filled area. Since each point uses five bytes, the maximum number of points equals the size of the buffer divided by 2.5 (the buffer is composed of two-byte words, since it must be lined up on a word boundary). In the example above, the Max_points variable would be set to 80 points, because there are 200 words (400 bytes) in the buffer area.

After you initialize the AreaInfo structure, you must tie it into your window's rastport by pointing the RastPort variable AreaInfo at the now-initialized structure:

```
RastPort ->AreaInfo = AInfo;
```

The AreaInfo structure is used to store the list of points that make up your filled shape. But the area fill routines also need a temporary work space in which to construct the image. Because we are dealing with windows, the image can't be drawn directly on the screen since part of the window may be obscured. Therefore, the image must be created in a temporary work space where it can be clipped to fit the window's display space upon moving it to the display window. Typically, this space will be as large as a single bit plane for the screen you are using so that an image as large as the display may be created. Since a single bit plane on the WorkBench screen requires 16,000 bytes, this can be a substantial amount of memory; you may wish to reduce the size of the buffer somewhat if you know that your image will be smaller.

The graphics work space is associated with a data structure called a TmpRas structure. As with the AreaInfo structure, your program must declare a TmpRas structure:

```
struct TmpRas TRas;
```

The actual work space associated with the TmpRas structure must be located in the lower 512K of memory, since the blitter chip can access only that portion of memory. Although most people have only 512K of memory (for now, at least), it makes sense to take the necessary steps to make sure that your program will work on systems with expansion memory as well. Therefore, instead of simply declaring an array for buffer space, you should use the Graphics library routine AllocRaster to properly allocate an area of graphics memory. This function is executed with the statement

```
Raster = AllocRaster (Width, Height);
                      (d0)   (d1)
```

The Width and Height variables give the size of the bit plane to be allocated. The Width is the number of pixels across, and the Height is the number of lines the display occupies. If the function is able to reserve for itself the proper amount of chip memory (as the lower 512K is called), it returns a pointer to the beginning of the display buffer.

So, you might allocate a buffer for the TmpRas structure as shown below:

```
PLANEPTR RBuffer;
RBuffer = (PLANEPTR)AllocRaster(640,200);
```

There are two important points to note about the AllocRaster function. The first is that if the system is unable to allocate the necessary memory, the routine returns a zero. You should therefore check the result of the AllocRaster operation and abort your program if the buffer cannot be allocated. The second point is that when you have allocated memory, it is up to you to free that memory once you are finished with it. This can be accomplished with the FreeRaster function, which is executed like this:

```
FreeRaster(Raster, Width, Height);
            (a0)    (d0)   (d1)
```

where Raster is the pointer returned by the AllocRaster routine, and Width and Height are the exact same values used by the original AllocRaster call. If you do not deallocate the memory, it will continue to be reserved even after your program ends, so be sure to free all of the memory you have allocated.

Once the buffer is allocated, the TmpRas structure must be initialized with a call to InitTmpRas:

```
InitTmpRas (&TRas, Rbuffer, Size);
            (a0)    (a1)    (d0)
```

where TRas and Rbuffer are the pointers to the TmpRas structure and buffer described above. The Size variable is the size of the buffer in bytes. Since AllocRaster allocates a bit plane whose width is measured in bits, you have to convert to bytes to find the size of the plane. A C macro, RASSIZE, can perform the conversion for you. For instance, you could find the size of a 640 × 200 bit plane with the statement

```
Size = RASSIZE(640,200);
```

The TmpRas structure must also be linked into the RastPort. We can combine both the initialization and linking steps into one statement:

```
RastPort->TmpRas = (struct TmpRas *)
    InitTmpRas(&TRas, Rbuffer, RASSIZE(640,200) );
```

After you've initialized both the AreaInfo and TmpRas structures, you may begin using the area fill commands. The first of these is AreaMove. AreaMove is used to start a new shape by defining the starting (and ending) point for that shape. The syntax for this statement is

```
AreaMove (RastPort,  X,   Y);
          (a1)      (d0) (d1)
```

where RastPort is a pointer to the RastPort structure, and *x* and *y* are the coordinates of the point. If another shape is already in progress when you call AreaMove, that shape will be completed (but not drawn) and the new one started.

The AreaDraw procedure is used to add another point to the shape. This procedure can be called with the statement

```
AreaDraw (RastPort,  X,   Y);
          (a1)      (d0) (d1)
```

Despite its name, AreaDraw does not do any drawing. The actual drawing does not happen until an AreaEnd statement is executed. The syntax for such a call is

```
AreaEnd(RastPort);
        (a1)
```

AreaEnd completes the current shape, and causes all of the shapes that have been defined to be drawn and filled. Note that unlike PolyDraw, you do not have to define the endpoint of the shape to get a closed figure. AreaEnd automatically completes the shape by joining the last point defined to the first one. By default, the area will be solidly filled in the color of the foreground pen. It is possible, however, to use a two-color patterned fill, as we shall see later.

Program 3-12 shows all of the steps needed to draw a filled version of the octagon we drew in the PolyDraw example.

Program 3-12. Filled Octagon, C Example

```
#include <window.c>

demo()
{

int count;
UWORD AreaBuf [200];
PLANEPTR TBuf;
struct TmpRas TRas;
struct AreaInfo AInfo;

static UWORD Points [] =
    {
     180,50,
     210,80,
     210,120,
     180,150,
     100,150,
     70,120,
     70,80
     };

   InitArea(&AInfo,AreaBuf, 80);
   Rp->AreaInfo = &AInfo;
   if ( (TBuf = (PLANEPTR)AllocRaster(640,200)) == NULL)
       exit(FALSE);
   Rp->TmpRas = (struct TmpRas *)InitTmpRas
     (&TRas,TBuf,RASSIZE(640,200) );

   SetAPen (Rp,1);
   AreaMove(Rp,100,50);
   for (count=0; count <14; count+=2)
      AreaDraw(Rp,Points[count],Points[count+1]);
   AreaEnd(Rp);

   FreeRaster(TBuf,640,200);

}

/* end of Areafill.c */
```

Area Fill from BASIC

The BASIC equivalent of the Graphics library area fill routines are AREA and AREAFILL. You use AREA like AreaMove and AreaDraw, to specify each point of the filled shape individually. The syntax is

AREA [STEP] (x,y)

The only value that you must specify is a coordinate for one of the points of the filled polygon. This coordinate may be expressed as an absolute position, for example, (10,20), or relative to the last point drawn, for example, STEP (10,–20).

To draw a filled polygon with AREAFILL, issue an AREA statement for each point of the polygon in the order in which you want the points drawn. You do not have to specify the starting point twice since the last point will automatically be connected to the first point. A maximum of 20 points may be used to define the polygon. If more AREA statements are used, all but the first 20 are ignored. When enough AREA statements have been given to describe all of the points in the polygon, use the AREAFILL statement to connect the points and fill the polygon.

Program 3-13 is a BASIC program that shows how to draw a filled version of the eight-sided figure used in the PolyDraw example.

Program 3-13. Filled Octagon, BASIC Example

```
FOR p=Ø TO 7
    READ x,y
    DATA 18Ø,5Ø,  21Ø,8Ø, 21Ø,12Ø, 18Ø,15Ø
    DATA 1ØØ,15Ø, 7Ø,12Ø, 7Ø,8Ø,   1ØØ,5Ø
    AREA (x,y)       'AREA for each coordinate pair
    NEXT

AREAFILL  'draw filled shape
```

Flood Fill

The last of the shape filling commands is a general-purpose flood fill routine. Unlike the previous commands that we've discussed, a flood fill does not first draw a shape and then fill it in. Rather, it colors in an existing enclosed area. By default, the area will be solidly filled with the color of the foreground pen, but as we shall see later, patterned filling is also possible.

Flood filling operates in one of two modes. In outline mode, the entire area enclosed by a border of the outline color is filled. Filling begins at the point which you specify and continues in all directions. As the fill moves outward, every horizontally and vertically adjacent pixel which is not colored with the pen designated as the area outline pen (AOlPen) is filled. The fill pattern stops spreading at each point where it encounters a pixel that is the color of the AOlPen. If the area of the fill is not completely surrounded by the outline color, the fill will "leak" out, and the entire window will be filled. We will discuss how to change the color of the area outline pen in the next section.

In color mode, all adjacent pixels of the same color are filled. You designate the point at which filling begins, and whatever color is located at that point becomes the color which the fill routine displaces. As the fill moves outward, every horizontally and vertically adjacent pixel which is colored with the displacement pen is filled. The fill stops spreading at each point where a pixel drawn in another pen color is encountered.

The syntax for the flood fill routine is

```
Flood (RastPort, Mode, X,   Y);
          (a1)        (d2)  (d0) (d1)
```

where *x* and *y* specify the coordinate at which the fill begins, and Mode specifies the fill mode (0 = outline mode, 1 = color mode). It is important to remember that like the area filling routines, Flood uses the AreaInfo and TmpRas structures. Therefore, you must always initialize an AreaInfo and TmpRas for the RastPort of a window in which you intend to do flood filling before any filling actually takes place.

Program 3-14 gives examples of both types of flood filling in C. It outlines four boxes in white, and then fills them in various colors, using the outline method. It then fills the center box in white and uses the color method to fill all the white areas on the screen, including the outline of the other four boxes.

Program 3-14. Flood Fill from C

```
#include <windowl.c>
#include <graphics/gfxmacros.h>
demo()
{

int count;
UWORD AreaBuf [200];
PLANEPTR TBuf;
struct TmpRas TRas;
struct AreaInfo AInfo;

static UWORD Points [] =    /* coordinates for boxes */
   {
      110,20,210,70,
      10,70,110,120,
      110,120,210,170,
      210,70,310,120
      };

   InitArea(&AInfo,AreaBuf, 80);
   Rp->AreaInfo = &AInfo;
   if ( (TBuf = (PLANEPTR)AllocRaster(640,200)) == NULL)
     exit(FALSE);
   Rp->TmpRas = (struct TmpRas *)InitTmpRas
     (&TRas,TBuf,RASSIZE(640,200) );

/* Draw four boxes outline in white, */
/* and flood fill them with different colors */

   SetOPen (Rp,1);
   for (count=0; count<16;count+=4)
      {
      Move(Rp,Points[count],Points[count+1]);
      SetAPen (Rp,1);
      Draw(Rp,Points[count+2],Points[count+1]);
      Draw(Rp,Points[count+2],Points[count+3]);
      Draw(Rp,Points[count],Points[count+3]);
      Draw(Rp,Points[count],Points[count+1]);
      SetAPen (Rp,2+(count/4));
      Flood(Rp,0,9+Points[count],9+Points[count+1]);
      }

/* Now, fill the center box with white */
/* and fill all white areas with purple */

SetAPen (Rp,1);
Flood(Rp,0,170,100);  /* use outline fill mode */

SetAPen (Rp,6);
Flood(Rp,1,170,100);  /* use color fill mode */

   FreeRaster(TBuf,640,200);

}

/* end of Areafill.c */
```

BASIC's Flood—PAINT

The BASIC version of the flood fill is the PAINT statement, which supports only the outline mode of filling. The syntax for the PAINT is

PAINT [STEP] (x,y) [,fill_pen [,border_pen]]

The only required value is the coordinates of the point at which the filling begins. The coordinates may be expressed as an absolute location or relative to the location of the last dot that was drawn.

The two optional values that you may specify are fill_pen, the number of the pen which is used to do the filling, and border_pen, the number of the pen at which the filling stops. The default value for the fill_pen is that of the foreground pen, while the border_pen defaults to the same value as is currently in fill_pen.

PAINT uses the outline method for filling. If the shape that you choose to PAINT is not completely enclosed by the border color, the fill color will escape through the gap and spread out to cover the entire window. Likewise, if you have not specified a border_pen that matches the border color, the fill will proceed right through the border.

There are some more serious concerns associated with using PAINT. The statement will not work with a window set for smart refreshing of the screen. If you try to use PAINT in a window which was opened with an attribute value of greater than 15, you'll crash the system. Since the default output window has an attribute value of 31, it is not safe to use PAINT in that window unless you reopen it with a WINDOW statement giving a lower attribute value.

Another point to watch for is specifying coordinates for PAINT that lie outside the window boundaries. This is particularly easy to do when you're specifying relative coordinates. Such a PAINT statement may fill areas of memory that do not belong to the display, and this can crash the system.

The following example draws a circle, PAINTS it white, and PAINTS the rest of the window orange.

```
WINDOW 1,,(0,)-(300,186),15   'Reopen output window to type 15
CIRCLE (150,100),100          'draw the circle
PAINT STEP (0,0)              'fill it with foreground pen
PAINT (0,0),3,1               'fill rest of screen with pen 3
```

Fill and the Area Outline Pen

We've talked about the functions of the rastport's foreground pen (APen) and background pen (BPen), but so far have only mentioned the area outline pen (AOlPen) in passing. This pen has two functions relevant to the fill commands. First, as noted above, it is used to designate a stop color for the outline mode of the flood fill command. Second, it can be used to designate a color to be used for drawing an outline around an area fill or RectFill shape.

Although there are Graphics library statements that can be used to change the color of the foreground and background pens, there is no equivalent statement to change the color of the area outline pen. There is a C language macro contained in the file graphics/gfxmacros.h, however, that can be used in place of such a function. To change the value in the AOlPen, use the statement

SetOPen (RastPort, pen);

This macro really performs two functions. First, it changes the value of the AOlPen variable in the rastport. Its second function is to change the Flags variable, setting a flag called AREAOUTLINE. This flag indicates that every shape created by the area fill routine should have an outline drawn around it in the color of the AOlPen. So, if you set the AOlPen with the SetOPen macro, you will automatically get outlines around your area fill shapes. To turn the outlining off, you must use another C macro statement:

BOUNDARY_OFF (RastPort);

To set area outlines from BASIC, you must use POKEs to perform the work done by the SetOPen macro. SetOPen does two things. First, it sets the color in the RastPort variable AOlPen. Then, it sets the AREAOUTLINE flag (which has a value of 8) in the Flags variable of the RastPort. Looking at the C language definition of the RastPort structure, we see that the AOlPen variable comes at an offset of 27 bytes from the beginning of the structure. Since the address of the RastPort is returned by the WINDOW(8) function, AOlPen = WINDOW(8)+27. Likewise, the Flags variable comes at an offset of 32 bytes from the start of the RastPort, so Flags = WINDOW(8)+32. Once we know these two locations, we can use the statement

```
POKE AOlPen,Pen
```

to set the color of the area outline pen. We can use the statement

```
POKEW Flags, PEEK(Flags) OR 8
```

to set the AREAOUTLINE flag. Note that we use an OR statement so as not to disturb the other Flags setting, and we also use a POKEW statement since Flags is a 16-bit variable.

To turn area outlining off, reset Flags with this statement:

```
POKEW Flags, PEEK(Flags) AND 8
```

Here, then, is the entire series of statements needed to set area outlining from BASIC:

```
AOlPen = WINDOW(8)+27
Flags = WINDOW(8)+32
POKE AOlPen, 3 'use pen 3 for border
POKEW Flags, PEEK(Flags) OR 8 'turn outlining on
```

Be sure to turn off outlining before your program ends so as not to disrupt the function of other programs. If you forget, you may find that text will not be printed properly in the BASIC output window, and the computer may even crash.

Patterned Fills

We have already seen how the SetDrPt macro could be used to set a pattern to be used in line drawing. Similarly, the SetAfPt macro can establish a pattern to be used for filled shapes.

The process of setting up the fill pattern is a little more complex since a two-dimensional area is involved. The area pattern is still 16 bits wide, but it is several lines high as well. You can choose the height of the pattern yourself, but you must stick to a power of 2 (2 lines, 4 lines, 8 lines, 16 lines, and so forth). Since you're working with a screen with a maximum height of 200 lines, don't make the pattern more than 64 lines high.

You may remember that the line pattern is a 16-bit number that represents a pattern of 16 dots. The area fill pattern may be thought of as an array of 16-bit patterns, stacked one on top of the other. You determine the values to be placed in this array in the same way that you determine the line pattern value. It may help to visualize the pattern if you write it out in binary digits, using ones to stand for dots filled with the foreground color and zeros to stand for dots filled with the background color. For example, let's look at a pattern that draws the letters *HI:*

```
0000000000000000 = 0x0000
0110011001111110 = 0x667E
0110011000011000 = 0x6618
0111111000011000 = 0x7E18
0110011000011000 = 0x6618
0110011000011000 = 0x6618
0110011001111110 = 0x667E
0000000000000000 = 0x0000
```

As you can see, we drew the pattern using zeros and ones, and then converted the resulting binary numbers to hexadecimal numbers. To set up an area fill pattern using these values, we first put them in an array:

```
WORD Pattern [ ] =
{
   0x0000,
   0x667E,
   0x6618,
   0x7E18,
   0x6618,
   0x6618,
   0x667E,
   0x0000
};
```

Then, we use the SetAfPt macro statement

```
SetAfPt (RastPort, &Pattern[0], Height_exp);
```

where &Pattern[0] is a pointer to the first byte of the array, and Height_exp is the exponent part of the height of the pattern expressed as a power of 2. As we have stated, the height of the pattern must equal 2 raised to some power. So if the pattern is 8 lines high (2^3), the Height_exp value is 3, and if the pattern is 32 lines high (2^5), Height_exp is 5.

Once the pattern is set, any RectFill, AreaEnd, or Flood operation will use this pattern to fill the designated area. The colors used for the fill will depend on the drawing mode chosen. When JAM2 is selected, the pixels represented by ones in the bit pattern are drawn in the color of the foreground pen, and the pixels represented by zeros are drawn with the background pen. When JAM1 is selected, the pixels represented by zeros are not affected. And when both JAM1 and COMPLEMENT are chosen, the pixels represented by ones will be complemented.

Program 3-15 demonstrates the use of the area fill pattern in C.

Program 3-15. Area Fill Pattern from C

```
#include <window1.c>
#include <graphics/gfxmacros.h>
demo()
{

UWORD AreaBuf [200];
PLANEPTR TBuf;
struct TmpRas TRas;
struct AreaInfo AInfo;

static WORD Points [] =   /* coordinates for polygon */
   {
   195,60,
   230,90,
   230,130,
   195,160,
   125,160,
   90,130,
   90,90,
   125,60
   };

static UWORD Pat1 [] =    /* 'HI' fill pattern */
   {
   0x0000,
   0x667E,
   0x6618,
   0x7E18,
   0x6618,
   Øx6618,
   Øx667E,
   ØxØØØØ
   };

static UWORD Pat2 [] =   /* geometric pattern */
   {
   ØxØFFØ,
   ØxFØØF,
   ØxAAAA,
   Øx5555,
   ØxA5A5,
   Øx5A5A,
   ØxFØFØ,
   ØxØFØF
   };

static UWORD Pat3 [] =   /* random fill pattern */
   {
   ØxØ48C,
   Øx159D,
   Øx26AE,
   Øx37BF,
   Øx3333,
   ØxAAAA,
   Øx3C3C,
   ØxD43D4
   };

/* initialize temporary data structures and buffers */

   InitArea(&AInfo,AreaBuf, 8Ø);
   Rp->AreaInfo = &AInfo;
   if ( (TBuf = (PLANEPTR)AllocRaster(64Ø,2ØØ)) == NULL)
       exit(FALSE);
   Rp->TmpRas = (struct TmpRas *)InitTmpRas
     (&TRas,TBuf,RASSIZE(64Ø,2ØØ) );

/* draw a filled triangle with Area commands */

   SetAfPt(Rp,&Pat2[Ø],3);
   SetAPen(Rp,2);
   SetBPen(Rp,5);

   AreaMove(Rp,28Ø,1ØØ);
   AreaDraw(Rp,28Ø,18Ø);
   AreaDraw(Rp,2ØØ,18Ø);
   AreaEnd();

/* draw a filled rectangle with RectFill */

   SetAfPt(Rp,&Pat1[Ø],3);
   SetAPen(Rp,4);
   SetBPen(Rp,3);
   SetOPen(Rp,1);   /* outline it in white */

   RectFill(Rp,15,15,80,96);

/* Draw a polygon with PolyDraw and Flood fill it */

   SetAfPt(Rp,&Pat3[0],3);
   SetAPen(Rp,6);
   SetOPen(Rp,6);
   SetBPen(Rp,7);

   Move(Rp,125,60);
   PolyDraw(Rp,8,&Points);
   Flood(Rp,0,160, 100);  /* use outline fill mode */
   FreeRaster(TBuf,640,200);

}

/* end of Fillpat.c */
```

Pattern Fill in BASIC

In BASIC, the same PATTERN statement that is used to set a pattern for line drawing can also be used to set the pattern for area filling. The area fill pattern should be stored in an array of 16-bit (short) integers. First, DIMension the array to a power of 2. For example, the proper DIM statement for an eight-element array called Pattern% is

DIM Pattern%(7)

since the array starts with element 0.

Next, you must determine the values with which to fill the array, as demonstrated above. Once we have determined the values for the pattern elements, we assign these values to the array Pattern%. Then, we set the area fill pattern to the values stored in this array with the statement

PATTERN ,Pattern%( )

When the pattern is set, BASIC makes its own internal copy of the array. Pattern% is no longer needed unless you want to set the pattern to another array and change back later. You may ERASE the array after the PATTERN statement is given in order to free up memory.

Program 3-16 fills a box with the HI pattern that we designed above. Notice that at the end of the program, we set the area fill pattern back to a solid pattern. If we had not done so, the cursor in our default output window would have been rendered difficult to see. We do not have to change the pattern back at the end of the next example because it opens its own window, rather than using the default window. Each window has its own private line pattern and area fill pattern.

The area fill pattern is used with all of the commands that produce filled shapes. Program 3-17 demonstrates the three different kinds of patterned fills: boxes, AREAFILLs, and PAINTing.

Program 3-16. Filling a Box with a Pattern

```
WINDOW 1,,(0,0)-(250,186)
'size the output window


DIM pat%(7)
FOR p=1 TO 6
   READ d
   pat%(p)=d
   'put the pattern into an array
   NEXT
   DATA &h667e, &h6618, &h7e18
   DATA &h6618, &h6618, &h667e

PATTERN ,pat%
'use the pattern for fills
LINE (16,32)-STEP(192,96),,bf


FOR p=0 TO 7
   pat%(p)=-1
NEXT
PATTERN -1,pat%
'return the pattern to solid
END
```

Program 3-17. Boxes, AREAFILLs, and PATTERNs

```
SCREEN 1,320,200,4,1
'16-color, lo-res
WINDOW 2,,,0,1
'full-sized window
```

```
PALETTE Ø,Ø,Ø,Ø
'black background


DIM pat%(7)
'pattern array has 8 elements


COLOR 9,14
f=Ø:GOSUB Fillpat
'set pattern
CIRCLE (16Ø,1ØØ),7Ø
PAINT (16Ø,1ØØ)
'flood fill


COLOR 5,6
f=&HA5A5:GOSUB Fillpat
'set pattern
LINE (1Ø,1Ø)-STEP(6Ø,9Ø),,bf
'rectfill


COLOR 11,9
f=&H5555:GOSUB Fillpat
'set pattern
AREA (28Ø,1ØØ)
AREA STEP (Ø,8Ø)
AREA STEP (-8Ø,Ø)
AREAFILL
'area fill


WaitForClick:  IF NOT MOUSE(Ø) THEN WaitForClick

WINDOW CLOSE 2
SCREEN CLOSE 1
WINDOW OUTPUT 1
END


Fillpat:
'create a repeating or random fill pattern
RANDOMIZE TIMER
FOR p=Ø TO 7
 IF f=Ø THEN pat%(p)=RND*6ØØØØ&-3ØØØØ ELSE pat%(p)=f
NEXT
PATTERN ,pat%
RETURN
```

Multicolor Fill Patterns

So far, we've just been using two-color patterns to fill our shapes. With a little extra effort, however, you can produce a multicolor fill pattern that fills shapes with as many colors as your screen allows.

The first step is to create the display data for the pattern. The process is the same as that for a two-color pattern, except here you must provide a complete pattern for every bit plane used. For example, if you are drawing on a screen that is three bit planes deep, and you want to create a fill pattern that is two lines high, you must supply six words of pattern data. The first four words in the pattern are drawn in plane 0, the next four words in plane 1, and the last four in plane 2. Therefore, if your data looked like this, the top line of the pattern would be one solid color, and the bottom line would be another solid color:

```
UWORD Filpat [ ] =
     {
     0x0000,    /* data for plane 0 */
     0x0000,

     0x0000,    /* data for plane 1 */
     0xffff,

     0xffff,   /* data for plane 1 */
     0xffff,
```

To find out which color is used, we group the corresponding bits from each plane as three-bit numbers. In this example, the bits from the top line of planes 0 and 1 are all set to 0. That means the rightmost two digits of the number will be 0. The bits from the top line of plane 2 are all set to 1. That means the leftmost digit of the number will be 1. Therefore, the pen number used to color the top line of the pattern will be 100 binary, or 4. The bits from the second line of the pattern are set to 1 (plane 2), 1 (plane 1), 0 (plane 0). This is equivalent to the binary number 110, or 6, so pen 6 will be used to color the bottom line of the pattern.

The other step you must take is to specify the Height_exp value as a negative number. In the example above, the height of the pattern is 2 (2^1), so Height_exp would normally be 1.

But since this is a multicolor pattern, we specify the height exponent as −1 instead. The SetAfPt macro statement for this example would look like this:

SetAfPt (RastPort, &filpat, −1);

Program 3-18 is a C program that sets up an eight-color fill pattern on our low-resolution custom screen. The pattern is in the shape of a grid, with four rows of two colors each. Sample shapes are created using all three fill modes, rectfill, area fill, and flood fill.

Program 3-18. Shapes with Multicolor Patterns, C Example

```
#include <windowl.c>
#include <graphics/gfxmacros.h>
demo()
{

UWORD AreaBuf [200];
PLANEPTR TBuf;
struct TmpRas TRas;
struct AreaInfo AInfo;

static WORD Points [] =    /* coordinates for polygon */
   {
   195,60,
   230,90,
   230,130,
   195,160,
   125,160,
   90,130,
   90,90,
   125,60
   };

static UWORD Pattern [] =    /* grid fill pattern */
   {
   0x0000, 0x0000, 0x0000, 0x0000,
   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
   0x0000, 0x0000, 0x0000, 0x0000,
   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,

   0x0000, 0x0000, 0x0000, 0x0000,
   0x0000, 0x0000, 0x0000, 0x0000,
   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,
   0xFFFF, 0xFFFF, 0xFFFF, 0xFFFF,

   0xFF00, 0xFF00, 0xFF00, 0xFF00,
   0xFF00, 0xFF00, 0xFF00, 0xFF00,
   0xFF00, 0xFF00, 0xFF00, 0xFF00,
   0xFF00, 0xFF00, 0xFF00, 0xFF00
   };
/* initialize temporary data structures and buffers */

   InitArea(&AInfo,AreaBuf, 80);
   Rp->AreaInfo = &AInfo;
   if ( (TBuf = (PLANEPTR)AllocRaster(640,200)) == NULL)
       exit(FALSE);
   Rp->TmpRas = (struct TmpRas *)InitTmpRas
     (&TRas,TBuf,RASSIZE(640,200) );

/* set pens, drawmode, and fill pattern */

   SetAfPt(Rp,&Pattern[0],-4);
   SetAPen(Rp,255);
   SetBPen(Rp,0);
   SetDrMd(Rp,JAM2);

/* draw a filled triangle with Area commands */

   AreaMove(Rp,280,100);
   AreaDraw(Rp,280,180);
   AreaDraw(Rp,200,180);
   AreaEnd();

/* draw a filled rectangle with RectFill */

   SetOPen(Rp,1);  /* outline it in white */

   RectFill(Rp,15,15,80,96);

/* Draw a polygon with PolyDraw and Flood fill it */

   SetOPen(Rp,255);

   Move(Rp,125,60);
   PolyDraw(Rp,8,&Points);
   Flood(Rp,0,160, 100);  /* use outline fill mode */

   FreeRaster(TBuf,640,200);

}

/* end of Colorpat.c */
```

Multicolor Fill Pattern from BASIC

Using a multicolor fill pattern from BASIC is a bit more difficult. BASIC uses the PATTERN in place of the SetAfPt macro, and this command is very particular about the type of input it receives. It requires that the pattern array you use be DIMensioned to a size equal to a power of 2. But when you use a multicolor fill pattern, you must supply patterns for a number of bit planes, and each pattern must be a power of 2 in length. If you want to use a 16-line pattern on a screen that is three planes deep, you must provide three 16-word patterns, or 48 words of data in all. But if you try to DIMension your array to 48 elements, PATTERN will give you an *Illegal function call* error message.

The solution to this problem is fairly simple. You must DIMension your array to the next largest power of 2. For our 48-word example, you must DIM the array to 64 elements and fill only the first 48. Remember, since by default arrays start with element 0, the way to DIM a 64-element array is with the statement

DIM pat%(63)

Of course, if you have used the OPTION BASE 1 statement to make your arrays start with element 1 rather than element 0, you would use the statement

DIM pat%(64)

The other problem is how to set the area pattern size to a negative number. PATTERN sets the pattern size to the power of 2 that is appropriate for the size of your array. If you DIMension a 64-element array, it sets the pattern size to 6 (2^6=64). For a 16-line multicolor pattern, you need to specify a pattern size of −4. Since you can't DIMension the array to a negative size, you must take a different approach. The SetAfPt macro puts the size of the pattern into a rastport variable called AreaPtSz. The BASIC equivalent would be to POKE this value into the rastport.

As we have mentioned, POKEing data into Intuition data structures should not be your first choice of programming methods, since the possibility exists that the composition of those data structures will change in the future. As a practical matter, however, the chance of such a change is slight, and there is no other way to used multicolor fills from BASIC. So all that remains is to find the address of the AreaPtSz variable.

From the C declaration for the RastPort data structure, we can tell that the AreaPtSz variable is located at an offset of 29 bytes from the beginning of the RastPort. The WINDOW(8) function can be used to find the address of the RastPort. Therefore, the address for AreaPtSz is equal to WINDOW(8)+29.

The next problem is POKEing a negative number. Because of the way that BASIC stores numbers internally, negative numbers cannot be represented with fewer than 16 bits. The AreaPtSz variable is only an 8-bit number, however, so we cannot POKE it with a larger value. So we must cut this negative number down to size. To do this, we can use the AND operator to mask off the top 8 bits. The proper expression to use is the negative number AND 255. Therefore, to POKE the number −4 into the AreaPtSz variable, you would use the statement

POKE WINDOW(8)+29, −4 AND 255

The final step is to set the foreground color to the maximum pen value and the background color to 0 (JAM2 is already our default drawing mode). The WINDOW(6) function tells us what the maximum drawing pen number is for that window. So we can set the pens correctly with the statement

COLOR WINDOW(6),0

regardless of how many bit planes are used by the screen.

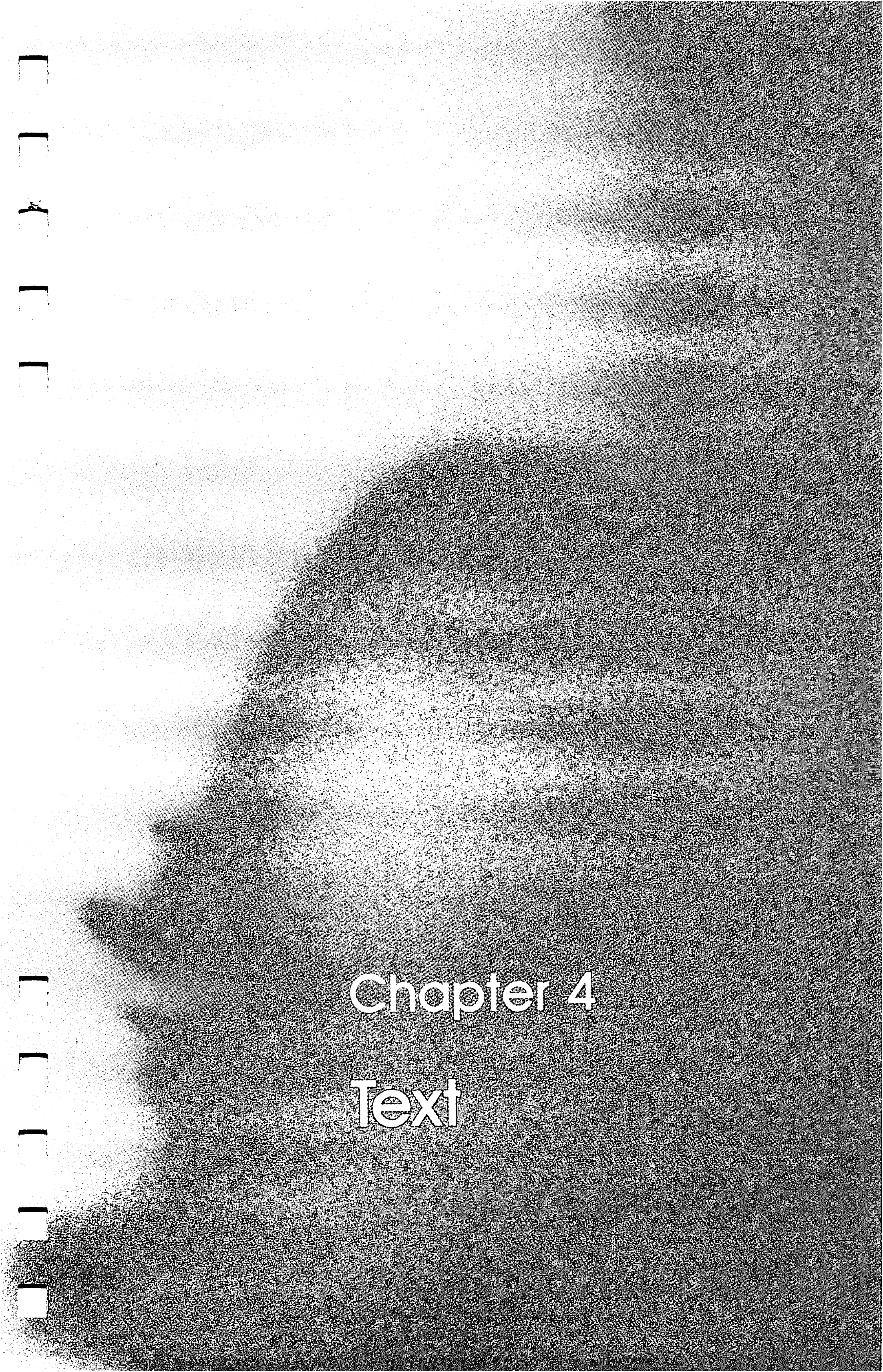
From here on in, the multicolor area pattern is used just like the two-color variety. Program 3-19 is a BASIC program which uses the eight-color pattern that we created for the C example (Program 3-18). It fills shapes with this pattern using all three fill modes (LINE bf, AREA, and PAINT).

Program 3-19. Shapes with Multicolor Patterns, BASIC Example

```
'*******************************************
'*  Multicolor pattern fill program        *
'*  uses an 8 color fill pattern to show   *
'*  how multicolor fills can be performed  *
'*  using the AREA, PAINT, and LINE bf     *
'*  commands                               *
'*                                         *
'*******************************************

GOSUB InitScreen

' Dimension a 64 byte pattern array,
' and fill first 48 bytes without pattern
' (3 planes of 16 bytes each)
```

```
DIM pat%(63)            'dim pattern array

FOR Pbyte=Ø TO 47       'partially fill array
     READ patdat        'with our data
     pat%(Pbyte) =  patdat
NEXT

PATTERN ,pat%           'set pattern
ERASE pat%              'we no longer need array

' The following code performs the work
' of the C Macro SetAfPt (pat%,-3).
' Since there is no library call,
' we must POKE the size value directly
' to the Rastport.

     Rp=WINDOW(8)       'Rastport address
     AreaPtSz = Rp+29 'Area Pattern size

POKE AreaPtSz,-4 AND 255
'-4 is pattern size,
' 16 words (4^2) per bit plane

REM--Here are the 48 words of data
REM--for grid pattern

'Plane Ø

DATA &hØØØØ,&hØØØØ,&hØØØØ,&hØØØØ
DATA &hffff,&hffff,&hffff,&hffff
DATA &hØØØØ,&hØØØØ,&hØØØØ,&hØØØØ
DATA &hffff,&hffff,&hffff,&hffff

 'Plane 1

DATA &hØØØØ,&hØØØØ,&hØØØØ,&hØØØØ
DATA &hØØØØ,&hØØØØ,&hØØØØ,&hØØØØ
DATA &hffff,&hffff,&hffff,&hffff
DATA &hffff,&hffff,&hffff,&hffff

'Plane 2

DATA &hffØØ,&hffØØ,&hffØØ,&hffØØ
DATA &hffØØ,&hffØØ,&hffØØ,&hffØØ
DATA &hffØØ,&hffØØ,&hffØØ,&hffØØ
DATA &hffØØ,&hffØØ,&hffØØ,&hffØØ


   COLOR WINDOW(6),Ø
   'It is important that
   'PenA = maxpen, PenB = Ø
'Now that the pattern is set up,
'we draw three shapes and use
'different fills to fill them
```

```
CIRCLE (160,100),70
PAINT (160,100)    'flood fill

LINE (10,10)-STEP(60,90),,bf   'rectfill

AREA (280,100)
AREA STEP (0,80)
AREA STEP (-80,0)
AREAFILL           'area fill


WaitForClick:
   IF NOT MOUSE(0) THEN WaitForClick

WINDOW CLOSE 2
SCREEN CLOSE 1
WINDOW OUTPUT 1
END

InitScreen:
    SCREEN 1,320,200,3,1 '8-color, lo-res
    WINDOW 2,,,0,1

    PALETTE 0,0,0,0 'black
    PALETTE 1,1,1,1 'white
    PALETTE 2,0,0,1 'blue
    PALETTE 3,1,0,0 'red
    PALETTE 4,0,1,0 'green
    PALETTE 5,1,1,0 'yellow
    PALETTE 6,1,0,1 'purple
    PALETTE 7,0,0,1 'cyan


    RETURN
```



Chapter 4

Text

Text

We normally don't think of the text that appears on a computer screen as graphics, but on the Amiga there is really very little difference between text and any other kind of graphics. Since text is drawn on the screen like any other image, graphics and text images may be mixed freely. The various display modes discussed in Chapter 1 and the drawing modes discussed in Chapter 3 all affect text output.

For C programmers, there are actually two methods of producing text on the Amiga. The first method is to attach a console device to your output window. This device provides an interface much like that of the old style Teletype terminal (TTY). From the standpoint of the program, text is sent to the console as it would be to a disk drive or any other device.

The console device takes this text and outputs it sequentially to the screen. It takes care of the details involved in maintaining a clean display, such as making sure that the borders are not overwritten, updating the block cursor, returning to the next line after the end of the current line, and scrolling the display when the last line is filled. So, while the device itself uses the graphics system to render the text on the screen, your program need not be aware of the details of text drawing.

Although the console device takes care of a lot of the work involved in text-intensive displays, it also takes over control of much of the display. For example, with the console device, you can specify the placement of text only by character position; you cannot achieve the pixel-by-pixel precision attainable through direct control of text. Graphics-intensive applications will therefore probably need to take advantage of the other method of producing text on the Amiga. This method involves drawing the text on the screen just as you would any other graphics image.

Text as Graphics

Using the Graphics library routines to draw text is very similar to using them for line drawing. The basic library routine for text rendering is called Text; it's executed like this:

```
Text(RastPort,Text_string,Chars);
        (a1)       (a0)          (d0)
```

where the variable Text_string points to a string of ASCII characters, and the variable Chars specifies the number of characters to be written. The text is printed at the current pen position. As we mentioned in Chapter 3, this position may be changed by using the Move( ) library routine. After the text has been printed, the horizontal pen position is moved to the end of the last character printed. The vertical position remains the same.

When you print text with the Text routine, the size and position of the text image depends a lot on the font that is used. The three most important factors to consider are the height of the font, its width, and the baseline.

The height of the font tells how many lines the characters in that font occupy vertically. If a font is *h* pixels high, you must print each line of text at least *h* pixels below the previous line so as not to overwrite part of that line.

The width of the font measures the number of pixels the average character in the font occupies horizontally. For *monospaced* fonts like the two standard system fonts, each character will actually be that wide. But the Amiga also supports proportionally spaced fonts in which the widths of individual characters may vary. For example, in a proportional font, the letters *i* and *l* can be much thinner than *m* and *w*. For proportional fonts, the width value gives an approximation of the average character width.

Finally, the baseline value specifies how the character will be positioned vertically. Since each character can be several lines tall, there must be some means of deciding if the vertical coordinate of the character position actually specifies the top line of the character, the bottom line, or somewhere in between. The baseline is the font characteristic that specifies how much of the character extends above the specified vertical position of the character and how much below.

The height, width, and baseline values for the font currently used by a particular window are kept track of in the RastPort structure for that window. Your program can learn these values by using the following statements:

```
Height = Rp->TxHeight;
Width = Rp->TxWidth;
Baseline = Rp->TxBaseline;
```

The two system fonts that are included in the operating system are known as Topaz 8 and Topaz 9. The number after the name of the font designates the height of the font, so each character of the Topaz 8 font is 8 lines high, and each character of the Topaz 9 font is 9 lines high. This means that a maximum of 25 lines of Topaz 8 characters can fit on a 200-line noninterlaced screen, while only 22 lines of Topaz 9 characters fit on the same size screen. The width of Topaz 8 characters is 8 pixels, so a maximum of 80 such characters can fit on a single high-resolution (640 pixel) line, 40 on a low-resolution (320 pixel) line. Topaz 9 characters are 10 pixels wide, so 64 of them can fit on a high-resolution line, and 32 on a low-resolution line.

The baseline for each of these fonts is 6. That means that each character extends six pixels above the point specified as the vertical coordinate for the character. In Topaz 8, the seventh line, the baseline, is the last line used to draw most characters, and the final line is usually left blank to leave a space between lines. Only characters that descend below the line, such as *g* and *y* are drawn down past the baseline. Since the Topaz 9 has the same baseline, though it is one line taller, it has two lines below the baseline. This means that even for characters with descenders, there is a blank line between the bottom of that character and the top of the character on the next line.

As with other drawing commands, the Text routine puts the burden on the user to make sure that the resulting image lies inside the window boundaries and does not overwrite the borders and gadgets. Because most of each character appears above the baseline, you must be sure to leave room for the top of the character when you position it vertically. This means that the minimum vertical position for a character is equal to the baseline. For the default fonts, you should not place a character higher than line 6, or the top might be clipped off.

Even when you offset your character six lines from the top, it still will be printed at the top of the window. Unless you are using a Gimmezerozero type window, in which the borders are drawn in their own separate layer, your text will overwrite the title bar. To avoid this, you must offset your line of text by the number of lines occupied by the title bar. To find out how much room is taken up by border graphics, you may use the Window structure variables BorderLeft, BorderTop, BorderRight, and BorderBottom. For instance, to find out how much room the title bar takes up at the top of the window, you could use the statement

BarHeight = Window->BorderTop;

Program 4-1 is a C program that illustrates correct (and incorrect) placement of text.

Program 4-1. Text Routines in C

```
#include <window.c>

demo()
{
UWORD h,b;
BYTE t;

   SetAPen(Rp,1);
   SetBPen(Rp,Ø);

   h = Rp->TxHeight;
   b = Rp->TxBaseline;
   t = Wdw->BorderTop;

   Move(Rp,5Ø,b);
   Text(Rp,"This overwrites the title bar",29);

   Move(Rp,5Ø,b+t);
   Text(Rp,"This is the first clear line of of text",39);

   Move(Rp,5Ø,b+t+h);
   Text(Rp,"This is the next line of text",29);
}

/* end of Text.c */
```

Keeping Text in the Window

In order to make sure that your text does not run off the right edge of the window, you may have to calculate the length of your text before it is printed. With the monospaced system fonts, this is simple, because the length of the text in pixels is eight or ten times the number of characters used, depending on whether Topaz 8 or 9 is the font used.

With proportionally spaced fonts, the task is harder, since each character has its own individual width. The Graphics library provides a function that will perform the calculation for you. It can be called with the statement

```
Length = TextLength(RastPort,Text_string,Chars);
 (d0)                  (a1)      (a0)      (d0)
```

where the input is the same as for the Text routine, and Length is the width of the text line in pixels. TextLength only calculates what the length of the string would be if it were actually printed. Since text that goes past the edge of the window will be clipped, it is possible that the part of the string that is actually printed is much shorter than Length.

As we mentioned earlier, the colors in which the text is drawn depend on the same factors as any other drawing, the colors in the foreground and background pens, and the drawing mode currently in use. In the default drawing mode, JAM2, the color of the foreground pen is used to draw the text itself, and the color of the background pen is used to draw the background behind the text characters.

Text in BASIC

The situation is somewhat different in BASIC, since the PRINT statement, the primary method for displaying text, works more along the lines of the console device than the Text( ) routine. However, because of the variable size of the output window, programmers still have some responsibility for making sure that the text is printed within the confines of the window in which they are working. For example, try typing the following one-line BASIC program in the immediate mode:

```
FOR x=0 TO 255:PRINT x;:NEXT
```

You'll see that the PRINTed output goes right off the edge of the screen. To insure that this does not happen in your program, use the WIDTH command to set the maximum line width. When used for this purpose, the syntax of the command is

WIDTH linesize [,print_tab]

where linesize is the maximum line length, and print_tab is an optional value that specifies the width of the columns used when a comma is added to the end of a PRINT statement.

The maximum line length depends both on the width of your window and the size of the text font that you are using. If you have set the 80-column font as the default using the Preferences program, then each character will be eight dots wide, resulting in a maximum line width of 80 characters for a high-resolution window and 40 for a low-resolution window. Actually, since some room is taken up by the border drawn around the window and the sizing gadget (if present), this will be reduced to 75 or 76 characters in high resolution and half that number in low resolution. If you have set the 60-column font as the default, then the width of each character will be ten pixels, and the maximum number of characters must be reduced accordingly.

This assumes, of course, that you are using a full-size window and that the window cannot be sized. If this is not the case, your program may have to check the size of the window in order to set the output WIDTH. The WINDOW(2) function returns the current width of the window in pixels. Therefore, to find the maximum line length, you can divide the WINDOW(2) figure by 8 or 10, depending on whether you are using the 80- or 60-column font.

This raises the related question of how your program can tell which size of system font it is using. The default character size is determined by the settings of the Preferences program, so there is no clear way to tell whether BASIC will start up in the 60-column (9 point) or 80-column (8 point) font. Since each letter of the former font is larger than the corresponding character of the latter, text that has been carefully positioned for one mode might appear completely out of line in the other.

One way around this problem is to find out what size font is being used and adjust your program output accordingly. As we said in the previous section, the height of the text font is stored in a rastport variable called TxHeight. According to the definition for the RastPort data structure, this variable comes at an offset of 58 bytes from the beginning of the structure. Therefore, a quick, if not elegant, way of finding out the height of the font being used in the output window is to use the statement

```
Height = PEEKW ( WINDOW(8) +58 )
```

If Height is equal to 8, you know that the Topaz 8 (80-column) font is being used, and if it is 9, the Topaz 9 (60-column) font is the default. A safer alternative, and one that we recommend, is to call the Graphics library routine AskFont. This routine will be discussed a little later.

Once you have found out what size text is being used by the program, you can insure accurate placement of text by using PTAB. The LOCATE statement, which is normally used for text placement, moves the text cursor to even character positions. The absolute coordinates of these character positions may vary according to the size of the text font. But PTAB moves the text cursor to an absolute pixel location. Its syntax is

```
PTAB(x)
```

where *x* is the horizontal coordinate for the text cursor. If you wish to position the text at an absolute vertical coordinate as well, you must use the operating system routine Move, which was demonstrated in Chapter 3 in the explanation of the PolyDraw routine in the "Drawing Polygons" section. Remember that the vertical position you specify will be used to determine the point where the baseline of the text is placed.

A simpler solution, perhaps, to the problems raised by the placement of variable-sized text is to have your program itself open a new window and specify the font to be used in that window. This procedure is described in the next section.

Like other BASIC graphics figures, the color of text depends on the setting of the foreground and background pens. You can change the color of text by using the COLOR statement that we described in Chapter 3 to reset these foreground and background pens.

Changing Fonts

Changing from one system font to another is a multistep process. First, you must set up a TextAttr (Text Attribute) data structure that describes the font that you wish to use. Next, you must open the font with the OpenFont routine. Then, you must specify that font as the one currently used by your window's rastport with the SetFont routine.

The syntax for the OpenFont routine is

```
FontPtr = OpenFont(TextAttr);
  (d0)                (a0)
```

where TextAttr is a pointer to a TextAttr data structure. This structure provides a description of the font that you wish to open. The C definition for the TextAttr structure looks like this:

```
struct TextAttr
   {
   STRPTR ta_Name;
   UWORD  ta_Size;
   UBYTE  ta_Style;
   UBYTE  ta_Flags;
   };
```

The first field, ta_Name, is where you specify the name of the font. This name is composed of the lowercase ASCII characters of the font name, followed by the characters *.font,* and ending with an ASCII 0. For example, the correct format for the name of the system fonts is topaz.font.

The next variable, ta_Size, contains the height of the font in lines. The ta_Style field contains flags that specify whether this font is designed as a normal typeface or as a special style of font. Possible special-style flags include FSF_NORMAL (0), FSF_UNDERLINED (1), FSF_BOLD (2), FSF_ITALICS (4), and FSF_EXTENDED (8).

Finally, the ta_Flags variable contains a number of flags that provide information about the origin and intended use for the font. These include FPF_ROMFONT (1), FPF_DISKFONT (2), FPF_REVPATH (4), FPF_TALLDOT (8), FPF_WIDEDOT (16), FPF_PROPORTIONAL (32), FPF_DESIGNED (64), and FPF_REMOVED (128). The first two indicate whether the font is in ROM or DISK-based. The next flag specifies that the font is designed to be printed from right to left (REVerse PATH). The next two indicate whether the font was designed specifically for high or low resolution, interlaced or not. The PROPORTIONAL flag indicates that the width of each character is specified individually. DESIGNED means that the font was designed and not generated by using some formula to modify an existing font. Finally, REMOVED means that the font is not currently linked into the system.

Therefore, to open the standard 80-column text font, you could use these statements:

```
Struct TextFont FontPtr;
struct TextAttr StdFont = { "topaz.font", 8, 0, 0};
FontPtr = OpenFont(&StdFont);
```

This indicates that you wish to open the Topaz font, eight lines high, normal style, and with no special preference flags. The system tries to find the best possible match to your description. If no match is found, the OpenFont routine returns a zero. If the font you described, or one close to it, is found, the routine returns a pointer to the TextFont structure that describes the font it found. In determining what the best match is, the system first matches the font name. It then tries to match the height, style, and flags fields, in that order. You can check to see whether the font that was found matched your request exactly by looking at the fields tf_YSize, tf_Style, and tf_Flags in the TextFont structure that was returned.

Once you're sure that a nonzero pointer to the TextFont structure was returned by OpenFont, you may use the SetFont routine to begin using that font. A call to SetFont takes the form

```
SetFont(RastPort, FontPtr);
        (a1)      (a0)
```

where FontPtr is the TextFont structure pointer that was returned by OpenFont. Program 4-2 shows how to write to a window using both sizes of the standard system font in the C language.

Notice that when we finished with the font that we opened, we used the CloseFont routine, which takes this form:

```
CloseFont(FontPtr);
          (a1)
```

That's because OpenFont not only supplies us with the address for the TextFont structure, but also marks the font as being in use by our application. This is significant because the operating system may unload a non-ROM font (like a disk-loaded font) if it is not in use and additional free memory is required. While the system fonts will not be unloaded, it is nonetheless a good idea to get into the habit of closing fonts after you are done with them. The exception to this, as we shall see a bit later, is with disk-loaded fonts under version 1.1 of the operating system, which should not be closed because of a system bug.

As was pointed out in the last section, if you do not specify a default font for your screen or window, you will get the system font set by the Preferences program. You can find out which font is being used by the current window with the function AskFont, which takes the form

```
AskFont(RastPort, TextAttr);
           (a0)     (a1)
```

where TextAttr is a pointer to an empty TextAttr structure that you have set up to receive the information about the current font. Once the statement has been executed, you can check the contents of this structure to determine the name, height, style, and preference flags for the current font. (See Program 4-2.)

Changing System Fonts in BASIC

To specify the font to be used in a window, you must use the operating system routines OpenFont, SetFont, and CloseFont. The first step is to use OpenFont to get a pointer to a font descriptor. Since this call returns a value, you must use the DECLARE FUNCTION statement as well as opening the Graphics library with the LIBRARY statement.

(Remember: When a call to the Graphics library is used, BASIC gets information about the location of the system graphics routines from a file called graphics.bmap. This file is included on the Amiga BASIC disk in the BasicDemos directory and must be present in the current disk directory when the program containing the LIBRARY statement is run.)

Program 4-2. Writing to a Window in C

```
#include <window.c>

demo()
{
struct TextFont *FontPtr;
static struct TextAttr SysFont =
  {"topaz.font",TOPAZ_EIGHTY,Ø,Ø};

   SetAPen(Rp,1);
   SetBPen(Rp,Ø);

   FontPtr = (struct TextFont *)OpenFont(&SysFont);
   if (FontPtr == Ø) exit(FALSE);
   SetFont(Rp,FontPtr);
   Move(Rp,5Ø,6Ø);
   Text(Rp,"This is the system Topaz font, 8 lines high",43);
   CloseFont(FontPtr);

   SysFont.ta_YSize = TOPAZ_SIXTY;
   FontPtr = (struct TextFont *)OpenFont(&SysFont);
   if (FontPtr == Ø) exit(FALSE);
   SetFont(Rp,FontPtr);
   Move(Rp,5Ø,8Ø);
   Text(Rp,"This is the system Topaz font, 9 lines high",43);
   CloseFont(FontPtr);


}

/* end of Sysfont.c */
```

The proper syntax for the OpenFont call is

FontPtr& = OpenFont&(VARPTR(textAttr&(0)))

The one value that must be supplied to the OpenFont command is the address of a TextAttr data structure. In BASIC, this can be set up as a long integer array having two elements. The first element of the array holds the address of a text string, ending with an ASCII 0, that names the font. In the case of the system fonts, the name is topaz.font. The other element of the array holds the height of the font, the style, and the preference flags. For our purposes, we can ignore the style and preference flags. The form of the text attribute array therefore is

textAttr&(0) = SADD("topaz.font"+CHR$(0))
textAttr&(1) = height*65536&

where height is either 8 (for the 80-column font) or 9 (for the 60-column font).

Once FontPtr& for a particular font has been found, it can be used to set that font for use in a particular window with the SetFont call. The syntax of that call is

CALL SetFont&(Rp, FontPtr&)

where RP is the address of the window's RastPort structure—found by the WINDOW(8) function—and FontPtr& is the pointer found by the OpenFont call.

Finally, when you are through with a font that you've opened, you should close it with the CloseFont call:

CALL CloseFont&(FontPtr&)

Programs 4-3 and 4-4 are in BASIC. Program 4-3 opens a window and writes one sentence in each of the two system fonts. Program 4-4 is an example of using the AskFont routine to find out which font is currently in use.

Program 4-3. Writing to a Window in BASIC

```
'This program prints both system fonts

DEFLNG a-z
'all variables default to long integer
DECLARE FUNCTION OpenFont LIBRARY
```

```
LIBRARY "graphics.library"
WINDOW 2,"System Fonts",(100,50)-(525,100),12
WIDTH 41

FOR height=8 TO 9
  textAttr(0)=SADD("topaz.font"+CHR$(0))
  textAttr(1)=height*65536&
  IF FontPtr THEN CloseFont FontPtr
  FontPtr=OpenFont(VARPTR(textAttr(0)))
  IF FontPtr THEN SetFont WINDOW(8),FontPtr
  PRINT
  PRINT " This shows the system font";height;
  PRINT "points high"
NEXT height

WINDOW OUTPUT 1
END
```

Program 4-4. Using AskFont from BASIC

```
'This program identifies the current system font

DEFLNG a-z
'all variables default to long integer
LIBRARY "graphics.library"

DIM TextAttr(2)
AskFont WINDOW(8),VARPTR(TextAttr(0))
height = TextAttr(1)\65535&

a=1:x=0

WHILE(a)
'get font name
  a=PEEK(TextAttr(0)+x)
  IF a > 0 THEN n$=n$+CHR$(a)
  x=x+1
  WEND

  PRINT "The current font it '";n$;

  PRINT "',";height;"lines high"
```

Software-Generated Font Styles

It would be nice if all the style variations for the font you're using were available as separately defined fonts, but this is very rarely the case. The operating system can provide some help, however. It contains a routine that can take the font that you're working with and algorithmically generate the styles that are not designed into the fonts. Let's say that you're using a normal font, like one of the system fonts, and want to print italics. The SetSoftStyle routine can take your normal font and "bend" every character as it is printed, according to a certain formula, to make it appear as though the font were italicized. This routine can currently emulate all of the possible styles, with the exception of extended (which provides expanded, or double-wide, print). The syntax for the Softstyle call is

```
Result = SetSoftStyle(RastPort, Style, Enable);
 (d0)                   (a1)      (d0)   (d1)
```

where Style is the combination of all the flags for the styles that you want set. For example, if you wanted the text to appear in boldface (2) and underlined (1), you would set Style to 3. The Enable value is a mask that specifies which of the style features is to be generated algorithmically (as opposed to style features that are inherent in the design of the font). After all, you wouldn't want to try to italicize an italics font. The Enable mask should have bits set for each of the styles that can be generated by the operating system software.

An easy way of discovering what styles can be generated from software for the current font is to use the AskSoftStyle routine, which follows this format:

```
Enable = AskSoftStyle(RastPort);
 (d0)                   (a1)
```

The Enable value returned by this routine can be used as the mask for SetSoftStyle, if you want your style setting to affect all possible flags.

You can also use the Enable mask to affect only a single software style, leaving the rest as they are. This allows you to make settings cumulative, so if you set underline with one call and bold with the next, the latter will not cancel out the former.

The SetSoftStyle routine returns the value Result, which represents the combination of flags for the style that was actually generated. This result may be different from the style you requested if the software was unable to comply with your request, or if you merely added one flag to the existing settings.

Program 4-5 uses the SetSoftStyle routine to change the font styles of the current system font. Program 4-6 is written in BASIC and shows how to use the SetSoftStyle routine to run through all the possible style combinations for the two system fonts.

Although the operating system does a pretty good job of manipulating normal fonts to create the various styles, you may find that if you're writing text a character at a time, sometimes the beginning of one character will blank out the end of the previous character. The solution to this problem is to join all the characters that you intend to write on a single line into one long string before you use Text( ) to print them. The Text( ) routine can adjust the intercharacter spacing if it is dealing with an entire phrase at once.

Disk-Based Fonts

Using disk-based fonts is very similar to using the system fonts, but there are some important differences. The first difference is that in order to use a disk-based font, the data describing that font must be present on your system disk.

To be more specific, in order to use a disk font, there must be two files describing that font located in the fonts: directory. The name *fonts:* is a logical device name that is assigned to the fonts: directory of your system disk at startup time. The first file has the same format as the name of the font as specified in the TextAttr data structure. For example, if the font name is Ruby, there must be a file called ruby.font in the fonts: directory. This file describes each separate font size and style available for the font and gives the filenames of the data files that actually contain the font shape information.

The font data file is the second file that is required. It is usually located in a separate subdirectory under the fonts: directory and bears the name of the font size. For example, the data describing the Ruby font that is eight lines high is found in the file fonts:ruby/8.

Another important difference between using the system fonts and disk-based fonts is that you must use the OpenDiskFont command to open the font, rather than the OpenFont routine. The OpenDiskFont command is found in the Diskfont library, and not the Graphics library. Therefore, to open a disk font, you must first open the Diskfont library. From C, you use the OpenLibrary routine to do this, just as you would for the Graphics library:

```
DiskfontBase =
    OpenLibrary("diskfont.library",LIBRARY_VERSION);
```

The Diskfont library is disk-based, so in order to open it, you must have the file diskfont.library in the libs: directory of your system disk. Once you open this library, you have access to the two routines that it contains. The first is OpenDiskFont. The format for this statement is similar to that of OpenFont:

```
FontPtr = OpenDiskFont(TextAttr);
  (d0)                  (a0)
```

OpenDiskFont not only returns a pointer to the TextFont data structure that describes the font, but it also loads the font into memory from the disk. All of the fonts described in the fontname.font file are loaded at the same time. Thus, if you open another size of the same font later, a disk access will not be necessary.

Once you have opened the font successfully, you may use it the same way as you would a system font, by executing a SetFont statement. Program 4-7 shows how to use C to load the Ruby font from disk and display text in two sizes of that font.

Notice that we did not close the font after using it. Because of a bug in the 1.1 version of the system software, a disk-based font is not correctly purged from the font list when it is unloaded. This means that a later attempt to access the font could result in the system trying to use a font that is no longer there. For the present time, the simplest fix is to leave the font open. Kickstart versions 1.2 and higher should fix the bug, so if you're using a version later than 1.1, remember to close the disk font when you are through with it.

Program 4-5. Changing Fonts in C

```
#include <window.c>

demo()
{
   SetAPen(Rp,1);
   SetBPen(Rp,Ø);

   PrintAt(5Ø,6Ø,"This is the NORMAL style of the system font");

   SetSoftStyle(Rp, FSF_UNDERLINED, 255);
   PrintAt(5Ø,8Ø,"This is the UNDERLINED style");

   SetSoftStyle(Rp, FSF_ITALIC, 255);
   PrintAt(5Ø,1ØØ,"This is the ITALIC style");

   SetSoftStyle(Rp, FSF_BOLD, FSF_BOLD);
   PrintAt(5Ø,12Ø,"This is both BOLD and ITALIC styles");

}

PrintAt(x,y,s)
int x,y;
char *s;

{
    Move(Rp,x,y);
    Text(Rp,s,strlen(s));

}

/* end of SoftStyle.c */
```

Program 4-6. Changing Fonts in BASIC

```
'This program prints all styles of both system fonts

DEFLNG a-z
DECLARE FUNCTION OpenFont LIBRARY

LIBRARY "graphics.library"
WINDOW 2,"Fonts and Styles",(Ø,3)-(3ØØ,182),8
WIDTH 3Ø
FOR height = 8 TO 9
  GOSUB ChangeFont
  GOSUB PrintStyles
  IF FontPtr THEN CALL CloseFont(FontPtr&)
NEXT height

WINDOW OUTPUT 1
LIBRARY CLOSE
END

PrintStyles:
  PRINT "System font,";height;"points high"
  FOR Style = Ø TO 7
    CALL SetSoftStyle (WINDOW(8),Style,255)
    IF Style AND 4 THEN PRINT "Italicized ";
    IF Style AND 2 THEN PRINT "bold ";
    IF Style AND 1 THEN PRINT "underline";
    IF Style=Ø THEN PRINT "plain";
    PRINT
  NEXT Style
PRINT
RETURN
```

```
ChangeFont:
  FontName$="topaz.font"+CHR$(Ø)
  TextAttr(Ø)=SADD(FontName$)
  TextAttr(1)=height*65536&
  FontPtr = OpenFont(VARPTR(TextAttr&(Ø)))
  IF FontPtr THEN SetFont WINDOW(8),FontPtr&
RETURN
```

Program 4-7. Loading a Font in C

```
#include <window.c>
#include <exec/libraries.h>

struct Library *DiskfontBase;

demo()
{
struct TextFont *FontPtr;
static struct TextAttr RubyFont =
  {"ruby.font",8,Ø,Ø};

   DiskfontBase = (struct Library *)
         OpenLibrary("diskfont.library",LIBRARY_VERSION);
   if (DiskfontBase == NULL) exit(FALSE);

   SetAPen(Rp,1);
   SetBPen(Rp,Ø);

   FontPtr = (struct TextFont *)OpenDiskFont(&RubyFont);
   if (FontPtr == Ø) exit(FALSE);
   SetFont(Rp,FontPtr);
   PrintAt(5Ø,6Ø,"This is the disk-based Ruby font, 8 lines high");
  RubyFont.ta_YSize = 12;
  FontPtr = (struct TextFont *)OpenDiskFont(&RubyFont);
  if (FontPtr == 0) exit(FALSE);
  SetFont(Rp,FontPtr);
  PrintAt(50,100,"This is the disk-based Ruby font, 12 lines high");

  CloseLibrary(DiskfontBase);

}

PrintAt(x,y,s)
int x,y;
char *s;

{
    Move(Rp,x,y);
    Text(Rp,s,strlen(s));

}

/* end of Diskfont.c */
```

Disk-Based Fonts from BASIC

You can use disk-based fonts from BASIC, providing that you have the diskfont.bmap file necessary to open the library. Program 4-8 shows how to create such a file on the RAM: disk. It displays print in two sizes of the Ruby font. To end the program, click the left mouse button while the pointer is in the display window.

The selection of disk-based fonts available depends on the files contained in your fonts: directory. The Diskfont library contains a routine that will let you check what fonts are currently available. The format for this statement is

```
Bytes_short = AvailFonts(Buf_ptr, Buf_size, Type);
    (d0)                      (a0)      (d0)      (d1)
```

In order to use this routine, you must first set up a buffer area in which to store the font descriptions. The value Buf_ptr contains the address of the first byte of the pointer, and the value Buf_size contains the size of the buffer in bytes. The Type variable is used to specify whether you want only the fonts already loaded into memory listed (1), only the disk fonts (2), or both (3). The size of the buffer that is needed depends on how many fonts there are on the disk. If you have only the original fonts that came on the Workbench disk, a buffer of 1000 bytes should be plenty. If it turns out that you have not allocated a big enough buffer, however, the value Bytes_short will be returned, telling you how many more bytes you need for your buffer.

If your call to AvailFonts is successful, your buffer will contain a data structure called an AvailFontsHeader, followed by a data structure called an AvailFonts structure for each of the fonts that was found. The AvailFontsHeader is just an unsigned 16-bit value which tells you how many fonts were found. The AvailFonts structures that follow contain an unsigned 16-bit value that tells whether the font was found in memory (1) or on disk (2). This value is followed by the TextAttr structure that describes the font.

One thing to watch for is duplicate listings for a font. A disk font that is opened resides in memory at least until it is closed and possibly longer. Therefore, if you request to see both the fonts that are loaded into memory and the disk fonts, you may find one entry for the font in memory and another for the version that is on disk.

Program 4-9 shows how to use the AvailFonts routine to get the relevant information about all available disk fonts.

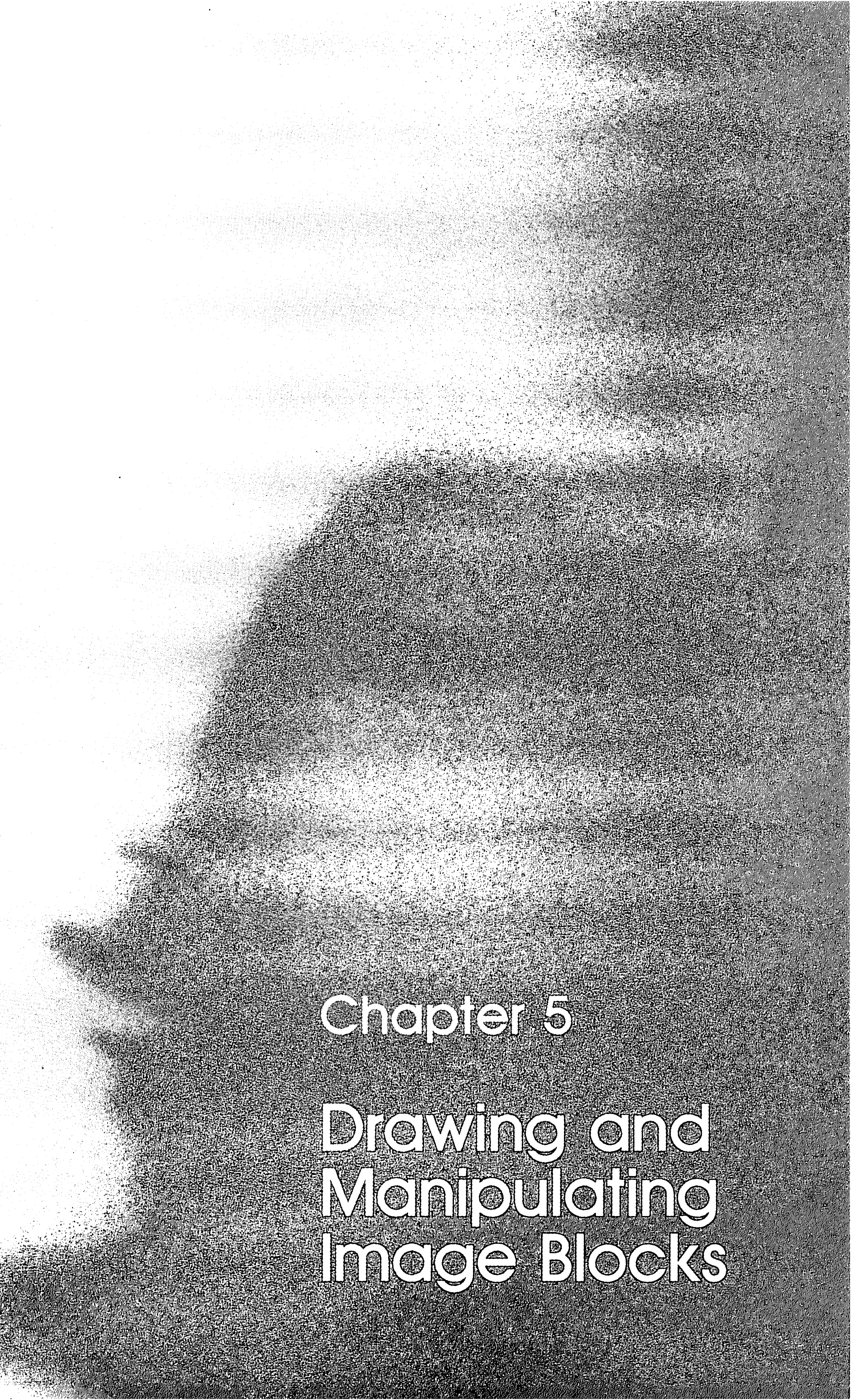
Program 4-8. Creating diskfont.bmap

```
DEFLNG a-z
'all variables default to long integer

DECLARE FUNCTION OpenDiskFont LIBRARY
GOSUB InitLib

WINDOW 2,"Disk Fonts",(100,50)-(525,100),0
WIDTH 41

FOR height = 8 TO 12 STEP 4
  textAttr(0)=SADD("ruby.font"+CHR$(0))
  textAttr(1)=height*65536&
  FontPtr=OpenDiskFont(VARPTR(textAttr(0)))
  IF FontPtr THEN SetFont WINDOW(8),FontPtr
  PRINT
  PRINT " This shows the ruby font";height;
  PRINT "points high"
  'IF FontPtr THEN CloseFont FontPtr
  ' don't close disk fonts when using
  'Kickstart 1.1
NEXT height

WaitForClick:
  IF NOT MOUSE(0) THEN WaitForClick

WINDOW CLOSE 2
WINDOW OUTPUT 1
END

InitLib:
CHDIR "ram:"
D$="OpenDiskFont"+CHR$(0)
D$=D$+CHR$(255)+CHR$(226)+CHR$(9)+CHR$(0)
OPEN "RAM:diskfont.bmap" FOR OUTPUT AS #1
PRINT #1,D$;
CLOSE #1

D$=D$+"SetFont"+CHR$(0)
D$=D$+CHR$(255)+CHR$(190)+CHR$(10)+CHR$(9)+CHR$(0)
OPEN "RAM:graphics.bmap" FOR OUTPUT AS #1
PRINT #1,D$;
CLOSE #1

LIBRARY "diskfont.library"
LIBRARY "graphics.library"
```

```
CHDIR "df0:grafprogs"
RETURN
```

Program 4-9. Using AvailFonts from BASIC

```
DEFLNG a-z
'all variables default to long integer

GOSUB InitLib

NumFonts= 0
Type=0
NamePtr=0
Height=0
n=0:a=0:x=0

DIM Buf%(200)
CALL AvailFonts (VARPTR(Buf%(0)), 400, 2)

NumFonts = Buf%(0)

FOR n = 0 TO NumFonts-1

  Type = Buf%(1+5*n)
  NamePtr = 65536&*Buf%(2+5*n)+Buf%(3+5*n)
  Height = Buf%(4+5*n)
' Style = Buf%(5+5*n)\256
' Flags = Buf%(5+5*n) AND 255

  a=1:x=0:D$=""
  WHILE(a)
    a=PEEK(NamePtr+x)
    IF a>0 THEN D$=D$+CHR$(a)
    x=x+1
  WEND

  PRINT D$;Height
NEXT n

END

InitLib:
CHDIR "ram:"
D$="AvailFonts"+CHR$(0)
D$=D$+CHR$(255)+CHR$(220)+CHR$(9)
D$=D$+CHR$(1)+CHR$(2)+CHR$(0)
OPEN "RAM:diskfont.bmap" FOR OUTPUT AS #1
PRINT #1,D$;
CLOSE #1

LIBRARY "diskfont.library"
CHDIR "df0:grafprogs"
RETURN
```



Chapter 5

Drawing and Manipulating Image Blocks

Drawing and Manipulating Image Blocks

So far, we've looked at the process of drawing images point by point. In this chapter, we'll explore methods of transferring an entire image—a whole block of data bytes—to the screen at once. Most such methods on the Amiga involve the use of the blitter, the powerful data-moving hardware chip.

Not only does the blitter move data at blinding speed, it also can combine and manipulate data from several different sources at once. This allows it to pick out just the bits that form the image and move them, leaving the background data behind.

The one disadvantage to using the blitter is that it, like the other special hardware chips in the Amiga, can access only the first 512K of memory. This may not seem a significant problem to those who have not expanded their systems with external memory. However, as expansion becomes more common, the use of chip memory, as it is called, will become a more important consideration. You should keep in mind that image data may have to be stored in chip memory in order for the routines described below to function correctly.

Therefore, to insure that your program will work correctly with expanded systems, you must take steps to make sure that this data will be loaded into the lower 512K. There are at least two methods of doing so.

The first method is to have your program allocate chip memory using the AllocMem routine, and then copy your data to that memory. This works, but is inefficient because you end up having two copies of the data in memory at once. If you are using a compiler that works with the *Alink* linker, you probably have access to the *Atom* utility program. This program allows you to specify the kind of memory into which certain segments of object code must be loaded. Compiler's other than *Amiga C* (such as the Manx *Aztec C Compiler)* usually include an option which allows initialized data to be loaded into chip memory. Using the *Atom* utility, or the equivalent provided by your compiler and linker, is the preferred method of making sure that your data ends up in chip memory.

Filling Memory

One of the simplest tasks that the blitter can perform is to fill a block of memory with a given number. This can be useful, for example, in clearing the screen or in setting an entire rastport to a certain color. The SetRast Graphics library routine allows you to do the latter. The format of this statement is

```
SetRast (RastPort, Pen);
           (a1)    (d0)
```

where Pen is the number of the pen (color register) whose color you wish to use to fill the rastport.

The BASIC equivalent of SetRast is the CLS statement that clears the screen. It will set the current window to whatever color is in the background pen, which is set by the second value in the COLOR statement.

The blitter can also be used for tasks like clearing memory that is not used for the display. The BltClear statement will fill a contiguous block of memory with zeros. The syntax of this statement is

```
BltClear (Memory, Bytes, Flags);
           (a1)    (d0)   (d1)
```

where Memory is a pointer to the block of contiguous memory, and Bytes specifies the number of bytes to clear. The Flags value determines how bytes are counted. If bit 1 of this value is clear (Flags AND 1=0), Bytes is used to specify the number of bytes to clear. The number of bytes must be even. If bit 1 of Flags is set, however, the lower 16 bits of Bytes specify the number of bytes per row, and the upper 16 bits specify the number of rows to clear. In this mode, the number of bytes per row must be fewer than 129, and the number of rows must be fewer than 1025. By setting bit 0 of Flags, you may force the BltClear function to wait until the blitter is finished clearing the memory before returning back to the program that called it.

Scrolling

Another of the tasks that the blitter can perform is to *scroll* a rectangular area of a window horizontally, vertically, or both at once. The Graphics library routine that is used for this purpose is called ScrollRaster, and it takes the following form:

```
ScrollRaster (RastPort, Dx, Dy, X1, Y1, X2, Y2);
                (a1)    (d0) (d1) (d2) (d3) (d4) (d5)
```

where Dx and Dy are the horizontal and vertical offsets, X1 and Y1 describe the top left corner of the rectangle, and X2 and Y2 specify the bottom right corner of the rectangle.

When you scroll a rectangle, only the data inside the rectangle moves. You can think of the process as chopping off one edge of the rectangle and moving the rest over to fill the part that was chopped off. The vacant area that was created by moving the rectangle is then filled with the color of the background pen. The Dx and Dy offsets specify how far a distance (in pixels) the rectangle is to be moved, and, consequently, how many pixels will be lost. If you wish to move it to the right, Dx should be negative; if you wish to move it to the left, Dx should be positive. Similarly, if you wish to move the rectangle down, use a positive Dy value, and if you want to move it up, use a negative Dy. For instance, if you specify a Dx of 2 and a Dy of 0, the rectangle will scroll two pixels to the left. The leftmost two pixels of image data will disappear off the edge, and the rightmost two pixels will be filled in background color.

If there is an area of background color surrounding the rectangle that you scroll, the image will appear to move smoothly. Written in C, Program 5-1 scrolls some colored rectangles around the screen.

Scrolling in BASIC

The BASIC statement SCROLL performs the same task as ScrollRaster. The syntax is

```
SCROLL rectangle, x_offset, y_offset
```

The rectangle value specifies the coordinates of the upper left and lower right corners of the area to be scrolled. It is expressed in the form

(left,top)-(right,bottom)

where left and right are the horizontal coordinates, and top and bottom the vertical coordinates. These are always expressed as absolute coordinates and cannot be expressed relative to the last position drawn, as the drawing commands can be.

The x_offset and y_offset values show how far to move the designated area horizontally and vertically. The image data for the edge of the rectangle that moves will be lost. The area from which the rectangle is moved will be filled in background color.

Program 5-2 roughly mimics the main action of the arcade game *Space Invaders* by scrolling a group of shapes from side to side and steadily downward.

Program 5-1. Scrolling in C

```
#include <windowl.c>
#include <graphics/gfxmacros.h>

demo()
{
int Row, Col;

   SetAPen (Rp,4);
   for (Row=1;Row<5;Row++)
        {
        for (Col=Ø;Col<8;Col++)
           RectFill(Rp,Col*3Ø,Row*2Ø,Col*3Ø+2Ø,Row*2Ø+12);
        }

Row=1Ø;
while (Row<1ØØ)
 {
 for(Col=Ø; Col<81; Col++)
        ScrollRaster (Rp,-1,Ø,Col,Row,Col+23Ø,Row+82);
     ScrollRaster (Rp,Ø,-1Ø,8Ø,Row,Col+31Ø,Row+92);
     Row+=1Ø;

     for(Col=Ø; Col<81; Col++)
        ScrollRaster (Rp,1,Ø,8Ø-Col,Row,31Ø-Col,Row+82);
     ScrollRaster (Rp,Ø,-1Ø,Ø,Row,23Ø,Row+92);
     Row+=1Ø;
 }

}
/* end of Scroll.c */
```

Program 5-2. Scrolling in BASIC

```
'Box Invaders
SCREEN 1,320,200,2,1                     'lo-res, 4-color Screen
WINDOW 2,,,0,1                           'full-size window
PALETTE 0,0,0,0                          'black  background
PALETTE 2,1,.2,.2                        'red

FOR Row = 0 TO 3                          'draw 4 rows
  FOR Column = 0 TO 7                     'of 8 boxes
    LINE (Column*30,Row*20)- STEP (20,12),2,bf
NEXT Column, Row

inc=-1: Column=-1

FOR Row= 0 TO 11
  Column = Column-SGN(Column)
  inc = -SGN(inc)
  'move boxes vertically
  SCROLL (Column,Row*10)-(Column+230,Row*10+82),0,10
    FOR Column = 0-80*(inc = -1) TO 80+80*(inc=-1) STEP inc
    'move them horizontally
    SCROLL (Column,Row*10)-(Column+230,Row*10+82),inc,0
NEXT Column,Row

WaitForClick:  IF NOT MOUSE(0) THEN WaitForClick

WINDOW CLOSE 2
SCREEN CLOSE 1
WINDOW OUTPUT 1
END
```

Copying Images

A more sophisticated blitter operation involves copying a rectangle of data from one area of a raster and moving that data to another part of the same raster or to another raster. Not only can the blitter make an exact copy of the original data, it can also combine that data in various ways with the data that already exists in the destination area.

The command used to copy a rectangular area of a raster bears the euphonious appellation ClipBlit. The format for this call is

ClipBlit
(SrcRp,SrcX,SrcY,DestRp,DestX,DestY, Width, Height, Minterm);
(a0) (d0) (d1) (a1) (d2) (d3) (d4) (d5) (d6)

As you can see, there are quite a few values that you must pass to this routine. The SrcRp and DestRp values are pointers to the rastports that contain the source rectangle (the area that you are copying) and the destination rectangle (the area to which you are copying it). SrcX and SrcY describe the coordinates for the upper left corner of the source area, and DestX and DestY specify the upper left corner of the destination area. Width and Height specify the size of the rectangle.

The final value, Minterm, is an eight-bit number that describes the kind of logic operation performed on the data in the source rectangle and the destination rectangle to achieve the final output rectangle. Only the high four bits are significant. The meaning of each bit is as follows:

Hexadecimal Value of Minterm	Logic Term Included in Destination Output
10	$\overline{B}\overline{C}$
20	$\overline{B}C$
40	$B\overline{C}$
80	BC

In these logic terms, *B* stands for data in the source rectangle, and C stands for data in the destination rectangle. The *B* with a line over it stands for NOT B, and the C with a line over it stands for NOT C. By combining the logic terms represented by the component values of Minterm, we can solve the resulting equations to find out what data will be included in the output. For example, if we use the Minterm 192 (0xC0), we get the logic equation

$$D = BC + B\overline{C}$$

where D stands for the destination rectangle. If we group this equation differently, we get

$$D = B(C + \overline{C})$$

Since C and NOT C cancel each other out, we are left with

$$D = B$$

where the destination rectangle is an exact copy of the source rectangle, both foreground and background. Here are a few more examples:

For Minterm 0x30: $D = \overline{B}C + \overline{BC} = \overline{B}\,(C + \overline{C}) = \overline{B}$

Destination is an inverted copy of the source rectangle in which every zero bit is changed to a one, and vice versa.

For Minterm 0x50: $D = B\overline{C} + \overline{BC} = \overline{C}\,(B + \overline{B}) = \overline{C}$

Destination is an inverted copy of the destination rectangle in which every zero bit is changed to a one, and vice versa.

For Minterm 0x60: $D = B\overline{C} + \overline{B}C$

Destination is a combination of both source rectangle data and destination rectangle data. Where either source or destination has a zero bit, the value of the other is displayed. Where both have a one bit, a zero is displayed.

For Minterm 0x80: $D = BC$

Destination combines source and destination rectangle data. Only the areas where both source and destination contain a one bit will be displayed in other than background color.

In all, there are 15 different logic term combinations. Even if solving logic equations is not one of your favorite pastimes, with a little experimentation, you should be able to figure out what the other combinations do.

Program 5-3 is a C program that demonstrates the use of ClipBlit. It sets up a smaller window on top of the full-size one and draws an octagon in it. It then copies that octagon to the larger window, using three different Minterm values to come up with three different destination rectangles.

Program 5-3. Using ClipBlit

```
#include <window.c>

demo()
{

struct Window *Wdw2;
#define Rp2 Wdw2->RPort
static UWORD Points [] =
   {
    18Ø,5Ø,
    21Ø,8Ø,
    21Ø,12Ø,
    18Ø,15Ø,
    1ØØ,15Ø,
    7Ø,12Ø,
    7Ø,8Ø,
    1ØØ,5Ø
    };

   NewWdw.Flags = SMART_REFRESH;
   NewWdw.LeftEdge = 32Ø;
   NewWdw.Width=319;


   if (( Wdw2 = (struct Window *)OpenWindow(&NewWdw))
      == NULL) exit(FALSE);

   SetAPen (Rp2,1);
   Move(Rp2,1ØØ,5Ø);
   PolyDraw(Rp2,8,&Points);

/*    SetAPen (Rp,2);
   RectFill(Rp,3Ø,3Ø,23Ø,75); */

   ClipBlit(Rp2,69,5Ø,Rp,Ø,5Ø,142,1Ø1,ØxCØ);
   ClipBlit(Rp2,69,5Ø,Rp,143,5Ø,142,1Ø1,Øx3Ø);
   ClipBlit(Rp2,69,5Ø,Rp,75,98,142,1Ø1,Øx6Ø);

Wait(1<<Wdw->UserPort->mp_SigBit);
/* wait till close box clicked */

    CloseWindow(Wdw);
    CloseWindow(Wdw2);
    CloseLibrary(GfxBase);
    CloseLibrary(IntuitionBase);
    exit(TRUE);

}

/* end of ClipBlit.c */
```

Drawing an Image from a Pattern

The Graphics library contains a routine that lets you fill selected points in an area, rather than the entire area. The points affected by this fill operation are specified by an image pattern that you set up in a data area. The routine is called BltPattern, and its syntax is

BltPattern (RastPort, Pattern, X1, Y1, X2, Y2, Width);
(a1) (a0) (d0) (d1) (d2) (d3) (d4)

where X1 and Y1 specify the upper left corner of the RastPort destination rectangle, and X2 and Y2 are the coordinates of the lower right corner. The Pattern value points to the beginning of the image mask data, and Width describes how the pattern is laid out by telling how many bytes there are per row. This number must be an even value. The total number of bytes of pattern data can be found by multiplying the number of bytes per row times the number of rows (Y2 − Y1).

You convert an image for BltPattern to pattern data in exactly the same way as you do in determining AreaPattern data. You draw the image as a series of filled boxes and empty boxes, and then convert that image to a binary number by substituting a one for each filled box and a zero for each empty one. For example, the data needed to draw a cross would look like this:

```
0000001111000000 = 0x3C0
0000001111000000 = 0x3C0
0000001111000000 = 0x3C0
0000001111000000 = 0x3C0
1111111111111111 = 0xFFFF
1111111111111111 = 0xFFFF
1111111111111111 = 0xFFFF
0000001111000000 = 0x3C0
0000001111000000 = 0x3C0
0000001111000000 = 0x3C0
0000001111000000 = 0x3C0
```

When you use the BltPattern statement, only the area that corresponds to the places where there are one bits in the pattern will be affected. Those points will be filled just as if they were part of an AreaFill, so just what ends up inside the image pattern depends on the foreground and background pen values, the drawing mode, and the area pattern.

Program 5-4 is a C program that demonstrates the use of BltPattern. It draws the image of a box with several holes cut out, filled with the same patterns as used in the pattern fill example in Chapter 3.

Program 5-4. Using BltPattern

```
#include <windowl.c>
#include <graphics/gfxmacros.h>
demo()
{

static UWORD Pat1 [] =    /* 'HI' fill pattern */
   {
   0x0000,
   0x667E,
   0x6618,
   0x7E18,
   0x6618,
   0x6618,
   0x667E,
   0x0000
   };

static UWORD Pat2 [] =   /* geometric pattern */
   {
   0x0FF0,
   0xF00F,
   0xAAAA,
   0x5555,
   0xA5A5,
   0x5A5A,
   0xF0F0,
   0x0F0F
   };

static UWORD Pat3 [] =   /* random fill pattern */
   {
   0x048C,
   0x159D,
   0x26AE,
   0x37BF,
   0x3333,
   0xAAAA,
   0x3C3C,
   0xD43D4
   };

static UWORD Image [] =
   {
   0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
   0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
   0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,

   0xFFFF,0x0000,0xFFFF,0x0000,0xFFFF,
   0xFFFF,0x0000,0xFFFF,0x0000,0xFFFF,
   0xFFFF,0x0000,0xFFFF,0x0000,0xFFFF,
        0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
        0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
        0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,

        0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
        0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
        0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,

        0xFFFF,0x0000,0xFFFF,0x0000,0xFFFF,
        0xFFFF,0x0000,0xFFFF,0x0000,0xFFFF,
        0xFFFF,0x0000,0xFFFF,0x0000,0xFFFF,

        0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
        0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF,
        0xFFFF,0xFFFF,0xFFFF,0xFFFF,0xFFFF
        };


        SetAfPt(Rp,&Pat2[0],3);
        SetAPen(Rp,2);
        SetBPen(Rp,5);

        BltPattern (Rp,&Image,50,50,130,66,10);

        SetAfPt(Rp,&Pat3[0],3);
        SetAPen(Rp,3);
        SetBPen(Rp,4);

        BltPattern (Rp,&Image,130,100,210,116,10);

        SetAfPt(Rp,&Pat1[0],3);
        SetAPen(Rp,6);
        SetBPen(Rp,7);

        BltPattern (Rp,&Image,210,150,290,166,10);

    }

    /* end of Bltpat.c */
```

Moving Images to and from BASIC Arrays (GET and PUT)

The BASIC equivalent of the Graphics library routines that use the blitter to move an entire image are the statements PUT and GET. The GET statement allows you to capture the image in a rectangular area of the screen in a data array. The PUT statement allows you instantly to redisplay that image elsewhere in the window (or in another window entirely).

Normally, before you can PUT an image to the screen, you must draw it in a window, using the normal drawing commands, and store it in an array by using the GET command. The syntax is

GET (x1,y1)-(x2,y2), array [(sub1[,sub2...])]

The two values that must be specified are the rectangle whose image is to be stored and the name of the array in which it will be saved. The area of the rectangle is specified by the coordinate pairs *(x1, y1)* and *(x2, y2).* The first pair represents the absolute position of the top left corner of the rectangular area, and the second specifies the bottom right corner.

Before we can use an array to store an image, the size of the array must be declared with the DIM statement. Its size must be large enough to hold all of the display data. To determine the size to which the array must be DIMensioned, let's first take a look at the format in which the image is stored. If a 16-bit (short) integer array is used, the first three words store the width, height, and depth of the array. Let's take the case of an image that is 40 dots wide, 20 lines high, and three bit planes deep:

```
a%(0) = 40
a%(1) = 20
a%(2) = 3
```

Since the image data is stored in 16-bit words, the width of the image is rounded up to the next highest multiple of 16 to find the least number of words required to store one line of the image. In this example, each line requires 3 words of data (48 bits) to hold the 40 dots. Since there are 20 lines per bit plane, each bit plane requires 60 words (3 words wide * 20 high) to hold the data. The correspondence of the bit patterns of the data words and the dots that make up the display is the same as that described in the section on pattern drawing above and in the section on patterned fills in Chapter 3. The data for plane 0 is assigned to array elements as follows:

```
a%(3)=line0left     a%(4)=line0middle    a%(5)=line0right
a%(6)=line1left     a%(7)=line1middle    a%(8)=line1right
      ...                 ...                  ...
      ...                 ...                  ...
      ...                 ...                  ...
a%(60)=line19left   a%(61)=line19middle  a%(62)=line19right
```

The same kind of assignment is made for each of the three bit planes. Since there are three bit planes, a total of 183 words are required (60 words / bit plane * 3 bit planes + 3 format words).

For purposes of your programs, if you use short integer arrays, you may use the formula

```
arraysize = 3 + INT((16 + x2 - x1) /16) * (1 + y2 - y1) * depth
```

To find the size to which you must DIMension the array, use

```
DIM a%(arraysize)
```

The GET statement allows you to specify subscripts for the array. This allows you to create multidimensional arrays, with a picture stored in each subscript. For instance, if you want to store five images that each require an integer array of 500 words, you may dimension one array for all five images using the statement form

```
DIM a%(500,5)
```

When you fill the array, use this form:

```
GET (x1,y1)-(x2,y2),a%(0,0) 'first image
GET (x3,y3)-(x4,y4),a%(0,1) 'second image
GET (x5,y5)-(x6,y6),a%(0,2) 'third image
```

Note that the first subscript always stays at zero, while the second keeps track of the image number.

To redisplay the stored image, you use the PUT statement. The form of this statement is

```
PUT [STEP] (x,y), array [(sub1[,sub2...])] [,combination_type]
```

Here you need only specify the coordinates of the top left corner of the image and the name of the array in which it is stored. The coordinates may be specified either as absolute points or as an offset relative to the last point drawn. As with GET, multiple array dimensions may be specified.

The kind of image drawn when you use the PUT statement depends on the value you choose for combination_type. As with the ClipBlit operation described above, PUT can be used not only to display the exact duplicate of the saved rectangle, but also to combine that image with that of the destination rectangle in various ways. Five types of combinations may be made between the image values stored in the array and the values that are currently displayed onscreen. Their names are PSET, PRESET, AND, OR, and XOR. The concepts behind these combinations should not be so strange; PSET and PRESET are graphics commands which we have discussed, and the others are logical operators.

If the PSET combination_type is selected, the entire rectangular area of the image will appear, exactly as it was saved. This includes the background color as well as any foreground colors used. If the PRESET type is chosen, the entire area of the image appears with each color, including the background color, complemented. This means, for example, that if the screen is two planes deep, parts of the image that were stored as color 0 would appear in color 3, parts that were stored as color 1 would appear in color 2, and vice versa. For more information about complementing, see the section "Drawing Modes" in Chapter 3.

The three remaining combination types use the logical operators AND, OR, and XOR (exclusive OR) to combine the pen values of the stored image with those displayed onscreen. In the AND mode, the bits of the image are logically ANDed with those of the display. The following chart shows all of the possible combinations of one pen color ANDed with another in a four-color display:

First Pen	Second Pen	Resulting Display Pen
0	0	0
0	1	0
0	2	0
0	3	0
1	1	1
1	2	0
1	3	1
2	2	2
2	3	0
3	3	3

The OR combination mode logically ORs the bits of the image with those of the display. The following chart shows all of the possible combinations of one pen color ORed with another in a four-color display:

First Pen	Second Pen	Resulting Display Pen
0	0	0
0	1	1
0	2	2
0	3	3
1	1	1
1	2	3
1	3	3
2	2	2
2	3	3
3	3	3

The XOR combination mode is the default mode used if no combination_type is specified. One reason for this mode being the default is that when it is used, the entire image always appears onscreen (though its color may vary), and the part of the stored image that was drawn with the background pen never appears. Also, because it undoes its own effects, the XOR operator is useful for a limited form of animation. Using XOR mode, if you PUT an image once, the image appears, but if you PUT the same image a second time in the same place using XOR, the display is restored to its original state before the PUT took place. See if you can figure out why from the following chart, which shows all of the possible combinations of one pen color XORed with another in a four-color display.

First Pen	Second Pen	Resulting Display Pen
0	0	0
0	1	1
0	2	2
0	3	3
1	1	0
1	2	3
1	3	2
2	2	0
2	3	1
3	3	0

Program 5-5 graphically demonstrates the various color combinations resulting from the use of the different combination types. An interesting feature of this program is that the array used by the PUT statement is not created by a corresponding GET statement. Rather than drawing the image, we use the same technique demonstrated above to create the BltPattern and the area fill pattern. Rows of binary data are laid out one on top of the other to form a picture. This data is then read into the appropriate elements of the image array.

Program 5-6 demonstrates the more usual method of drawing a picture and storing it into the array to be used later by the PUT statement. To keep the user from seeing the picture when we first GET it, the PALETTE statement is used to change the foreground pen color to the same shade as the background pen color, rendering the drawing invisible. Also, note that a single two-dimensional array is used to store the image of all six dice. We can display the proper dice face by just changing the second array subscript.

Program 5-5. Graphics Demo Using PUT

```
DEFINT a-z
WINDOW 1,,(0,0)-(500,180)

DIM man(40)
man(0)=16  'image is 16 bits wide
man(1)=18  'by 18 lines high
man(2)=2   'and 2 bit planes deep

FOR x=3 TO 20
  READ d       'read image data
  man(x)=d     'into the PUT array
NEXT

FOR row= 1 TO 3   '3 rows
  FOR col=0 TO 4  'of 5 columns each
    LINE (48+100*col,9+50*row)-STEP (20,10),row,bf
  NEXT col


  PUT (50,50*row), man,PSET
  PUT (150,50*row), man,PRESET
  PUT (250,50*row), man,AND
  PUT (350,50*row), man,OR
  PUT (450,50*row), man,XOR
NEXT row
```

```
WIDTH 60
LOCATE 5,1    'print heading
PRINT PTAB(40) "PSET" PTAB(130) "PRESET";
PRINT PTAB(240) "AND" PTAB(350) "OR";
PRINT PTAB(440) "XOR"
LOCATE 1,1

REM--18 words of image data
DATA &H07E0, &H0FF0, &H1998, &H1FF8
DATA &H1C38, &H0FF0, &H03C0, &H0FF0
DATA &HFFFF, &HFFFF, &H0FF0, &H0FF0
DATA &H1FF8, &H1FF8, &H1E78, &H1C38
DATA &H7C3E, &H7C3E
```

Intuition Images

For C programmers, the Intuition library provides an excellent general-purpose image-display mechanism. Although Intuition uses this system specifically for the purpose of rendering the graphics images associated with such Intuition features as gadgets and menus, it can be used to create ordinary graphics objects for whatever purpose you might have.

At the heart of this image-production system is a data structure known as the Image structure. This structure provides all the information needed to draw the Image. The C language definition for the Image data structure looks like this:

```
struct Image
{
    SHORT LeftEdge, TopEdge;
    SHORT Width, Height, Depth;
    SHORT *ImageData;
    UBYTE PlanePick, PlaneOnOff;
    struct Image *NextImage;
};
```

As you can see, it's necessary to supply a number of values in order to define the image. The first two, LeftEdge and TopEdge, specify the coordinates of the top left corner of the image. When used to draw Intuition objects like gadgets, these values specify the exact position of the object. However, when used with the DrawImage routine, which we will examine below, these values may be modified by offset values, which effectively allows you to place them anywhere in the window.

Program 5-6. Using PUT and GET to Draw a Picture

```
DEFINT A-Z
SCREEN 1,320,200,3,1    'lo-res, 8 colors
WINDOW 2,,,8,1          'full-screen window

PALETTE 0,0,0,0  'black background
PALETTE 2,1,0,0   'red foreground
PALETTE 4,0,1,0   'green foreground
PALETTE 5,1,1,1   'white dice spots
PALETTE 3,1,1,1   'white dice spots

GOSUB InitDice 'set up dice image arrays
WIDTH 37: LOCATE 19,6
PRINT "Strike any key to roll again"
RANDOMIZE TIMER 'initialize RND function
y1=80  'top line of dice

Rolldice:
FOR Change=0 TO 5:FOR Die=1 TO 2     'use two dice, roll each 6 times
  x1=64*Die+40                       'set to left or right die
  Roll(Die)=INT(RND*6)               'pick a random roll
  LINE(x1,y1)-STEP (47,39),Die*2,BF  'blank die
  PUT (x1,y1),Spots(0,Roll(Die))     'draw spots
  SOUND 10000,.001:SOUND 150,0       'make click
NEXT Die, Change

WIDTH 40:LOCATE 9,1
PRINT PTAB(117) Roll(1)+1; PTAB(180)Roll(2)+1 'print numbers above dice

CheckForRoll:
'If user closed the window, end
IF WINDOW(8)=0 THEN SCREEN CLOSE 1:WINDOW OUTPUT 1:END
IF INKEY$="" THEN CheckForRoll ELSE Rolldice
```

```
InitDice:
REM This subroutine draws the spots of the dice
REM and then GETs the image data into array SPOTS

DIM Spots(5ØØ,7), Roll(2)
PALETTE 1,Ø,Ø,Ø 'make foreground=background, so spots are invisible

FOR Pair=Ø TO 4 STEP 2                   'for 3 pairs of dice shapes
  FOR Spot=Ø TO 3
    READ x,y
    CIRCLE(x,y),5
    PAINT(x,y)
  NEXT Spot                              'draw two dice
  GET (1Ø4,8Ø)-(159,119),Spots(Ø,Pair)
  GET(168,8Ø)-(223,119),Spots(Ø,Pair+1)  'read their data
NEXT Pair

PALETTE 1,1,1,1
CLS                        ' clear screen and make spots white again
RETURN

DATA 178,86,2Ø6,112,127,99,127,99
DATA 2Ø6,86,178,112,112,86,142,112
DATA 178,99,2Ø6,99,142,86,112,112
```

The Width value is the width of the image in pixels. An image can be any width, provided that you furnish enough data to define its shape. As we'll see below, each line of image data is composed of a number of 16-bit words sufficient to contain the image. For example, if the image is 12 pixels wide, you must use only one 16-bit word of image data per line for each bit plane used. If the image is 40 pixels wide, you must use three words per line, because 48 (3 * 16 bits) is the lowest multiple of 16 into which 40 pixels will fit. When your image width is not an even multiple of 16, the image will take the form of the most significant bits. In other words, if your image is 40 pixels wide, the image will take its shape from the first two words in each line (32 bits), plus the leftmost eight bits of the third word. The low-order eight bits of the last word in each line will be ignored.

The Height value gives the height of the image in number of lines. The Depth value specifies the number of bit planes used to define the image, which in turn determines the number of different colors that can be displayed within the image. This value does not have to be the same as the depth of the screen on which the image will be drawn. If it is greater than the depth of that screen, however, not all of the bit planes of data will be displayed.

Together, the Width, Height, and Depth values determine how the image data will be interpreted. The data for each bit plane will consist of Height number of lines, each composed of sufficient 16-bit words to contain Width number of pixels. The bit planes are laid out one after the other so that the data starts with the first word of the first line of the first bit plane, continues with the second word of that line, until the whole line has been defined. The data after that is used to describe the next line of the first bit plane. After the last line of the first bit plane has been defined, the next data word starts the first line of the second bit plane, and so on, until Depth number of planes have been defined.

The next variable is ImageData. This is a pointer to the actual display data that you've defined. We will examine the process of setting up that data in more detail below.

The next two values, PlanePick and PlaneOnOff, allow you to specify the bit planes into which the image data will be drawn. This affords a certain amount of flexibility in the choice of colors that can be used to depict various parts of the image. As a result, two images, using the exact same image data, can be drawn in two completely different sets of colors.

PlanePick and PlaneOnOff can be thought of as masks that can change the order in which your image would normally be drawn into the various bit planes available. Ordinarily, your image would be transferred to the display in sequential order, with the first bit plane of your image going into the first bit plane of the display, the second image plane to the second display plane, and so forth. If you have an image that is two planes deep, however, and a screen that is five planes deep, you may not always want to place your image into the first two bit planes. That's where PlanePick and PlaneOnOff come into play.

PlanePick is used to determine which bit planes of the display receive your image data. Let's say that you have a two-plane image that uses pens 2 and 3, and you want to display it on a three-plane screen. Normally, the two image planes would be displayed in planes 0 and 1. You can set PlanePick, however, to display these as two entirely different planes. You choose which planes will be used by setting PlanePick to the sum of the bit values of the planes in which you wish the object displayed. Each bit value corresponds to 2 raised to the *n*th power, where *n* is the number of the plane.

For example, the bit value of plane 0 is 1 (2^0), the bit value of plane 1 is 2 (2^1), and so forth. The PlanePick value that corresponds to the normal setting of planes 0 and 1 would be 3 (1 + 2). To display the image in planes 1 and 2, you would set the PlanePick value to 6 (2 + 4). The part of the image that was created using pen 1 will now be displayed in the color of pen 2, and the part of the image that was created using pen 2 will now be displayed in the color of pen 4. The part of the image that was originally colored in pen 3 (both planes set) will now be shown in the color of pen 6. The following chart shows each of the possible PlanePick values for a three-plane screen, the binary representation for that value, and the meaning of such a PlanePick setting.

PlanePick Value	Binary Value	Display Planes Used
0	000	No image planes are displayed.
1	001	The first image plane goes into display plane 0; the rest are not displayed.
2	010	The first image plane goes into display plane 1; the rest are not displayed.
3	011	The first image plane goes into display plane 0, the second into plane 1; the rest are not displayed.
4	100	The first image plane goes into display plane 2; the rest are not displayed.
5	101	The first image plane goes into display plane 0, the second into plane 2; the rest are not displayed.
6	110	The first image plane goes into display plane 1, the second into plane 2; the rest are not displayed.
7	111	The first image plane goes into display plane 1, the second into plane 2, the third into plane 3.

The PlaneOnOff value can be used to further enhance the selection of colors. Let's go back to the example above of a two-plane (four-color) image and a three-plane (eight-color) screen. If you wanted to display your object in pen colors 3, 5, and 7 instead of 2, 4, and 6, it would not be possible using PlanePick alone since these colors require that a bit be set in each two-color plane. PlaneOnOff lets you decide whether the color planes that were not chosen by PlaneOnOff will always be set to all zero bits or all one bits. In our example, an image that originally used planes 0 and 1 (pen colors 1, 2, and 3) was changed to use planes 1 and 2 (pen colors 2, 4, and 6). This means that plane 0 is not used at all. PlaneOnOff lets you determine how plane 0 will be set as well. If you chose a PlaneOnOff value of 1, which corresponds to a one bit in plane 0, every bit in plane 0 will be set to 1. This has the effect of adding 1 to the pen values made possible by PlanePick. If PlanePick is set to 6, and PlaneOnOff is set to 1, the parts of the object that were originally drawn in pens 1, 2, and 3 will appear in pen colors 3, 5, and 7 . The background color, which was originally drawn in pen 0, will now be set to pen 1.

The last value in the Image structure is NextImage. If you place the address of another Image structure in this variable, the two Images become linked, and anytime the first is drawn, the second will be drawn also (though the reverse is not true). This image may in turn be linked to another. The last image in the chain should have a null (zero) value in the NextImage variable to show that there are no more images in the chain. Linking images together in this is often much more memory efficient than designing a multipart image as one big image that contains a lot of blank space.

The final step is to create the image display data. This process is similar to creating the pattern for area filling or any of the other image commands that we've covered. You must draw a picture composed of filled and unfilled boxes, and convert those boxes to binary numbers, where ones represent filled boxes and zeros represent unfilled boxes. The pattern can be any number of pixels wide, but the data must be aligned on word boundaries. When it comes time to draw the image, the operating system routine uses the leftmost bits of each line of image data.

It's important to note that in order for an Intuition image to display properly on a machine that uses expansion memory, the image display data (not the Image structure) must be in the lower 512K of memory. You can insure this by using the *Atom* utility program before using *Alink* or by using the equivalent option on your own linker.

When the image data has been created and the Image structure which points to that data has been set up, you may draw the image by calling the DrawImage routine. The syntax for this routine is

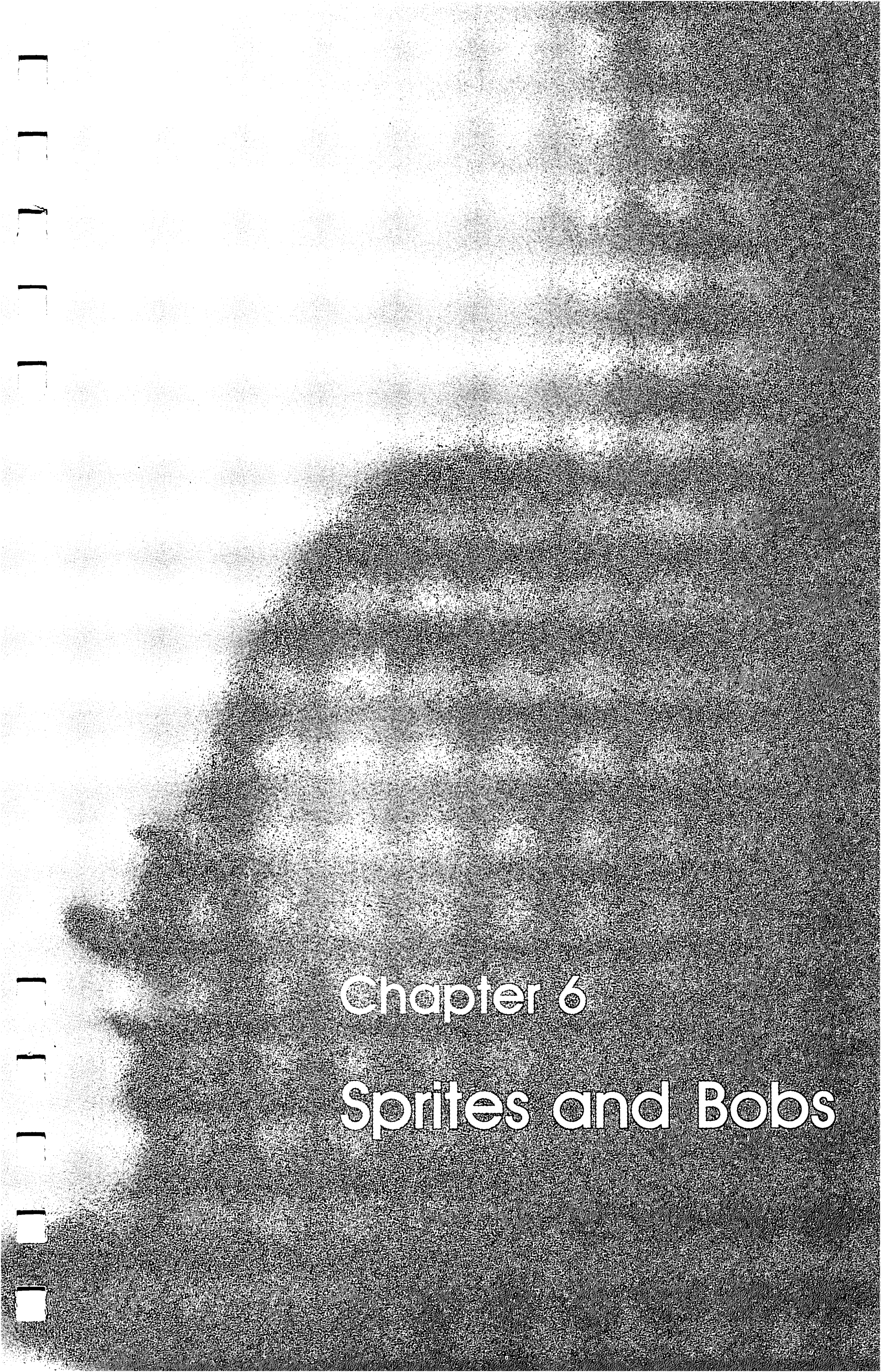
```
DrawImage (RastPort, Image, LeftOffset, TopOffset);
             (a0)      (a1)     (d1)        (d2)
```

where Image is a pointer to the Image data structure, and LeftOffset and TopOffset are position values that are added to the LeftEdge and TopEdge values in the Image structure to arrive at the actual positioning of the image on the screen. This means that even though you specify a position for the image in the Image structure, you are not bound to use that position, but can draw the image anywhere onscreen.

Program 5-7 shows many of the features of the Intuition Image structure. It defines two images, one of a little man and the other of a balloon, each two planes deep. It links these two images by having the Image structure of one point to the other so that both can be drawn with a single call to DrawImage; it also shows how to draw just the last image in the linked list. It uses the PlanePick and PlaneOnOff values to change the colors used to display these two-plane images on a three-plane screen.

Program 5-7. Using the Intuition Image Structure

```
#include <window1.c>
#include <graphics/gfxmacros.h>

UWORD ManData [] =     /* picture of man */
   {
   0x0FC0,     /* Plane 0 */
   0x3FF0,
   0x3330,
   0x3330,
   0x3FF0,
   0x3CF0,
   0x0FC0,
   0x0300,
   0xFFFC,
   0xFFFC,
   0x0FC0,
   0x0FC0,
   0x3FF0,
   0x3CF0,
   0x3CF0,
   0xFCFC,
   0xFCFC,

   0x0000,     /* Plane 1 */
   0x0000,
   0x0CC0,
   0x0CC0,
   0x0000,
   0x0300,
   0x0000,
   0x0000,
   0x0000,
   0x0000,
   0x0000,
   0x0000,
   0x0000,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ
   };

UWORD BaloonData [] =
   {
   ØxØF8Ø,    /* Plane Ø */
   Øx3FEØ,
   Øx7FFØ,
   Øx7FFØ,
   ØxFFF8,
   ØxFFF8,
   ØxFFF8,
   Øx7FFØ,
   Øx7FFØ,
   Øx3FEØ,
   ØxØF8Ø,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,
   ØxØØØØ,

   ØxØF8Ø,    /* Plane Ø */
   Øx3FEØ,
   Øx7FFØ,
   Øx7FFØ,
   ØxFFF8,
   ØxFFF8,
   ØxFFF8,
   Øx7FFØ,
   Øx7FFØ,
   Øx3FEØ,
   ØxØF8Ø,
   ØxØ2ØØ,
   ØxØ2ØØ,
   ØxØ2ØØ,
   ØxØ2ØØ,
   ØxØ2ØØ,
   ØxØ2ØØ,
   ØxØ2ØØ,
   ØxØ2ØØ,
   ØxØ2ØØ,
   ØxØ2ØØ,
   ØxØ2ØØ
   };


struct Image ManImage =
   {
   Ø,12,               /* left, top position */
   14,17,2,           /* width, height, depth */
   &ManData[Ø],       /* pointer to image data */
   3,Ø,               /* PlanePick, PlaneOnOff */
   NULL               /* pointer to next Image */
   };

struct Image BaloonImage =
   {
   8,Ø,                /* left, top position */
   13,22,2,             /* width, height, depth */
   &BaloonData[Ø],      /* pointer to image data */
   3,Ø,                 /* PlanePick, PlaneOnOff */
   &ManImage            /* pointer to next Image */
   };


demo()
{

DrawImage (Rp,&BaloonImage,1ØØ,5Ø);

BaloonImage.PlanePick = 6;
ManImage.PlanePick = 6;
DrawImage (Rp,&BaloonImage,13Ø,5Ø);

DrawImage (Rp,&ManImage,7Ø,5Ø);

BaloonImage.PlaneOnOff = 1;
ManImage.PlaneOnOff = 1;
DrawImage (Rp,&BaloonImage,16Ø,5Ø);

BaloonImage.PlanePick = 5;
ManImage.PlanePick = 5;
BaloonImage.PlaneOnOff = Ø;
ManImage.PlaneOnOff = Ø;
DrawImage (Rp,&BaloonImage,19Ø,5Ø);

BaloonImage.PlaneOnOff = 2;
ManImage.PlaneOnOff = 2;
DrawImage (Rp,&BaloonImage,22Ø,5Ø);

}

/* end of Image.c */
```



Chapter 6

Sprites and Bobs

Sprites and Bobs

Using the block-image transfer methods discussed in Chapter 5 can be very useful for stamping images in various locations on the display screen and for a limited form of animation. But for full animation effects, something a little more is needed.

There are basically two ways to achieve animation of a graphics object on a microcomputer. The first is by quickly reshuffling the contents of the display bitmap. The second method makes use of special display hardware to create distinct animation objects. The Amiga supports both of these methods of animating images. For bitmap animation, it offers animation objects called *bobs* (short for *b*litter *ob*jects). In addition, the special display hardware of the Amiga supports animation objects known as sprites.

Sprites

Sprites are graphics objects that are created by special hardware, and are displayed and moved around the screen entirely independently of the normal bitmap display (which is sometimes also called the playfield display). Since the display data for sprites is stored in a different area than the bit planes used by the normal display, sprites do not interfere with or change bitmap data as they move about. And since the sprite display data includes information about where to position the sprite on the screen, moving a sprite is as easy as changing its display data.

Sprite graphics affords the programmer tremendous power for animating objects. But sprites have their limitations. First of all, the hardware supports only eight hardware sprites on any given horizontal line. The impact of this limitation is lessened somewhat by the system software, which lets you move sprites around while the screen is being drawn. While a sprite can appear only once on a particular horizontal line, it may be redrawn on a different horizontal line farther down the screen. The software mechanism for reusing sprites so that one sprite may appear in different shapes, sizes, colors, and locations on the same screen is known as *vsprites* (for *v*irtual *sprites).* We'll discuss vsprites in detail a little later on.

Another limitation of sprites is their size. Each sprite can be a maximum of 16 bits (dots) wide. Though this may seem somewhat narrow, you should note that sprites are always displayed in low-resolution mode, regardless of the resolution of the normal bitmap graphics screen. This means that if the playfield (bitmap) screen is high resolution, each dot of the sprite will be twice as wide as a dot of background graphics. Though the width of a sprite is limited to 16 bits, its height is determined strictly by the number of lines of shape data that you provide for it. Each sprite can be as tall as the screen.

Normally, each dot of a sprite can be colored in any one of three colors, or it can be transparent. In effect, each sprite is two bit planes deep. Instead of color 0 being a distinct background pen, however, the parts of the sprite that normally would be colored by pen 0 take on the color of the bitmap underneath. The actual colors that the sprite displays are determined by the upper 16 color registers. Each pair of sprites shares a set of color registers. The register assignments for each sprite are shown in the table below:

Sprite Numbers	Color Registers (pen numbers)
0 and 1	17, 18, and 19
2 and 3	21, 22, and 23
4 and 5	25, 26, and 27
6 and 7	29, 30, and 31

Note that the sprites share these color registers with the normal bitmap graphics. This means that if you're using five bit planes for the playfield, you will not be able to select 32 unique colors for the playfield and 12 more unique colors for the sprites. You'll be confined to a total of 32 colors. On the other hand, not every program requires that you use five bit planes at once, and programs that require a high-resolution screen don't even allow you to use five bit planes.

The use of sprites may actually allow you to use fewer bit planes for your program display. If most of your program display uses one or two colors, but some areas require a few additional colors, you may use sprites to provide those colors, rather than allocating additional bit planes unnecessarily. This practice can free up a lot of memory for use by other programs in the Amiga's multitasking environment.

Although normally each sprite is limited to three colors, there is a special mode in which you may attach two sprites together to form one colorful sprite. Although this sprite is only 16 bits wide, each dot may be one of 15 colors or transparent. You may attach only sprites that share the same color registers (for example, sprites 0 and 1, 2 and 3, and so forth). When you attach them, the two bit planes of each sprite are combined to form four bit planes. These four bit planes display the colors from registers 17 through 31. A one value in the bit planes displays pen 17, a two displays pen 18, and so on.

A final word of caution before we start exploring how to manipulate sprites. Sprites don't fit into the framework of the Intuition user interface very well. In order to manage windows on the display effectively, Intuition must have complete control over it. Sprites, however, are not affected by manipulation of the normal bitmap graphics system. As a result, Intuition has no way of keeping sprites in a particular window. If you resize the window, move it with the drag bar, or use the depth arrangers to send it to the back, the sprite will just stay put. In fact, sprites won't even move with the screen if you drag it up and down, unless they have to in order to keep from moving onto another screen. This can be inconvenient at best. For serious use in your own programs, therefore, you'll usually get the best results by using sprites on a custom screen and in windows that can't be moved, sized, or depth arranged.

Simple Sprites

The operating system provides two methods for using sprites on the Amiga. The first is less flexible, but very easy to set up. This system is known as simple sprites. The simple sprite system follows the hardware model for sprites very closely. It allows for only eight sprites, used in only one place per display.

Since the Intuition pointer is actually a sprite, there are only seven sprites available for your use.

Setting up simple sprites requires you to provide little more than the data actually used by the sprite hardware. The first step is to set up the display data that defines the shape of the sprite. This data uses a slightly different format from that of Intuition images or the shape information for patterned fills.

Image data for those constructs are set up so that each bit plane is defined separately. First comes the shape information for the entire first bit plane, then the shape information for all of the next bit plane. With simple sprites, however, the data for both bit planes is intermixed. Each line of sprite shape definition data consists of one 16-bit word of data for plane 1, followed by another 16-bit word of data for plane 0 of that line. Another difference is that the sprite shape data contains two additional 16-bit words having a value of zero, one at the beginning and the other at the end of the data table. These added words hold a place for position data created by the simple sprite machine.

To illustrate the above information, let's create the data needed for a sprite whose shape is a striped rectangle. We'll make the top stripe color 3, the middle stripe color 2, and the bottom stripe color 1. We will separate each of the colored stripes with a transparent stripe (color 0). The data for such a sprite looks like this:

```
UWORD Sprite_data  [] =
  {
 0x0000, 0x0000,
  /* Holds place for position and control data */

 /* Stripe of color 3--both planes have all 1's */
0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF,
0xFFFF, 0xFFFF,

 /* Stripe of color 0--both planes have all 0's */
0x0000, 0x0000,

 /* Stripe of color 2--plane 1 has 1's, plane 0 has 0's */
0xFFF0, 0x0000,
0xFFF0, 0x0000,
0xFFF0, 0x0000,

0x0000, 0x0000,
 /* Stripe of color 1--plane 1 has 0's, plane 0 has 1's */
0x0000, 0xFFFF,
0x0000, 0xFFFF,
0x0000, 0xFFFF,

 /* This line can be used to indicate attached sprites */
0x0000, 0x0000
  };
```

Remember our caution in Chapter 5: Sprite shape data is used directly by the display hardware chips. Since these chips have access only to memory located in the bottom 512K of the computer's address space, you must take steps to insure that your sprite data is loaded into this area if you want your program to function correctly on machines that are equipped with external expansion memory. To do this, you must either use the *Atom* utility on your Amiga C object code file before linking or whatever equivalent provision is made by compiler and linker.

The next step in creating a simple sprite is to set up a SimpleSprite data structure. The C language declaration for such a structure is as follows:

```
struct SimpleSprite
   {
   UWORD *posctldata;
   UWORD height;
   UWORD X,Y;
   UWORD num;
   };
```

Most of these values are self-explanatory. The first, posctldata, is a pointer to the address of the position and control data that the sprite machine will use to position the sprite. Since we have already said that space for that data is reserved at the beginning of the shape data table, this value should contain the starting address of the sprite shape data.

The height variable tells how many lines of actual shape data are contained in the table. Remember that the table contains an extra line at the beginning and the end, and count only the lines that contain actual shape data. For example, the data table that we created above contains 13 lines of data (each line consisting of two 16-bit words), but only 11 of these lines contain sprite shape data, so the correct value for height is 11.

The *x* and *y* values correspond to the horizontal and vertical position at which the sprite is initially displayed.

Finally, the num value indicates the number of the hardware sprite used to display this simple sprite. You do not need to specify this number since it is automatically supplied by the sprite machine when you allocate a sprite for your use.

An initialized SimpleSprite data structure looks like this:

```
struct SimpleSprite StripeSprite =
  {&Sprite_data[0], 11, 100, 50, 0 };
```

It is not absolutely necessary to preinitialize every value in the SimpleSprite structure. The posctldata value may be supplied by using the ChangeSprite statement. The *x* and *y* position values may be supplied by the MoveSprite statement. And the sprite number will be supplied by the GetSprite statement. All of these statements will be discussed in greater detail below. What's important to remember is that the one value that you must supply is the height of the sprite. If this value is left at zero, nothing will be displayed.

After you've set up the shape data and the SimpleSprite data structure, the next step is to allocate a sprite for your use from the sprite machine. You can request a particular sprite or ask to be given the first available sprite. The statement used to reserve a sprite for your exclusive use is GetSprite. The proper form for this statement is

```
Sprite_got = GetSprite (SimpleSprite, Sprite_number);
  (d0)                       (a0)           (d0)
```

The GetSprite statement requires that you pass it two values. The SimpleSprite value is the address of the SimpleSprite data structure that you've set up for this sprite. The Sprite_number value is the number of the hardware sprite that you are requesting (from 0 to 7). If you don't care which sprite you get, you can supply a value of −1, and the system will allocate the first available sprite for your use. The GetSprite routine returns the value Sprite_got, which tells you which sprite (0–7) was allocated for your use.

If the GetSprite call succeeds, the number of the sprite that you have reserved will also be placed in the num variable of the SimpleSprite structure. When the system is unable to allocate the particular sprite that you requested, or when no sprites are available for allocation, the value returned in Sprite_got is −1. Your program should test the value that GetSprite returns to make sure that a sprite is allocated before going any further. As was mentioned above, sprite 0 is used by Intuition's mouse pointer, so don't try to reserve it for your own use. And since sprite 1 shares color registers with sprite 0, if you use that sprite, it will always be the same colors as the pointer.

The final step in displaying your sprite is to link your SimpleSprite data structure and shape data into the simple sprite machine. The way you do this is by calling the ChangeSprite routine, which tells the sprite machine what shape you want the sprite to be. The format for this statement is

```
ChangeSprite (Vp, SimpleSprite, Sprite_data);
             (a0)     (a1)          (a2)
```

where the SimpleSprite and Sprite_data values are pointers to those two data structures. The Vp value stands for the address of the viewport, which can be found with the Intuition statement ViewPortAddress, which we discussed in Chapter 3. If you wish only to position the sprite relative to the current view, however, you can use a zero for this value. If the call to ChangeSprite is successful, the address of the sprite data structure is placed in the posctldata variable of the SimpleSprite structure, and your sprite is displayed at the position specified by the *x* and *y* variables in the SimpleSprite structure. You can use ChangeSprite not only to initially specify the shape of your sprite, but to change this shape while the sprite is on display as well.

While your sprite is being displayed, you may wish not only to change its shape, but to change its position as well. To move the sprite, use the command MoveSprite. The syntax for MoveSprite is

```
MoveSprite (Vp, SimpleSprite,  X,  Y);
           (a0)     (a1)     (d0) (d1)
```

where Vp is a pointer to the viewport (or zero if the sprite is positioned relative to the current view), SimpleSprite is a pointer to the structure of the same name, and X and Y specify the new position for the sprite.

The *x* and *y* position is relative to the entire display (not just the screen you are using). The position (0,0) places the top left corner of the sprite at the top left corner of the display screen. As the *x* value increases, the sprite moves to the right, and as the *y* value increases, the sprite moves downward. Remember, the horizontal resolution for sprites is only 320 across, so the maximum *x* value at which you can see at least part of the sprite is 319. Also, you should note that it is possible to use negative values for *x* and *y*. Using a negative value for *x* moves the sprite off the screen to the left, and using a negative value for *y* moves it off the top of the screen.

Once a sprite has been allocated to you, it can no longer be used by other programs or by the vsprite system of virtual sprites. Therefore, you should always remember to release the sprite as soon as you are finished using it. This is accomplished with the FreeSprite statement, whose syntax is

```
FreeSprite (Sprite_number);
                 (d0)
```

Notice that all that is required by this statement is the number of the sprite. This means that you could, in theory, free sprites used by other programs as well as your own. Obviously, such a practice would be bad manners at the very least and could well crash the system. Take care to free only the sprites allocated to you.

Program 6-1 is a C language program which shows the use of all of the SimpleSprite commands discussed.

Notice that we used the Graphics library routine WaitBOVP before moving the sprite. This routine uses the syntax

```
WaitBOVP(Vp);
         (a0)
```

where Vp is a pointer to the window's viewport, which we find with the Intuition function ViewPortAddress. The WaitBOVP call is used by the program to synchronize sprite movements with the video beam. Its function is to wait until the video beam gets to the bottom of the viewport. At that point, we can safely move the sprite without having to worry that the movement will take place while the video beam is redrawing the sprite. The result of such an occurrence could be a noticeable flicker or jerkiness to the motion.

Program 6-1. Simple Sprite Demonstration, C Example

```
#include <window.c>
#include <graphics/sprite.h>

struct SimpleSprite Sprite;

UWORD  Sprite_data[] =
{
Ø,Ø,         /* position, control */

ØxFFFF,ØxFFFF,
ØxFFFF,ØxFFFF,

ØxCØØ3,ØxCE73,
ØxCØØ3,ØxCE73,

ØxFF8F,ØxCØ73,
ØxFF8F,ØxCØ73,

ØxCØØ3,ØxCE73,
ØxCØØ3,ØxCE73,

ØxF1FF,ØxCEØ3,
ØxF1FF,ØxCEØ3,

ØxCØØ3,ØxCE73,
ØxCØØ3,ØxCE73,

ØxFFFF,ØxFFFF,
ØxFFFF,ØxFFFF,

Ø,Ø              /* end */
};

demo()
{
SHORT spgot;
SHORT n, r, dx, dy;

WVP = (struct ViewPort *) ViewPortAddress(Wdw);

spgot = GetSprite(&Sprite,3);
if (spgot != 3) exit (FALSE);
```

```
   Sprite.x = Sprite.y =Ø;
   Sprite.height = 14;

SetRGB4 (WVP,21,12,3,8);
SetRGB4 (WVP,22,3,13,4);
SetRGB4 (WVP,23,12,1Ø,4);

   ChangeSprite (Ø,&Sprite,Sprite_data);
   MoveSprite(Ø,&Sprite,6Ø,9Ø);

dx = -1;
dy = 1;
for (n=Ø; n < 2Ø; n++){
   if (n & 1) dy = -dy;
   else dx = -dx;

   r = Ø;
   while (r++<11Ø){
      WaitBOVP(WVP);
      MoveSprite (Ø,&Sprite,Sprite.x+dx,Sprite.y+dy);
      }
}

FreeSprite(spgot);

} /* end of Simpspr.c */
```

Using SimpleSprites from BASIC

The steps for using SimpleSprites from BASIC are largely the same as those required by C language programs. First, we must set up the sprite shape data and SimpleSprite data structures. Normally, BASIC programmers use subscripted arrays for such tasks, but in this case there are two problems with that approach. We must make sure that the sprite image data remains in chip memory, and BASIC has no mechanism for specifying where an array will be located in memory.

The second problem is that all variables of the same type are stored together, and space for new variables is allocated dynamically, while the program is running. This means that if a nonsubscripted variable is first used after an array is DIMensioned, the whole array is moved up in memory to make room for the new simple variable. The data for sprites must always stay in the same place, and since subscripted variables can move around, it is impractical to use them for this purpose.

One reasonable alternative is to use the Graphics library routine AllocRaster to allocate a small bit of free memory for storing sprite data. The AllocRaster statement was described in detail in Chapter 3, but to summarize, it allocates storage for data that is a certain number of bits wide by a specified number of lines tall. Since the routine returns the address of the memory that is allocated, you must use the DECLARE FUNCTION statement at the beginning of your program to let it know that AllocRaster will return a value. Of course, the DECLARE FUNCTION statement must be used in conjunction with the LIBRARY "graphics.library" statement that is used to open the Graphics library routines for access from BASIC. Remember also that you must have the graphics.bmap file in your current directory when you try to open the Graphics library.

The storage that you allocate with AllocRaster should be 16 bits wide, since that is smallest width of each data word that you will be using. The number of words to request depends on the size of your sprite. Each SimpleSprite data structure requires 6 words of storage, 2 for the pointer to posctldata, and 1 each for height, *x*, *y*, and num. In addition, each shape data structure requires 4 more words of overhead for the two zero words at the beginning and end of each shape definition. Therefore, you must reserve 10 data words in addition to 2 words for each line of shape data. For a shape that is 14 lines high, like the one defined in Program 6-2, you need a total of 38 words, 28 for shape data and 10 for the overhead of the SimpleSprite structure and posctldata.

Once you allocate RAM with the AllocRaster routine, it is lost to the system until you release it with the FreeRaster call. Always remember to free the memory you've allocated at the end of your program and to release exactly the same amount of memory as you initially requested.

After you have allocated the necessary RAM, you should clear it to all zeros. You could do this by using the POKE statement. In Program 6-2 we use the BltClear routine to let the blitter do the work.

Once you have the data space allocated and initialized, you will have to use the POKEW statement to fill in the necessary values. If the base address of data space is stored in the variable Sprite, then the height of the sprite must be stored in Sprite+4, the *x* position in Sprite+6, and the *y* position in Sprite+8. The num value at Sprite+10 will be filled in by the GetSprite call, and the two zero words at the beginning of the shape data take up the positions Sprite+12 through Sprite+15. The actual shape data begins at Sprite+16.

To allocate a sprite for your use, you use the GetSprite call, just as you would from C. Since GetSprite returns a value telling you what sprite was allocated, you must use DECLARE FUNCTION at the beginning of the program to let BASIC know that it should pass back this value. Your program should check the value returned and continue only if the sprite that you requested was allocated. Remember that sprite 0 is reserved for the mouse pointer, so don't try to allocate that sprite. Also keep in mind that sprite 1 shares color registers with sprite 0, so if you choose that sprite, its colors will always be the same as those of the pointer.

After you have allocated the sprite and set up the sprite data, you may use ChangeSprite to link the data and SimpleSprite structure with the operating system sprite machine. If you have stored the data as suggested above, with the SimpleSprite structure first, followed by the shape data, the addresses to pass to ChangeSprite are Sprite (the SimpleSprite stucture) and Sprite+12 (the shape data). This call will display your sprite in the chosen shape. You may use ChangeSprite again to change the shape to another form that you've set up in data. To move the sprite, use the MoveSprite routine just as you would from C.

Program 6-2 displays the same sprite shape as the C example in Program 6-1 and makes the sprite follow the mouse pointer around the screen.

Program 6-2. Simple Sprite Demonstration, BASIC Example

```
LIBRARY "graphics.library"
T$="Mouse moves sprite, CTRL-C to end"
WINDOW 2,T$,,Ø

DEFLNG A-z
DECLARE FUNCTION GetSprite() LIBRARY
DECLARE FUNCTION AllocRaster()LIBRARY
GOSUB InitSprite
```

```
 ON BREAK GOSUB Cleanup    'set cleanup on BREAK
 BREAK ON

 WHILE MOUSE(Ø)<4                'Continue 'til BREAK
 x=MOUSE(1)/2-16                  'find mouse x
 y=MOUSE(2)-4
 CALL WaitTOF
 CALL MoveSprite(Ø,sprite,x,y)
 'move paddle accordingly
 WEND

 Cleanup:
 FreeSprite(1)                  'free the sprite
 Cleanup1:
 CALL FreeRaster(sprite,16,38)  'free the memory
 WINDOW CLOSE 2                 'close the window
END

InitSprite:
 sprite = AllocRaster(16,38)
 'allocate memory for sprite shape data
 IF sprite= Ø THEN PRINT "No RAM":END
 'if memory can't be allocated, quit
 CALL BltClear (sprite,76,Ø)
 'use the blitter to clear the memory

 IF GetSprite(sprite,1) <> 1  THEN
   WINDOW OUTPUT 1
   PRINT "Can't Get Sprite"
   GOTO Cleanup1
  END IF
'try to allocate sprite 1-if can't, quit
 POKEW sprite+4,14              'set height of sprite
 POKEW sprite+6,MOUSE(1)/2-16   'set x
 POKEW sprite+8, MOUSE(2)-4     'set y

 FOR x=16 TO 68 STEP 4          'set shape = block
   READ A%,B%
   POKEW sprite+x,A%
   POKEW sprite+x+2,B%
 NEXT

 CALL ChangeSprite (Ø, sprite, sprite+12) 'take shape

RETURN


DATA &HFFFF, &HFFFF
DATA &HFFFF, &HFFFF
DATA &HCØØ3, &HCE73
DATA &HCØØ3, &HCE73
DATA &HFF8F, &HCØ73
DATA &HFF8F, &HCØ73
DATA &HCØØ3, &HCE73
DATA &HCØØ3, &HCE73
DATA &HF1FF, &HCEØ3
DATA &HF1FF, &HCEØ3
DATA &HCØØ3, &HCE73
DATA &HCØØ3, &HCE73
DATA &HFFFF, &HFFFF
DATA &HFFFF, &HFFFF
```

Attaching Simple Sprites

It is possible to create colorful sprites that can display up to 15 different colors at once, plus transparent, by combining two adjacent sprites. The procedure is fairly simple. First, create two sprites of the same size and position them at the same spot on the screen. These sprites must make up a single pair that normally shares the same color registers (for example, sprites 0 and 1, 2 and 3, and so forth). After you've used ChangeSprite to display them, set the SPRITE_ATTACHED bit in the second control word at the beginning of the shape-definition data for the second sprite. From then on, the two sprites will be attached.

When you attach two sprites in this manner, the two bit planes of shape data for each line of each sprite are combined to form four bit planes. The bit planes of the lower numbered sprite are used as planes 0 and 1 for the combined sprite, and the bit planes of the higher numbered sprite are used as planes 2 and 3. For instance, if you are using sprites 3 and 4, here is the order in which each word of each data line is used:

Plane Number	Data Word
0	First word of sprite 3
1	Second word of sprite 3
2	First word of sprite 4
3	Second word of sprite 4

Since there are four bits used to define each dot, each dot can be any one of 16 colors. The attached sprites use the upper 15 color registers. Color 0 being transparent, color 1 uses pen 17, color 2 uses pen 18, and so forth. Therefore, if the data words for line 1 of sprite 3 are 0x0000, 0xFFFF, and the data words for line 1 of sprite 4 are 0x0000, 0xFFFF, they will combine to form a color value of 1010 binary, or 10, for each dot. This means that the first line of the sprite will be a solid bar of pen color 26 (10+16).

Attached sprites use all 15 color registers only so long as the position of each sprite is identical. If you move them apart, they will be displayed as two separate sprites, each using up to three colors plus transparent. The color registers used to display those colors will differ from those normally assigned, however. Regardless of which pair of sprites is used, the lower numbered sprite will select its colors from registers 17–19, and the higher numbered sprite will use registers 20, 24, and 28, which are normally not used by any of the hardware sprites. Fortunately, it is easy to keep the two sprites together. If you use MoveSprite to move the lower numbered sprite of the pair, the routine will automatically move both sprites together.

Program 6-3 is a C language sample that shows the use of attached sprites. It creates a box-shaped sprite made up of stripes of every available color.

From BASIC, creating attached sprites is very similar to the process of creating two single sprites. The only real difference is that you must use the POKE statement to set the attach bit of the second sprite. The value for this attach bit is &H80 (128), and the control word whose bit you must set is located at an offset of three bytes from the beginning of the shape data block. Therefore, if your SimpleSprite data structure for the second sprite appears at address Sprite2, and the shape data comes directly after it at address Sprite2+12, then the control word that you must POKE is located at Sprite2+15. The correct statement to set the attach bit in this case would be

```
POKE Sprite2+15, PEEK(Sprite2+15) OR 128
```

We OR the contents of the control word with 128 in order to make sure that only the attach bit is changed.

Program 6-4, a BASIC program, shows how to create two attached sprites. It uses the 15-color striped sprite of the previous example and again lets you move the sprite with the mouse.

Program 6-3. Attached Sprites, C Example

```
#include <window.c>
#include <graphics/sprite.h>

demo()
{
SHORT spgot;
SHORT n, r, x, y, dx, dy;

struct SimpleSprite Sprite2;
struct SimpleSprite Sprite3;

UWORD  Sprite2_data[32];
UWORD  Sprite3_data[32];

Sprite2_data[Ø] = Sprite2_data[1] = Ø;
Sprite3_data[Ø] = Sprite3_data[1] = Ø;
Sprite2_data[3Ø] = Sprite2_data[31] = Ø;
Sprite3_data[3Ø] = Sprite3_data[31] = Ø;

for (n = 2; n<29 ;n+=2)
  {
  Sprite2_data[n] = Øx5555;
  Sprite2_data[n+1] = Øx3333;
  Sprite3_data[n] = ØxØFØF;
  Sprite3_data[n+1] = ØxØØFF;
  }

spgot = GetSprite(&Sprite2,2);
if (spgot != 2) exit (FALSE);

spgot = GetSprite(&Sprite3,3);
if (spgot != 3) exit (FALSE);

   Sprite2.x = Sprite2.y =Ø;
   Sprite3.x = Sprite3.y =Ø;
   Sprite2.height = Sprite3.height = 14;

   WVP = (struct ViewPort *) ViewPortAddress(Wdw);

   ChangeSprite (Ø,&Sprite2, Sprite2_data);
   ChangeSprite (Ø,&Sprite3,Sprite3_data);

   Sprite3_data[1]|=SPRITE_ATTACHED;
   MoveSprite(Ø,&Sprite2,6Ø,9Ø);

dx = -1;
dy = 1;
for (n=Ø; n < 2Ø; n++){
   if (n & 1) dy = -dy;
   else dx = -dx;
```

```
    r = Ø;
    while (r++<11Ø)
        {
        x=Sprite2.x+dx;
        y=Sprite2.y+dy;
        WaitBOVP(WVP);
        MoveSprite (Ø,&Sprite2,x,y);
        WaitTOF();
  }
  }

 FreeSprite(2);
 FreeSprite(3);

 } /* end of Attspr.c */
```

Program 6-4. Attached Sprites, BASIC Example

```
 LIBRARY "graphics.library"
 T$="Mouse moves sprites, CTRL-C to end"
 WINDOW 2,T$,,Ø

 DEFLNG a-z
 DECLARE FUNCTION GetSprite() LIBRARY
 DECLARE FUNCTION AllocRaster()LIBRARY
 GOSUB InitSprite
 ON BREAK GOSUB Cleanup    'set cleanup on BREAK
 BREAK ON

 WHILE MOUSE(Ø)<4                'Continue 'til BREAK
 x=MOUSE(1)/2-16                  'find mouse x
 y=MOUSE(2)-4
 CALL WaitTOF
 CALL MoveSprite(Ø,sprite,x,y)
 'move paddle accordingly
 WEND

 Cleanup:
 FreeSprite(2)                  'free the sprite
 FreeSprite(3).                 'free the sprite
 Cleanupl:
 CALL FreeRaster(sprite,16,76)  'free the memory
 WINDOW CLOSE 2                 'close the window
END

InitSprite:
 sprite = AllocRaster(16,76)
 'allocate memory for sprite shape data
 IF sprite= Ø THEN PRINT "No Ram":END
 'if memory can't be allocated, quit
 sprite2 = sprite+76
 CALL BltClear (sprite,152,Ø)
 'use the blitter to clear the memory
 IF GetSprite(sprite,2) <>2 THEN
   WINDOW OUTPUT 1
   PRINT  "Can't Get Sprite 2"
   GOTO Cleanupl
 END IF
 IF GetSprite(sprite2,3) <>3 THEN
   WINDOW OUTPUT 1
   PRINT  "Can't Get Sprite 3"
   GOTO Cleanupl
 END IF
' try to allocate sprites 2-3,
' if you can't, clean up and quit

 POKEW sprite+4,14  'set height of sprite
 POKEW sprite2+4,14

 FOR x=16 TO 68 STEP 4    'set shape = block
   POKEW sprite+x,&H5555
   POKEW sprite+x+2,&H3333
   POKEW sprite2+x,&HFØF
   POKEW sprite2+x+2,&HFF
 NEXT

 CALL ChangeSprite (Ø, sprite, sprite+12) 'take shape
 CALL ChangeSprite (Ø, sprite2, sprite2+12) 'take shape

'Set the ATTACH bit to attach the sprites
 POKE sprite2+15,PEEK(sprite2+15) OR 128

'Set sprite color registers randomly
RANDOMIZE TIMER
FOR p=2Ø TO 31
  PALETTE p, RND(1), RND(1), RND(1)
NEXT p

RETURN
```

The Intuition Pointer as a Sprite

By default, the Intuition mouse pointer is displayed as a red arrow, outlined in black and beige. You can change this default pointer with the Edit Pointer function of the Preferences program. The shape of the new pointer is then saved in a file called system-configuration in the devs: directory and is used every time you boot up with the system disk on which it resides.

It is also possible, however, to change the shape of the pointer under program control. The Intuition library includes a function called SetPointer that is used for this purpose. Its syntax is

```
SetPointer
  (Window, Sprite_data, Height, Width, XOffset, YOffset);
     (a0)        (a1)        (d0)    (d1)    (d2)      (d3)
```

The variable Window is a pointer to the Window data structure. The pointer is tied to whichever window is currently active. Therefore, when you use SetPointer, the shape of the pointer changes only when the window to which you've linked that new shape becomes active.

Since we have already established that Intuition uses hardware sprite 0 for the mouse pointer, it should hardly come as a surprise that the Sprite_data variable used by SetPointer is the address of the same kind of sprite data table as used by ChangeSprite. Again, the first two words of the data table are set to zero, followed by two words of shape data for each line of the pointer, and finally the last two words of the table, which are also set to zero.

The next two variables which you pass to SetPointer are the height of the pointer shape in lines and the width in dots. The width value must be less than or equal to 16, since that is the maximum width of a sprite.

The final two values specify the position of the pointer's *hot spot*. This is the point that is considered the exact current location of the pointer. In the default pointer arrow, the second dot of the second line of the pointer is considered to be the point. If you wish this point to register as another spot on the custom pointer that you have designed, you must specify *x* and *y* offsets to move this point from its default position of (0,0). A negative *x* offset moves this point to the right, and a negative *y* offset moves it down. For instance, if you have designed a pointer that is 15 dots wide and 15 dots high, you would specify an *x* offset of −7 and a *y* offset of −7 to center the hot spot right in the middle of your pointer. An *x* offset of −15 would move the point to the top right corner of your new pointer.

When you wish to change the pointer back to its default shape, you may use the Intuition function ClearPointer to do so. The format for this function is

```
ClearPointer(Window);
             (a0)
```

Program 6-5 demonstrates how to change the shape of the pointer in a window. It transforms the pointer into the popular crosshairs shape used by many drawing programs.

Setting up a new pointer shape from BASIC is very similar to setting up a simple sprite. Because of the tendency of subscripted arrays to move around in memory as new variables are defined, you must allocate space for the sprite shape data with AllocRaster, just as you do for simple sprites. In this case, however, you don't have to allocate room for the SimpleSprite data structure. You need only allocate enough space for the shape data and four extra words for the control words before and after the shape data. Once you have placed the shape data into memory, you may use the SetPointer call to change the pointer shape.

Program 6-6 shows how to change the shape of the pointer from the default arrow to crosshairs from BASIC.

In addition to creating custom pointers, SetPointer can be used to make the pointer disappear as well. This can be particularly useful for games in which you wish to limit the range of motion of the pointer. The BASIC program, Program 6-7, uses SetPointer to change the pointer to a one-line-high shape that is completely transparent. It then uses some simple sprites to substitute for the pointer, but limits their range of motion to the bottom line of the screen. The result is a new pointer that acts like the paddle used in many arcade games. Note that although the pointer is invisible, it still functions. If you press the right mouse button to activate the menu bar and move the mouse pointer to the top of the screen, you will find that you can make menu selections even though you cannot see the pointer.

Program 6-5. Changing the Shape of the Pointer, C Example

```
#include <window.c>

UWORD Sprite_data[] =
{
0,0,          /* position, control */

0x0440,0x02C0,
0x0440,0x02C0,
0x0440,0x02C0,
0x0440,0x02C0,
0x0440,0x02C0,

0xFC7E,0x02FE,

0x0000,0xFEFE,

0x0000,0x0000,

0x0000,0xFEFE,

0xFC7E,0x0FE80,

0x0440, 0x0680,
0x0440, 0x0680,
0x0440, 0x0680,
0x0440, 0x0680,
0x0440, 0x0680,

0,0               /* end */
};

demo()

{

/* set the new pointer */
SetPointer(Wdw, &Sprite_data[0], 15, 15, -7, -7);

} /* end of SetPoint.c */
```

Program 6-6. Changing the Shape of the Pointer, BASIC Example

```
LIBRARY "intuition.library"
LIBRARY "graphics.library"
T$="New Pointer Window"
T$=T$+" -- CTRL-C to end"
WINDOW 1,T$
```

```
 DEFLNG a-z
 DECLARE FUNCTION AllocRaster()LIBRARY
 GOSUB Init                  'change pointer
 ON BREAK GOSUB Cleanup      'set cleanup on BREAK
 BREAK ON

 WHILE 1                    'Continue 'til BREAK
 WEND

 Cleanup:
 CALL ClearPointer (WINDOW(7))  'restore pointer
 CALL FreeRaster(ptr,16,34)  'free the memory

END

Init:
 ptr = AllocRaster(16,34)
 'allocate memory for sprite shape data
 IF ptr = Ø THEN PRINT  "No RAM": END
 'if memory can't be allocated, quit

 FOR x=Ø TO 64 STEP 4           'set shape = block
   READ D1,D2
   POKEW ptr+x,D1
   POKEW ptr+2+x,D2
 NEXT

 CALL SetPointer ( WINDOW(7), ptr,15, 15, -7, -7 )
 'Change pointer shape

RETURN

DATA Ø,Ø
DATA &H44Ø,&H2CØ
DATA &H44Ø,&H2CØ
DATA &H44Ø,&H2CØ
DATA &H44Ø,&H2CØ
DATA &H44Ø,&H2CØ
DATA &HFC7E, &HØ2FE
DATA &HØ, &HFEFE
DATA &HØ, &HØ
DATA &HØ, &HFEFE
DATA &HFC7E, &HFE8Ø
DATA &H44Ø, &H68Ø
DATA &H44Ø, &H68Ø
DATA &H44Ø, &H68Ø
DATA &H44Ø, &H68Ø
DATA &H44Ø, &H68Ø
DATA Ø,Ø
```

Program 6-7. The Invisible Pointer

```
 LIBRARY "graphics.library"
 LIBRARY "intuition.library"
 T$="Mouse moves paddle, CTRL-C to end"
 WINDOW 2,T$,,Ø

 DEFLNG a-z
 DECLARE FUNCTION GetSprite() LIBRARY
 DECLARE FUNCTION AllocRaster()LIBRARY
 GOSUB InitSprite
 ON BREAK GOSUB Cleanup    'set cleanup on BREAK
 BREAK ON

 WHILE MOUSE(Ø)<4          'Continue 'til BREAK
 x=MOUSE(1)/2-16           'find mouse x
 CALL WaitTOF
 CALL MoveSprite(Ø,sprite,x,193)
 CALL MoveSprite(Ø,sprite+44,x+12,193)
'move paddle accordingly
 WEND

 Cleanup:
 FreeSprite(2)             'free the sprite
 FreeSprite(3)             'free the sprite
 Cleanupl:
 CALL FreeRaster(sprite,16,5Ø)  'free the memory
 CALL ClearPointer(WINDOW(7))
 WINDOW CLOSE 2                 'close the window
END

InitSprite:
 sprite = AllocRaster(16,5Ø)
 'allocate memory for sprite shape data
 IF sprite= Ø THEN PRINT "No RAM": END
 'if memory can't be allocated, quit
 sprite2 = sprite+44

 CALL BltClear (sprite,1ØØ,Ø)
 'use the blitter to clear the memory

 IF GetSprite(sprite,2) <>2 THEN
    WINDOW OUTPUT 1
    PRINT "Can't get Sprite 2"
    GOTO Cleanupl
  END IF

  IF GetSprite(sprite2,3) <>3 THEN
    WINDOW OUTPUT 1
    PRINT  "Can't get Sprite 3"
    GOTO Cleanupl
  END IF
  ' try to allocate sprites 2-3,
  ' if can't, quit
 POKEW sprite+4,6      'set height of sprite
 POKEW sprite2+4,6

 FOR x=16 TO 36 STEP 4   'set shape = block
   POKEW sprite+x,-1
   POKEW sprite+x+2,Ø
   POKEW sprite2+x,-1
   POKEW sprite2+x+2,Ø
 NEXT

 CALL ChangeSprite (Ø, sprite, sprite+12)
    'take shape
 CALL ChangeSprite (Ø, sprite2, sprite2+12)
    'take shape
 CALL SetPointer(WINDOW(7),sprite+88,1,1,Ø,Ø)

RETURN
```

Vsprites and Bobs

As their name would suggest, simple (or hardware) sprites operate fairly closely to the hardware model for sprites, and are relatively simple to create and manipulate. The two other animation objects supported by the Amiga, vsprites and bobs, require a much greater amount of software overhead for their operation. This makes them much more versatile than simple sprites, but also makes them correspondingly more complex.

Indeed, the Amiga animation system is so intricate that it would take an entire book to explain it adequately. Therefore, what is provided here is more of an overview of vsprite and bob operations than an exhaustive explanation. The BASIC interpreter provides a simple means of accessing a full range of vsprite and bob commands, so we'll be using a number of BASIC programs to illustrate the features of these animation objects.

Although vsprites and bobs are similar in many ways, the hardware basis for each is very different. Vsprites are based on the hardware sprites and share many of their limitations, such as size (16 pixels wide) and number of colors (a maximum of three plus transparent).

The animation software, however, provides a way of overcoming some of the limitations of hardware sprites, such as the restriction on the total number of sprites that can be displayed onscreen at once. While there are only eight hardware sprites, the vsprite system can dynamically allocate these hardware sprites so that one hardware sprite can be used to display several *virtual sprites* in different parts of the screen at once. In addition, it can change the color registers used by these sprites from line to line so that a sprite that appears in one set of colors at the top of the screen may appear in a totally different set of colors at the bottom of the screen. The only limitations are those imposed by the hardware; you can't have more than eight vsprites on a single horizontal line at once, and you may be limited to fewer than that if the colors used by the vsprites are all different.

Bobs are animation objects that are created by using the normal display hardware, not the separate sprite system. This playfield animation system uses the blitter chip to sequentially save the background area of the image destination, move the image, and restore the background of the image source.

Since bobs are part of the bitmap, their size and color limitations are the same as those of the bitmap. You can display as many bobs as will fit on the screen, and they may be as large as the bitmap display and may have as many colors as are supported by the screen upon which they are displayed. Their principal drawback is one of speed. If you try to move very large bobs, or a lot of them at the same time, the enormous overhead requirements may make things somewhat slow, even with the powerful hardware assistance of the blitter.

Displaying Vsprites and Bobs

Though the hardware basis of vsprites and bobs is different, the system treats them very similarly, and so we shall discuss them together. In the Amiga animation system, the individual animation objects like vsprites and bobs are referred to as GELs (short for *G*raphics *EL*ements). To display one of these GELs, you must first create a linked list of graphics elements. You then set up data for each of your vsprites or bobs, and add it to this list. The data you create for each GEL includes items like its size, shape, and position. Before displaying the GELs, you must sort the list to get them into the order in which they will be displayed on the screen. Then you ask the animation system to display them. Every time you make a change to one of the members of the list, you must sort the list again before displaying the changes.

The first step is to declare a GelsInfo structure, initialize it, and link it into the rastport into which you will be displaying the vsprites or bobs. You initialize the GelsInfo structure with the InitGels statement. The syntax for this statement is

```
InitGels(vsprite1, vsprite2, GelsInfo);
         (a0)      (a1)      (a2)
```

where vsprite1 and vsprite2 are dummy VSprite data structures that are used to mark the beginning and end of the GEL list, and GelsInfo is the address of the GelsInfo data structure that will be used to keep track of the GEL information. You must link this initialized GelsInfo structure into your rastport by placing its address in the rastport GelsInfo variable. The following C language fragment demonstrates the process of initializing and linking the GelsInfo structure:

```
struct VSprite vs1, vs2;
struct GelsInfo GInfo;
InitGels(vs1, vs2, GInfo);
RastPort->GelsInfo = &GInfo
```

The next step is to set up the data that defines the shape and color of your image. The format of the image data that you'll create depends on whether you are using a vsprite or a bob. Vsprites use the format for sprite shape data that we discussed above; two 16-bit words of data for each line of the sprite, where the first word is used for color plane 0, and the second word is used for color plane 1. For bobs, you must set up a shape data table similar to the one used by the Intuition Image structure that we discussed in Chapter 5. There are as many bit planes as the image is deep, and each plane is composed of a number of lines of data. Each line contains as many 16-bit words as are necessary to hold the widest part of the image, and there are as many lines as the image is high.

The next data structure that you must set up is the VSprite structure. The C language definition of this structure looks like this:

```
struct VSprite
  {
    struct VSprite *NextSprite;
    struct VSprite *DrawPath;
    struct VSprite *ClearPath;
    WORD OldY, OldX;
    WORD Flags;
    WORD Y,X;
    WORD Height;
    WORD Width;
    WORD Depth;
    WORD MeMask;
    WORD HitMask;
    WORD *ImageData;
    WORD *BorderLine;
    WORD *CollMask;
    WORD *SprColors;
    struct Bob VSBob;
    BYTE PlanePick;
    BYTE PlaneOnOff;
    VUserStuff VUserExt;
  };
```

As you can see, the VSprite structure contains a lot of variables which describe many different aspects of the vsprite's display. Not all of these variables must be initialized by the user, however. The first six variables, for example, are for the use of the system. And the last member of the structure provides a place for users to add their own extensions to the animation system.

The variables Y and X are used to store the vertical and horizontal positions of the GEL. The Height, Width, and Depth variables are used to describe the size and color resolution of the object (note that vsprites will always be 16 bits wide and two color planes deep).

The ImageData variable is used to store the address of the shape data table that you created to describe the object's shape. The SprColors variable is used only for vsprites and points to a table that contains three 16-bit words of color data. These contain the colors for planes 1, 2, and 3 (plane 0 is always transparent). The format for this color table is 0x0RGB, where the *R* stands for a 4-bit red color level value, *G* stands for a 4-bit green color level value, and *B* stands for a 4-bit blue color level value. Therefore, a table consisting of the data words 0xF00, 0x0F0, and 0x00F would contain the colors red, green, and blue.

The colors used by bobs are the same as those used by the rest of the background graphics. You may use the PlanePickOn and PlanePickOff variables, however, to vary the colors that are displayed. Use of these variables is covered in the section on Intuition Images at the end of Chapter 5, and is also summarized in the section on BASIC animation objects below.

The Flags variable can be set to the value of one or more flags. These control the way in which the object will be displayed. The flag VSPRITE should be set if the object is a vsprite, but not if it is a bob. The other flags that can be set by the user concern the way a bob is displayed. The SAVEBACK flag indicates that the background is to be saved before the bob is drawn and restored after it is moved. If this option is desired, the user must provide a storage area for the saved background image that is as large as the bob shape data area.

The OVERLAY flag indicates that portions of the bob image definition that use the background color (pen 0) should be treated as transparent when moving the bob. If this flag is not set, the entire rectangle of the bob, background and all, will cover the image it is placed on top of.

The MeMask, HitMask, CollMask, and BorderLine variables are used in detecting a collision between GELs or between a GEL and the screen borders. CollMask and BorderLine are used by the system to maintain collision masks. CollMask should point to a storage area the size of one of the image's bit planes, while BorderLine should point to a storage area the size of the width of one line of image data. After setting these pointers, you may initialize the contents of these masks with the call InitMasks. The format for this statement is

```
InitMasks(VSprite);
          (a0)
```

where VSprite is a pointer to the VSprite structure that contains the addresses of the mask areas. HitMask and MeMask are used to determine the kinds of collisions between objects that the system will detect. Use of these variables is discussed below in the section on collision detection for BASIC animation objects.

The final variable, VSBob, is used only if there is a bob associated with this vsprite. If this is the case, it should be set to point to the Bob data structure.

If the GEL that you're setting up happens to be a bob, then you must initialize a Bob data structure in addition to the VSprite structure. Here is the C language declaration for this structure:

```
struct Bob
  {
   WORD Flags;
   WORD *SaveBuffer;
   WORD *ImageShadow;
   struct Bob *Before;
   struct Bob *After;
   struct VSprite *BobVSprite;
   struct AniComp *BobComp;
   struct DBufPacket *DBuffer;
   BUserStuff BUserExt;
  };
```

The Flags variable contains only a couple of user settings. A setting of SAVEBOB tells the system not to erase the old bob image when moving the bob. This allows you to use the bob like a paintbrush. The BOBISCOMP flag is used to show that this bob is part of a larger animation object known as an AnimComp. If this is the case, the variable BobComp is set to point to the address of the AnimComp structure.

The BobVSprite variable should contain the address of the VSprite data structure associated with the bob. Although not every vsprite has a corresponding bob, every bob must have a corresponding VSprite data structure.

The SaveBuffer variable should be set to point to a buffer area that is as large as the shape data area for the bob. This is where the background image behind the bob is temporarily saved. The ImageShadow variable should contain the address of another buffer area that is the same size as the Collision Buffer or as large as one bit plane of the bob image. In fact, you'll often use the same area for the two buffers.

Bob Priority

The variables marked Before and After are used to determine the priority between bobs. Normally, the order in which bobs are drawn depends strictly upon their positions on the screen. The bob that is drawn last is said to have the highest priority, because if it overlaps with another bob, it will cover up part of that bob and thus appear to be in front of it.

When it is important to have one bob always appear to be in front of another, regardless of their positions, you can specify the drawing order with these two variables. For example, if you want Bob1 to always appear in front of Bob2, you would set the After variable of Bob1 to point to the Bob2 data structure to make sure that Bob1 is always drawn after Bob2. Also, you would have to set the Before variable of the Bob2 data structure to point to the Bob1 data structure to make sure that Bob2 is always drawn before Bob1. The C code would look like this:

```
Bob1.After = &Bob2;
Bob2.Before = &Bob1;
```

If the priority of the rest of the bobs doesn't matter, set all the unused Before and After variables to zero.

The last variable is called Dbuffer. It points to a buffer area that is used in a special technique known as double-buffering. When you use double-buffering, you set up two separate display areas. While you are drawing into one area, you display the other area, then switch when the drawing is done. That way, the drawing never takes place while the user is watching. This prevents the flicker and smear effects that can occur when part of the screen contains the old image and part contains the new. Setting up a double-buffered display is a more advanced technique, and we will not discuss the details fully here. However, you should remember that if you're using a large number of bobs, or very big ones, you may need to perform double-buffering to make them move smoothly. If you are not using double-buffering, this value should be set to zero.

Add Bob to the GEL List

Once you have set up the image, vsprite, and bob data, you may add the bob to the GEL list with the call AddBob:

```
AddBob(Bob, RastPort);
       (a0)  (a1)
```

where Bob is the address of the Bob data structure, and RastPort is the addresss of the rastport into which the bob is to be drawn. If you are using vsprites rather than bobs, the call is

```
AddVSprite(VSprite, RastPort);
           (a0)     (a1)
```

When you have added all of your GELs, you must put them into order before displaying them. Use the SortGList statement to sort the members of the GEL list by vertical and horizontal positions:

```
SortGList(RastPort);
          (a1)
```

Finally, you may display bobs with the call DrawGList, which takes the form

```
DrawGList(RastPort, ViewPort);
          (a1)      (a0)
```

When you execute this statement, the bobs are actually drawn on the screen. This drawing is performed in synchronization with the electronic raster beam to keep the display smooth. Nonetheless, if you are using a large bob or many smaller ones, you may have to take steps of your own to make sure that the display does not flicker when you move them.

While DrawGList actually draws the bobs, it does not display vsprites, since their display depends on the contents of the hardware registers of the display chips, rather than the contents of display memory. DrawGList can only generate new copper instructions to change the contents of these registers. In order to merge these instructions in with the present copper instruction list, you must use the following two routines:

```
MrgCop(View);
       (a1)
```

and

```
LoadView(View);
         (a1)
```

Program 6-8 gives a brief demonstration of creating a bob and moving it around the screen.

If you wish to change some aspect of the bob data, such as its shape or position, you must call SortGList and DrawGList after making the change so that it will be displayed. If you wish to remove a bob from the GELs list entirely, you may do so with the RemBob statement. The form of this statement is the same as that of AddBob:

```
RemBob(Bob, RastPort);
       (a0)  (a1)
```

The OBJECT Commands—Vsprites and Bobs in BASIC

As mentioned at the beginning of the chapter, Amiga BASIC provides full support for the operating system animation routines. This support takes the form of a number of statements, all beginning with the word OBJECT.

Like the operating system animation routines, all of the Amiga BASIC OBJECT statements described below work with either bobs or vsprites. There are slight variations on how these statements are carried out, however, due to the inherent differences in the nature of these two kinds of objects. Whether you choose to make an object a bob or a vsprite will depend to some degree on the needs of your program. Here are some of the differences between the two object types which you should consider.

Size and resolution. Bobs have the same resolution as the screen on which they appear, while vsprites always appear in low resolution. Vsprites can be only 16 pixels wide; bobs can be virtually any size, so long as you have enough memory to store their shape data.

Number of objects. Only eight vsprites of the same color can appear on a single horizontal line, and only four of different colors. Theoretically, at least, you can display as many bobs onscreen as you want. In practice, however, you'll find that Amiga BASIC does not handle a lot of bobs any better than it does a lot of vsprites. In fact, the amount of jitter introduced by numerous bobs onscreen at once may make them quite unattractive when used in large numbers.

Program 6-8. Moving Bobs

```
#include <windowl.c>
#include <graphics/gels.h>

WORD Image_data[4 * 4Ø * 3];
WORD Sbuffer[4 * 4Ø * 3];
WORD Cmask[4 * 4Ø];
WORD Bline[4];

struct VSprite v =
{
    NULL, NULL, NULL,
    NULL, NULL, NULL,    /* system variable */
    OVERLAY | SAVEBACK,  /* Flags */
    6Ø,    /* Y position  */
    1Ø,    /* X position */
    4Ø,    /* Height = 4Ø lines high */
    4,     /* Width = 4 words per row = 64 bits */
    3,     /* Depth = 3 planes deep, same as Screen */
    1,     /* MeMask  */
    1,     /* HitMask  */
    &Image_data[Ø],      /* ImageData */
    &Bline[Ø],  /* BorderLine  */
    &Cmask[Ø],    /* CollMask   */
    NULL,  /* bob doesn't use sprite colors */
    NULL,  /* pointer to bob filled in below */
    Øx07,  /* PlanePick = pick 3 planes */
    ØxØØ,  /* PlaneOnOff = all Ø's to unpicked plane */
    NULL   /* No user extensions */
};
```

```
struct Bob b =
{
     Ø,                  /* Flags */
     &Sbuffer[Ø],     /* SaveBuffer --to save backgnd */
     &Cmask[Ø],       /* ImageShadow -- coll = shadow */
     NULL,               /* Before  */
     NULL,               /* After */
    /* drawing order doesn't matter */
     &v,    /* BobVSprite   -- link vsprite to its bob */
     NULL,  /* BobComp  -- bob isn't animation component */
     NULL,  /* DBuffer  -- not double buffered */
     NULL   /* No User Extension   */
};


demo()
{

struct VSprite sl, s2;      /* dummy sprites for gels list */
struct GelsInfo gelsinfo;  /* gelsinfo to link Rp */
struct collTable Ctable;

WORD dx;
WORD dy;
WORD x;

/******* Initialize Gels Info **********/

     gelsinfo.nextLine = NULL;
     gelsinfo.lastColor = NULL;
     gelsinfo.collHandler = &Ctable;
     InitGels(&sl, &s2, &gelsinfo);
     Rp->GelsInfo = &gelsinfo;

/******* Init image data, masks **********/

     for(x=Ø; x< 157; x+=4)
       {
       Image_data[x]=ØxFFFF;
       Image_data[x+1]=ØxFFFF;
       Image_data[x+2]=ØxØØØØ;
       Image_data[x+3]=ØxØØØØ;

       Image_data[x+16Ø]=ØxFFFF;
       Image_data[x+161]=ØxØØØØ;
       Image_data[x+162]=ØxØØFF;
       Image_data[x+163]=ØxFFØØ;

       Image_data[x+32Ø]=ØxFFØØ;
       Image_data[x+321]=ØxFFØØ;
       Image_data[x+322]=ØxØØFF;
       Image_data[x+323]=ØxØØFF;
       }

     v.VSBob = &b;
     InitMasks(&v);

/* ************* add the Bob and draw it ******* */

     AddBob(&b, Rp);
     SortGList(Rp);
     DrawGList(Rp,WVP);

/* ***************** move it around ******* */
```

```
     dx = 1;
     dy = 2;

for(x=Ø;x<1ØØØ;x++)
     {
     if( (b.BobVSprite->Y += dy) > 18Ø || b.BobVSprite->Y < 1Ø)
       dy = -dy;
     if( (b.BobVSprite->X += dx) > 3ØØ || b.BobVSprite->X < 1Ø)
       dx = -dx;
     SortGList(Rp);
     if (b.BobVSprite->Y < 9Ø)
        WaitTOF();
     DrawGList(Rp,WVP);
}

}/* end of bob.c */
```

Speed of motion. Vsprites move quickly, but bobs can be slower, particularly if you're using very large bobs or many of them. Generally speaking, the more bobs, the slower they move. If you create a lot of bobs, they can be so slow as to make their use impractical from BASIC.

Number and selection of colors. Bobs can use the maximum number of colors available on the screen on which they appear. They are limited to the exact same color selection as any other bitmap object that is drawn on that screen. Vsprites can have only three foreground colors and one background color. But these colors can be completely different from the ones selected for the rest of the screen. The only limitation is that no more than four vsprites of different colors can appear next to each other on the display. Vsprites can therefore be used to add color to a display without using up more bit planes' worth of memory. The additional colors are made available by changing the sprites' color registers as vsprites move up and down the screen. Since vsprites use the same color registers as the upper 16 bitmap pens' registers, bitmap objects drawn in these colors may change color as the vsprites move up and down. For this reason, it is not advisable to use vsprites on 32-color screens.

Color priority. Bobs have a selectable priority; you can determine which will be displayed in front of the others. As implemented by BASIC, vsprites always appear in front of bobs and in a fixed order in front of each other.

Hardware system used. Because bobs are part of the normal bitmap display, they fit much better into the windowing environment of Intuition. They move when their windows are moved, they never move outside the borders of their windows into other windows, and they disappear when their windows are covered or closed. None of this can be said for vsprites. Because they use a completely different hardware display system, sprites don't stay in their windows and will be displayed even after the window is closed. They can also cause color conflicts with the mouse pointer. Because the pointer is actually a sprite, its color registers may be affected when the operating system software changes the sprite color registers. This means that the pointer color may be different in one horizontal part of the screen than in another when you use vsprites. This can be solved, but at the cost of reducing the number of hardware sprites, and thus vsprites, available.

Creating and Displaying OBJECTs

The first step in creating a movable object is to define its shape. This is done by using the Amiga BASIC program ObjEdit, which is found in the BasicDemos drawer of the Amiga Extras disk. This program allows you to use the mouse to draw a bob or vsprite image and then save that image to a disk file. The format of the disk file that the ObjEdit program saves is such that it can be read by your program and used to form a movable object in that image.

Instructions for using the ObjEdit program are found in your *Amiga BASIC* manual. You should remember, however, that if you edit a sprite using the program supplied, the image of the sprite in the editor will be only half as wide as the vsprite object that appears in your program, because the editor uses the high-resolution mode (640 dots across), while vsprites always appear in low resolution (320 across). Also, unless you alter the program as indicated in the REMarks at the start of the listing, it will edit only four-color objects.

Once you have drawn the shape and saved its image to a file (which for purposes of this example we will name ImageFile), the next step is to read that file into a string in your program. The statement lines that your program may use to accomplish this task are

```
OPEN "ImageFile" FOR INPUT as 1
ObjectImage$ = INPUT$(LOF(1),1)
CLOSE 1
```

These statements read the entire image file into one long string. Once the information resides in this string, it may be used by the OBJECT.SHAPE statement to create an object having that shape. The syntax for this is

OBJECT.SHAPE object_num, shape_definition_string

where the object_num value is a number greater than zero that you assign to the object to identify it for future commands, and shape_definition_string is the string into which you have read the image file information (here, ObjectImage$). Once you assign the shape data in the string to the object, that string is no longer needed, and you may free up the memory it required by assigning its value to that of the null string (" ").

The OBJECT.SHAPE statement also allows you to create a new object which has exactly the same shape as an existing object. The syntax for this form is

OBJECT.SHAPE new_object_num, existing_object_num

where the value new_object_num is the identification number of the new object that you are creating, and existing_object_num is the identification number of the object whose shape you are using. When you create a new object using this form of OBJECT.SHAPE, both objects share that memory area where the image data is shared, and this saves some memory. In all other ways, however, the two objects are separate and may be treated as unique objects. As we will see, they may even be of different colors.

Once you have assigned a shape to an object or objects, you need only to give the OBJECT.ON statement in order to display them. The format is

OBJECT.ON [object_num [,object_num...]]

where the values marked object_num are an optional list of the identification numbers of the objects that you wish displayed. If you supply a list of one or more object numbers, only those objects will be displayed. If you use OBJECT.ON with no object_num values, all objects that have been defined using OBJECT.SHAPE will be displayed.

To suspend the display of an object temporarily, you may use an OBJECT.OFF statement of the form

OBJECT.OFF [object_num [,object_num...]]

where the values marked object_num are an optional list of the identification numbers of the objects that you wish to disappear. As above, if you supply a list of one or more object numbers, only those objects will vanish, but if you use the command with no object_num values, all objects that have been defined using OBJECT.SHAPE will be turned off.

To disable an object permanently and release all of the memory associated with maintaining its shape and other attributes, you may use the OBJECT.CLOSE statement whose syntax is

OBJECT.CLOSE [object_num [,object_num...]]

where the values marked object_num are an optional list of the identification numbers of the objects that you wish to disable. As before, if you supply a list of one or more object numbers, only those objects will be closed, but if you use the command with no object_num values, all objects that have been defined using the OBJECT.SHAPE statement will be closed.

Setting the OBJECT Color

Vsprites and bobs use different mechanisms for determining the colors in which the object will be displayed. Vsprites use some of the upper 15 color registers, as explained in the section on hardware sprites, above, and they change the contents of those registers as they move. The colors that a vsprite will display are determined by the last six bytes of its ObjEdit file. These six bytes contain three byte pairs representing the three foreground pens. Each pair has the red value in the first byte, and the green and blue-green packed in the second byte. Each color value is represented by a number from 0 to 15. The green-blue byte contains a number that is the sum of 16 times the green value plus the blue value. In other words,

grnblu =16*green + blue

The ObjEdit program always sets the colors of the three foreground pens to white, black, and orange. If you wish to use other colors for your vsprites, you must alter the ObjEdit program or the file that it produces, or change the color values in the string after it has been read in from the file. Since the last is the simplest approach, it is the one we'll use.

The following program fragment demonstrates how to change the string. It should appear in your program after the image file has been read into the string ObjectImage$, and before the OBJECT.SHAPE command assigns the image in the string to an object. We use the colors black (0,0,0), purple

(15,0,15), and cyan (0,15,15), but you can change the red and grnblu values to suit your needs.

```
L=LEN(ObjectImage$)
red1 = 0
grnblu1 = 0
red2 = 15
grnblu2 = 0 * 16 + 15
red3 = 0
grnblu3 = 15 * 16 +15
Col$=CHR$(red1) + CHR$(grnblu1)
Col$ = Col$+CHR$(red2) + CHR$(grnblu2)
Col$ = Col$+CHR$(red3) + CHR$(grnblu3)
MID$(ObjectImage$,L-5) = Col$
```

Bobs, on the other hand, take their colors from the same pens as any other normal graphics image on the screen. Which color pen is used to draw the bob is dependent on the bit image data that you create with the ObjEdit program. You can change these colors with the PALETTE statement, but the rest of the graphics images that were drawn with the same pen will change also.

The OBJECT.PLANES statement allows you to change the pen used by your bob without changing the composition of its bit planes. It is not really useful for vsprites, since their color selection works differently, as explained above. The syntax is

OBJECT.PLANES object_num [,PlanePick] [,PlaneOnOff]

PlanePick and PlaneOnOff can be thought of as masks that can change the normal order in which the bit planes are displayed. Their use was described in detail in Chapter 5, in the section on "Intuition Images," but we will summarize that information again because it is so useful.

PlanePick is used to determine what bit planes are used for the display. Let's say that you have a two-plane image that uses pens 2 and 3, and you want to display it on a three-plane screen. Normally, the two planes would be displayed in planes 0 and 1. But you can set PlanePick to display these as two entirely different planes. You choose these planes by setting PlanePick to the sum of the bit values of the planes in which you wish the object displayed. Each bit value corresponds to 2^*n*, where *n* is the plane number.

For instance, the bit value of plane 0 is 1 (2^0), the bit value of plane 1 is 2 (2^1), and so on. The PlanePick value that corresponds to the normal setting of planes 0 and 1 would be 3 (1+2). To display the image in planes 1 and 2, you would set the PlanePick value to 6 (2+4). The part of the image that was created using pen 1 will now be displayed in the color of pen 2, and the part of the image that was created using pen 2 will now be displayed in the color of pen 4. The part of the image that was originally colored in pen 3 (both planes set) will now be shown in the color of pen 6.

The PlaneOnOff value can be used to further enhance the selection of colors. Let's say that in the above example you wanted to display your object in pen colors 3, 5, and 7 instead of 2, 4, and 6. Using PlanePick alone, this would not be possible, since each of these colors requires that two color planes be set. PlaneOnOff lets you set the color planes that were not chosen in PlaneOnOff.

In our example, an image that originally used planes 0 and 1 (pen colors 1, 2, and 3) was changed to use planes 1 and 2 (pen colors 2, 4, and 6). PlaneOnOff lets you set plane 0 as well. If you choose a PlaneOnOff value of 1, which corresponds to plane 0, everywhere that a pixel is set in plane 1 or 2 will also be set in plane 0. This has the effect of adding 1 to the pen values made possible by PlanePick. If PlanePick is set to 6, and PlaneOnOff is set to 1, the parts of the object that were originally drawn in pens 1, 2, and 3 will appear in pen colors 3, 5, and 7.

OBJECT Priority

When two bobs overlap, there is a question as to which one is drawn on top of the other. Left to its own devices, the operating system will make its own determination based on the position of the objects. If you wish one object always to be displayed in front of others, you may specify this with the OBJECT.PRIORITY statement. This statement takes the form

OBJECT.PRIORITY object_num, priority

where object_num is the identification number of the object, and the priority value is a number from −32768 to 32767.

Objects with a higher priority number are displayed on top of objects with a lower priority number. Note that this statement applies only to bobs; vsprites always appear in front of normal graphics objects like bobs.

Positioning and Moving OBJECTs

You position your movable objects with OBJECT.X and OBJECT.Y. These statements use the syntax

OBJECT.X object_num, x_position
OBJECT.Y object_num, y_position

where object_num is the object ID, and the x_position and y_position values are the coordinates of the top left corner of the object. Although vsprites are always displayed in low resolution (320 pixels across), their x_position values are relative to the screen resolution. If vsprites appear on a high-resolution screen, their visible range of motion is from −15 to 639. This range is not affected at all by the size of the current output window, unlike that of bobs, which can be seen only in the visible part of their windows. Regardless of the visible range of the object, the position commands will keep track of an object's position through the range of −32768 to 32767.

You may find that the positions of the objects do not change immediately when an OBJECT.X or OBJECT.Y is issued. If no objects are in motion, you may have to wait until a motion command or another command that affects the display occurs.

The position statements also may be used as functions to determine the current *x* and *y* position of an object. The syntax for the functions is

x_position = OBJECT.X (object_num)
y_position = OBJECT.Y (object_num)

where x_position and y_position represent the current coordinates for the object whose ID number is object_num.

Normally, a bob will be displayed if positioned anywhere within its window. It is possible to further restrict the visible range of a bob with the OBJECT.CLIP statement. The format is

OBJECT.CLIP (left,top)-(right,bottom)

where the first pair of coordinates represents the top left corner of the visible area, and the second pair specifies the bottom right corner. If you position the bob anywhere outside the specified area, it will not be displayed. Although clipping does not apply to vsprites, the OBJECT.CLIP statement sets the boundaries for the purpose of collision detection (see below) for both bobs and vsprites.

While it's possible to move your graphics objects by changing their *x* and *y* positions, it may require a number of program statements to keep them in motion. Amiga BASIC provides statements which let you move these objects at a constant rate of speed with just a couple of statements. These statements are OBJECT.VX and OBJECT.VY, and their syntax is

OBJECT.VX object_num, x_velocity
OBJECT.VY object_num, y_velocity

The x_velocity and y_velocity values represent the speed of the object in pixels per second. A positive *x* value moves the object to the right, and a positive *y* value moves the object down. Negative velocity values move the object in the opposite direction.

After you have set the velocity for an object, you must use OBJECT.START to set it in motion. This statement takes the form

OBJECT.START [object_num [,object_num...]]

If you specify a list of one or more object_num values, only those objects will start moving. If no object_num value is given, all previously defined objects will move.

To stop an object, you can use the OBJECT.STOP statement, whose syntax is

OBJECT.STOP [object_num [,object_num...]]

This statement will also apply to specific objects only if a list of object_num values is furnished. Otherwise, all motion is stopped. An object's motion is also stopped when it is made invisible with an OBJECT.OFF statement.

Once an object is put into motion, it will keep going until it collides with a border or with another object. Such a collision has the same effect on the object as an OBJECT.STOP statement. Therefore, if you want to keep the object in motion, you must periodically check for collisions. This can be accomplished either by using the OBJECT.X and OBJECT.Y functions to check the position of the object, or by using the ON COLLISION statement discussed below to change its direction when it reaches the border of the screen. When you have detected a collision, you must start up the object again with an OBJECT.START statement.

Like the positioning statements, the velocity statements also can be used as functions to determine the current velocity of an object. The syntax of the functions is

x_velocity = OBJECT.VX (object_num)
y_velocity = OBJECT.VY (object_num)

where x_velocity and y_velocity are the current velocities of object_num that are returned by the function.

Program 6-9 brings together a number of the elements discussed above. Since this book cannot transmit the equivalent of an image file created by the ObjEdit, we use a subroutine called InitImage to create the string equivalent of such a file. The image described by the file is that of a bob in the shape of a flying saucer. This image is assigned to two bobs, and the color of one is changed using the OBJECT.PLANES statement. Both bobs are then positioned and set into motion.

Just as the velocity statements allow you to change the position of an object automatically, Amiga BASIC includes acceleration statements that allow you to change the velocity automatically. The format of these statements is

OBJECT.AX object_num, x_acceleration_rate
OBJECT.AY object_num, y_acceleration_rate

where x_acceleration_rate and y_acceleration_rate represent the velocity to be added to an object's current velocity every second. In other words, it specifies the velocity change in pixels per second.

The above caution to watch an object once it is set in motion applies even more strongly when you use the acceleration statement. A high rate of acceleration may cause an object to hit a border very quickly. It may even develop "escape velocity," where it is redrawn at such large intervals that collision checking no longer works. In such a case, it will not stop at the border, but will keep right on going and disappear from the display entirely.

These statements also may be used as functions to determine an object's current acceleration rate. The format of the acceleration functions is

x_acceleration_rate = OBJECT.AX (object_num)
y_acceleration_rate = OBJECT.AY (object_num)

where the function returns the *x* or *y* acceleration rate of object_num.

Program 6-9. Flying Bobs

```
    WINDOW 1,"Unidentified Flying BOBs",(0,0)-(300,186),4

   GOSUB InitImage
   'create ShipShape$ from data,
   'instead of reading image file

   OBJECT.SHAPE 1, ShipShape$  'create first space ship
   OBJECT.Y 1,50               'position it vertically
   OBJECT.VX 1,60              'give it horizontal motion

   OBJECT.SHAPE 2,1      'create second ship
   OBJECT.PLANES 2,2,1  ' make it white with orange windows
   OBJECT.X 2,150        'position white ship horizontally
   OBJECT.Y 2, 180       'and vertically
   OBJECT.VY 2,-45       'give it vertical velocity upward

   OBJECT.ON             'display both ships
   OBJECT.START          'start them moving

  FOR delay = 1 TO 2300   'kill time while they move
  NEXT delay
  OBJECT.CLOSE            'wipe out both objects
  END

 InitImage:
 'Create the string equivalent
 'of an ObjEdit image file

 FOR x=0 TO 89
 READ d%
 ShipShape$=ShipShape$+CHR$(d%)
 NEXT
 RETURN

 DATA 0, 0, 0, 0 ,0, 0, 0, 0
 DATA 0, 0, 0, 2, 0, 0, 0, 32
```

```
DATA Ø, Ø, Ø, 8, Ø, 24, Ø, 3
DATA Ø, Ø
' Bit Plane Ø
DATA &HØØ, &HØØ, &HØØ, &HØØ
DATA &HØØ, &HØØ, &HØØ, &HØØ
DATA &HØØ, &HØØ, &HØØ, &HØØ
DATA &HØ3, &HC3, &HC3, &HCØ
DATA &HØ3, &HC3, &HC3, &HCØ
DATA &HØØ, &HØØ, &HØØ, &HØØ
DATA &HØØ, &HØØ, &HØØ, &HØØ
DATA &HØØ, &HØØ, &HØØ, &HØØ
'Bit Plane 1
DATA &HØØ, &H3F, &HFC, &HØØ
DATA &HØ3, &HFF, &HFF, &HCØ
DATA &HFF, &HFF, &HFF, &HFF
DATA &HFF, &HFF, &HFF, &HFF
DATA &HFF, &HFF, &HFF, &HFF
DATA &HFF, &HFF, &HFF, &HFF
DATA &HØ3, &HFF, &HFF, &HCØ
DATA &HØØ, &H3F, &HFC, &HØØ
```

Detecting Collisions

When a movable object collides with another object or with one of the borders of the window, Amiga BASIC notes the collision and saves the information about it on a *stack.* This stack can hold information about only 16 collisions at a time. After the stack is full, BASIC ignores any subsequent collisions.

You can receive several kinds of information about collisions from the COLLISION function. The syntax for the various forms of this function is

object_num = COLLISION (0)
collision_window = COLLISION (−1)
collision_code = COLLISION (object_num)

COLLISION (0) gives the object_number of the object that was involved in the collision whose information is the top item on the stack. This form leaves the collision information on the stack, where it can be retrieved by a subsequent call of the third form, COLLISION (object_num).

COLLISION (−1) identifies the window in which the collision recorded on the top item of the stack occurred. It also leaves the collision information on the stack.

COLLISION (object_num) is the most common. It returns a number, collision_code, that identifies what collided with the object in question during the collision recorded by the top entry on the stack. In the process, it also removes the item from the stack to make room for new entries. If you specify the object_num of an object that was not involved in the collision recorded on the top of the stack, the collision_code will be zero, indicating no collision, and you'll have lost the chance to find out what happened in that collision. Therefore, if you're unsure which object was involved in the collision recorded on the top of the stack, check it first with COLLISION (0).

Besides zero, indicating no collision, other possible collision_codes include positive numbers, which correspond to the object_num of another object with which the object collided, and negative numbers, which indicate a collision with one of the window borders. The significance of these negative values is

−1	Object collided with top border
−2	Object collided with left border
−3	Object collided with bottom border
−4	Object collided with right border

There is a way to detect collision other than having your program check the COLLISION function every so often. If you use the ON COLLISION statement, BASIC will notify your program every time that it detects a collision and will cause your program to execute a specified subroutine after the current statement finishes its execution. The format of this statement is

ON COLLISION GOSUB label

where label is the program label for the subroutine that is to be executed. You can change the subroutine that is to be executed at any time by issuing the ON COLLISION GOSUB statement with another label, or you can disable collision trapping with the statement

ON COLLISION GOSUB 0

Like other event-trapping statements, the ON COLLISION statement will not actually direct the program to your subroutine when a collision happens until you give the statement

COLLISION ON

It will, however, still place collision event information in its stack, so when the COLLISION ON statement comes, the program will be directed to the specified subroutine once for each collision event that has been stored. After you have given the COLLISION ON statement, you may suspend event trapping with the statement

COLLISION STOP

which will stop it until the next COLLISION ON statement. To end collision trapping entirely, use the statement

COLLISION OFF

Normally, Amiga BASIC records collisions between every object, and between objects and the window borders. In some cases, however, you may not want to take any action if certain objects collide with each other or with the border. You therefore might not want BASIC to take any notice of these collisions at all. You can prevent the detection of certain collisions with the OBJECT.HIT statement. This statement takes the form

OBJECT.HIT object_num [MeMask] [,HitMask]

where MeMask and HitMask are values whose bit patterns determine which type of object will collide. Think of MeMask as a number that defines the collision type of this object and HitMask as a number that describes the collision type of the object with which this object will collide. If you logically AND the MeMask of one object with the HitMask of another, a collision will be detected only if the result is not zero. In addition, if the HitMask of an object is an odd number (has a one as the least significant bit), it will collide with borders.

For example, let's take the following four objects:

Object_num	MeMask	HitMask	Collides With
1	0010 (2)	1101 (13)	obj2, obj3, borders
2	0100 (4)	1010 (10)	obj1, obj3
3	1000 (8)	0110 (6)	obj1, obj2
4	0010 (2)	0001 (1)	borders only

Object 1 has a HitMask that indicates it collides with all object types except those that have the same MeMask as it does. Its HitMask value is 13, which produces a nonzero result when ANDed with either the MeMask of object 2 (4) or the MeMask of object 3 (8). But 13 and the MeMask of object 4 (2) equals zero. Therefore, object 1 collides with both objects 2 and 3, but not object 4. Since its HitMask is odd, it also collides with borders.

Objects 2 and 3 have HitMasks with each other's MeMask bit set, in addition to that of object 1. They collide with each other and with object 1, therefore, but not with the border, since both are even numbers.

Object 4 has a HitMask with only the least significant bit set. It has zeros in the bit places represented by the MeMasks of all the other objects. Therefore, it collides only with the borders.

In the above example, none of the objects has HitMasks that indicate that they can collide with another object, unless that other object also has a HitMask that indicates it collides with the first. Since the position of the objects will determine whether object 1 collides with object 2, or object 2 collides with object 1, it is a good practice to make sure that the HitMasks of each is set so that both collide with each other or neither collides with each other. Otherwise, the collision of the two objects will not always be reported.

Program 6-10 was adapted from the Demo program which appears on the Extras disk. It demonstrates many of the statements explained in this section. It creates the shape of a flying saucer vsprite from data and changes the data so that a second vsprite is shown in different colors. It moves and accelerates these vsprites, and uses collision trapping to bounce them off the borders. It uses collision masking to make sure that only border collisions are detected, not collisions between the ships. The demonstration stops after a certain number of bounces to make sure that the ships do not reach escape velocity and disappear off the screen.

You may notice that if you move the mouse pointer up and down the screen as the ships are bouncing, the pointer colors will change. This is because the vsprite system uses the available hardware sprites in order, and hardware sprite 1 shares its color registers with sprite 0, the one used for the pointer. So, when the animation system changes the color of sprite 1, it also changes the color of the pointer. One way to get around this problem is to use the GetSprite command to reserve sprite 1, thus making it unavailable to the vsprite system. Try substituting the following lines for the beginning of Program 6-10 (the part before the subroutines):

```
LIBRARY "graphics.library"
DEFINT a-z
WINDOW 1, "Bouncing Space Ships"
DIM Sprite(1Ø)
CALL GetSprite&(VARPTR(Sprite(Ø)),1)
GOSUB initialize 'set up sprites
WHILE Running
   SLEEP   'only do something
           'when the sprites collide
WEND
OBJECT.CLOSE  'release all objects
PALETTE Ø,Ø,.3,.6 'screen back to blue
CALL FreeSprite&(1)
END
```

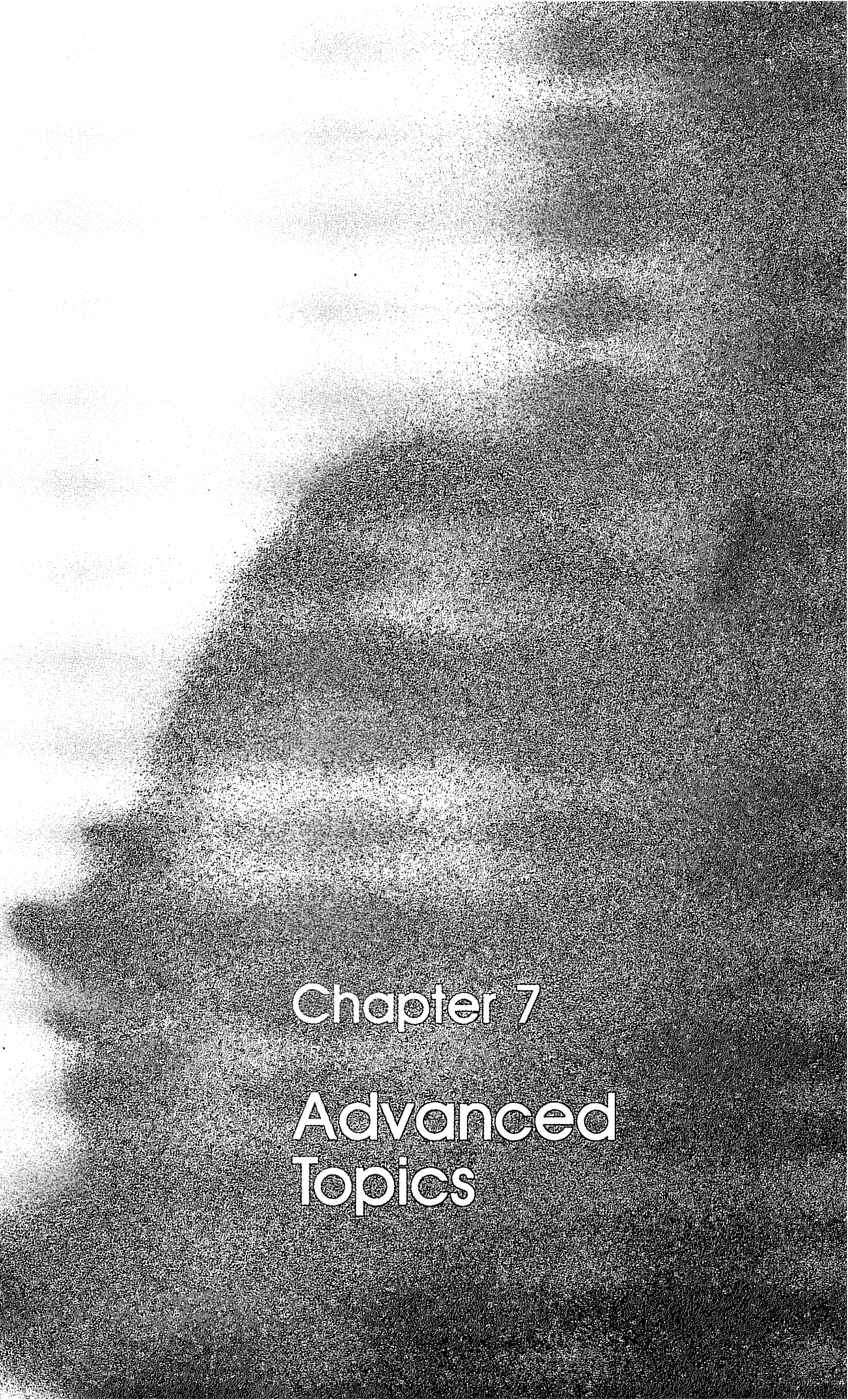
Notice how the mouse pointer no longer changes colors when you move it on top of the spaceships.

Program 6-10. Vsprite Demo

```
DEFINT a-z
WINDOW 1,"Bouncing Space Ships",(Ø,Ø)-(3ØØ,186),4

GOSUB Initialize 'set up sprites
WHILE Running
  SLEEP          'only do something when the sprites collide
WEND

OBJECT.CLOSE          'release all objects
PALETTE Ø,Ø,.3,.6     'screen back to blue
END

Bounce:
 T = T+1:IF T = 25 THEN Running = Ø
 s = COLLISION(Ø)      'which object collided?
 IF s = Ø THEN Exeunt  'none
 c = COLLISION(s)      'what did it collide with?
 vx = OBJECT.VX(s)
 vy = OBJECT.VY(s)
 IF (c=-1 AND vy < Ø) OR (c=-3 AND vy > Ø) THEN
   'object bounced off top or bottom border
   OBJECT.VY s,-vy
 ELSEIF (c=-2 AND vx < Ø) OR (c=-4 AND vx > Ø) THEN
   'object bounced off left or right border
   OBJECT.VX s,-vx
 END IF
 GOTO Bounce
Exeunt:
 OBJECT.START          'make it go again
 RETURN
```

```
Initialize:
  Running = 1
  PALETTE Ø,Ø,Ø,Ø
  GOSUB InitImage
  'Create ShipShape$ from data
  'rather than reading image file

  OBJECT.SHAPE 1, ShipShape$
  OBJECT.Y 1,2Ø
  OBJECT.VX 1,1Ø
  OBJECT.VY 1,7
  OBJECT.AY 1,1
  OBJECT.AX 1,2

  L=LEN(ShipShape$)        'change color data of vsprite
  red1=Ø
  grnblu1=Ø              ' Ø,Ø,Ø = black
  red2=15
  grnblu2 = Ø*16+15      ' 15,Ø,15 = purple
  red3=Ø
  grnblu3=15*16+3        ' Ø, 15, 15 = aqua
  Col$=CHR$(red1)+CHR$(grnblu1)
  Col$=Col$+CHR$(red2)+CHR$(grnblu2)
  Col$=Col$+CHR$(red3)+CHR$(grnblu3)
  MID$(ShipShape$,L-5)=Col$

  OBJECT.SHAPE 2, ShipShape$   'create a saucer with new colors
  OBJECT.Y 2,3Ø
  OBJECT.VX 2,2
  OBJECT.VY 2,2
  OBJECT.AY 2,2
  OBJECT.AX 2,2
```

```
   OBJECT.ON                      'make them visible
   OBJECT.START                   'start them moving
   OBJECT.CLIP (Ø,Ø)-(275,184) 'set borders
   OBJECT.HIT 1,2,1
   OBJECT.HIT 2,2,1               'only collide with borders
   ON COLLISION GOSUB Bounce      'set collision trapping
   COLLISION ON                   'turn it on

   RETURN

InitImage:
FOR x=Ø TO 63
READ d%
ShipShape$=ShipShape$+CHR$(d%)
NEXT
RETURN

DATA Ø, Ø, Ø, Ø, Ø, Ø, Ø, Ø
DATA Ø, Ø, Ø, 2, Ø, Ø, Ø, 16
DATA Ø, Ø, Ø, 8, Ø, 25, Ø, 3
DATA Ø, Ø, Ø, Ø, Ø, Ø, Ø, Ø
DATA 25, 152, 25, 152, Ø, Ø, Ø, Ø
DATA Ø, Ø

'Sprite Image Data
'2 bytes wide by
'8 lines high

DATA &HØ7, &HEØ
DATA &H1F, &HF8
DATA &HFF, &HFF
DATA &HFF, &HFF
DATA &HFF, &HFF
DATA &HFF, &HFF
DATA &H1F, &HF8
DATA &HØ7, &HEØ

 'Sprite colors
 'RGB values are held in two
 'hex bytes--ØR AND GB

DATA &HØØ, &HØØ
DATA &HØF, &HØØ
DATA &HØF, &HFØ
```



Chapter 7

Advanced Topics

Advanced Topics

During our exploration of the graphics capabilities of the Amiga, we have touched on a number of topics, which, because of their complexity, could not be explained fully when they were mentioned. In this chapter, we will be discussing three such topics in greater detail: SUPER_BITMAP windows, Hold and Modify (HAM) display mode, and Extra Halfbrite display mode. None of these graphics modes is directly supported by BASIC, and by nature they don't lend themselves to being implemented in BASIC through the use of POKEs or library calls. This discussion, therefore, will be confined to programming in the C language.

SUPER_BITMAP Windows

In Chapter 2, we discussed the three methods for refreshing windows, which is the term used to describe the process of restoring the contents of a window after it has been covered by another window and then uncovered again, or when it has been moved.

One method is to create what is known as a SUPER_BITMAP window. This involves setting up a separate bitmap storage area for the window. This storage area is large enough to contain the entire contents of the window. Although we normally think of a bitmap as an area of memory that is used to hold data that is actually displayed, the contents of the SUPER_BITMAP are not directly shown onscreen. Rather, the part of the bitmap that represents the area of the window that is currently uncovered is copied to the screen's bitmap. No matter what happens to the copy that is being displayed, the contents of the SUPER_BITMAP remain safe. This means that the programmer never has to worry about refreshing a SUPER_BITMAP window, because Intuition always has a copy of the data available to it.

Another advantage of the SUPER_BITMAP window is that the bitmap area may be any size, up to 1024 × 1024 pixels. This allows the programmer to create a picture that is larger than the window used to display it, or even larger than can fit on the display screen at one time. There are a couple of implications to this. When you try to output graphics to other types of windows, the output is clipped or cut off if it is directed outside the boundaries of the window. With a SUPER_BITMAP window, data directed outside the window's screen boundaries may not appear on the display immediately, but it will be stored in the bitmap if it falls within the boundaries of this larger area. The window may then be scrolled over the contents of the bitmap so as to reveal these hidden portions of the big picture one at a time.

The price that you pay for the advantages of a SUPER_BITMAP window is a little extra programming effort and additional memory usage. The amount of extra memory required may be quite substantial if you create a very large bitmap containing many bit planes.

In order to set up a SUPER_BITMAP window, you must create your own bitmap data structure and storage area, and initialize them before you can open the window. The first step is to declare a BitMap data structure:

```
struct BitMap BitMap;
```

This structure contains information about the display layout of the data, the number of bit planes, and the height and width of each bit plane. To initialize these values, use the Graphics library call InitBitMap. The format for this statement is

```
InitBitMap(BitMap, Depth, Width, Height);
           (a0)    (d0)   (d1)   (d2)
```

where BitMap is a pointer to the BitMap data structure, Depth is the number of bit planes used, and Width and Height give the dimensions of each bit plane in pixels.

The BitMap data structure also contains pointers to the actual memory areas used to store the data for each bit plane. In order to initialize these BitMap.Planes values, you must allocate memory for the bitmap storage. You do this with the AllocRaster memory allocation function that we discussed in Chapter 3. For example, if you have initialized a BitMap data structure for a bitmap that is 640 × 400 pixels and two planes deep, the following code fragment would allocate the required memory areas and link the addresses of these areas into the BitMap structure:

```
for (x=0;x<2;x++)
    if
    ((BitMap.Planes[x] = (PLANEPTR)AllocRaster(640,200)) ==0)
        exit(400);
    else BltClear (BitMap.Planes[x],16000,0);
```

Notice that we also used the BltClear statement, which we discussed in Chapter 3, to initialize the bitmap to all zeros.

Having completed initialization of the BitMap data structure, we must still link that structure into the NewWindow structure before opening the window. We do this by storing a pointer to the BitMap structure in the Bitmap variable of the NewWindow structure, like this:

```
NewWdw.BitMap = (struct BitMap *)&BitMap;
```

Before opening the window, make sure that the Flags variable in the NewWindow structure shows the refresh type as SUPER_BITMAP also.

As we mentioned above, once you have set up the SUPER_BITMAP window, you may position the graphics cursor anywhere within the bitmap and place graphics there, whether that portion of the bitmap is currently being displayed or not. When you wish to scroll the window around the bitmap to reveal the hidden portions of it, you may use the library routine ScrollLayer. Before you use this routine, however, you must first open the Layers library, of which this function is a part. The process of opening this library is very similar to that of opening the Intuition library or Graphics library. First, you declare a structure for storing the base address of the library:

```
struct LayersBase *LayersBase;
```

Next, you open the library, assigning the result to the base address variable. If the value returned is zero, the library is not available, and your program should exit:

```
LayersBase = (struct LayersBase *)
      OpenLibrary("layers.library",LIBRARY_VERSION);
if (LayersBase == NULL) exit(FALSE);
```

Once you have opened the library, you may use the ScrollLayer routine. The syntax for this statement is

```
ScrollLayer (Layer_Info, Layer,  dx, dy);
                (a0)        (a1)   (d0) (d1)
```

where Layer_Info and Layer are pointers to data structures, and Dx and Dy are the horizontal and vertical offsets by which you wish to move the window. The Layer_Info data structure is part of the Screen data structure; in Program 7-1 this value is set to zero, and the Layer_Info structure from our window's screen is used. A pointer to the Layer structure is part of the Window data structure. If the pointer to your window was declared as

```
struct Window *Wdw;
```

then the Layer structure can be referred to by the expression

```
Wdw->WLayer
```

Program 7-1 is a simple example of setting up a bitmap that is larger than the window. It sets up a BitMap area that is as large as the Workbench screen, but opens a SUPER_BITMAP window that is only a quarter of the size of the display. Several lines of text are written to the bitmap, and then the ScrollLayer statement is used to scroll the contents of the bitmap through the window to reveal the hidden parts of the text.

One important point to keep in mind is that in order for a portion of the SUPER_BITMAP to be displayed, its data must be moved to the portion of the screen's bitmap display area that is used by its window. Therefore, if you directly modify the contents of the SUPER_BITMAP itself, it will not alter the screen display of the SUPER_BITMAP window until the window actually needs to be refreshed by Intuition. For example, when the ScrollLayer command is used to change the portion of the bitmap that is displayed, such a refresh takes place.

Program 7-1. SUPER_BITMAP Window

```
/****************************************************
*
*       SuperBit.C
*       Shows how to set up a SuperBitMap window
*       whose bitmap is bigger than the window,
*       and scroll the window around in the bitmap
*
*
****************************************************/


/* Include the definitions we need */

#include <exec/types.h>
#include <intuition/intuition.h>

/* Structures needed for libraries */

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;
struct LayersBase *LayersBase;

/* Structures required for graphics */

struct BitMap BitMap;
struct Window *Wdw;


/******** Program Constants *****************/
```

```
#define Rp   Wdw->RPort   /* to shorten this up */
#define BLUP     Ø
#define WHTP     1
#define BLKP     2
#define ORNP     3

/* Pre-initialized NewScreen and NewWindow Structures */

struct NewWindow NewWdw =
     {
     Ø,Ø,            /* Left Edge, Top Edge */
     32Ø,1ØØ,        /* Width, Height */
     BLUP,WHTP,      /* Block Pen, Detail Pen */
     CLOSEWINDOW,    /* IDCMP Flags */
     SUPER_BITMAP + ACTIVATE
      + GIMMEZEROZERO + WINDOWCLOSE,   /* Flags */
     NULL,           /* Pointer to First Gadget */
     NULL,           /* Pointer to Check Mark image */
     "Scrolling SuperBitMap Window",   /* Title */
     NULL,           /* Pointer to Screen structure, dummy */
     NULL,           /* Pointer to custom Bit Map */
     Ø,Ø,            /* Minimum Width, Height */
     Ø,Ø,            /* Maximum Width, Height */
     WBENCHSCREEN    /* Type of Screen it resides on */
     };

/* ************** Program Begins Here ********** */

main()
{
int x;
```

```
/* Open the Intuition, Graphics and Layers libraries.
 * Gets pointer to WCS routines, and if = Ø,
 * libraries aren't available, so quit.
 */

   IntuitionBase = (struct IntuitionBase *)
         OpenLibrary("intuition.library",LIBRARY_VERSION);
   if (IntuitionBase == NULL) exit(1ØØ);

   GfxBase = (struct GfxBase *)
        OpenLibrary("graphics.library",LIBRARY_VERSION);
   if (GfxBase == NULL) exit(2ØØ);

    LayersBase = (struct LayersBase *)
        OpenLibrary("layers.library",LIBRARY_VERSION);
    if (LayersBase == NULL) exit(3ØØ);

/* Initialize the bitmap and open the  Window.
 *  If the window pointer is Ø, it wasn't opened. */

InitBitMap(&BitMap,2,64Ø,2ØØ);

for (x=Ø;x<2;x++)
   if
   ((BitMap.Planes[x] = (PLANEPTR)AllocRaster(64Ø,2ØØ)) ==Ø)
      exit(4ØØ);
   else BltClear (BitMap.Planes[x],16ØØØ,Ø);


NewWdw.BitMap = (struct BitMap *)&BitMap;

   if (( Wdw = (struct Window *)OpenWindow(&NewWdw)) == Ø)
      exit(5ØØ);
```

```
demo();

Wait(1<<Wdw->UserPort->mp_SigBit);

for (x=Ø;x<2;x++)   /*  free bit-map memory */
     if (BitMap.Planes[x] != Ø)
          FreeRaster(BitMap.Planes[x],64Ø,2ØØ);

/* Close the window and the libraries */

CloseWindow(Wdw);
CloseLibrary(GfxBase);
CloseLibrary(IntuitionBase);
CloseLibrary(LayersBase);

}  /*  end of main */

demo()
{
int line;

SetAPen(Rp,WHTP);

for (line=2Ø; line < 19Ø ; line += 1Ø)
   {
   Move(Rp,1Ø,line);
   Text(Rp,"This text line is in a scrolling super-bitmap window",52);
   }

Scroll (1,1);
Scroll (1,-1);
Scroll (Ø,1);
Scroll (-1,Ø);
Scroll (-1,Ø);
}
```

```
Scroll (x,y)
int x,y;
{
int line;

for (line = Ø; line < 12Ø ; line++)
   {
   ScrollLayer (Ø, Wdw->WLayer, x,y);
   WaitTOF();
   }

} /* End of Superbit.c */
```

Hold and Modify Mode

In Chapter 2, we said that in most circumstances, five is the maximum number of bit planes that a screen can use. In certain special display modes, however, six bit planes may be used at a time. Since high-resolution and interlaced screens cannot use more than four bit planes, these special modes are confined to use in low-resolution, noninterlaced mode only.

The first of these special modes is called Hold and Modify (HAM) mode. Normally, the maximum number of colors that you can display onscreen at once is 32, since there are only 32 hardware color registers available for use. Hold and Modify mode, however, allows you to bypass this limitation and to display all 4096 shade onscreen at once.

The secret is in how the pen number for each dot of the screen display is interpreted. When HAM is active, the highest two bits of the pen number control how the lower four bits of that number are interpreted. When the top two bits are both zeros, the lower four bits are interpreted as a pen number from 0 to 15, as normal. When one or more of these top two bits are set to one, however, the interpretation is very different. In that case, determining the color of the pixel is a two-part process.

First, you start by making the color of the pixel the same as that of the next pixel to the left. Then, you modify one of the three color components of the pixel (red, green, or blue) by changing it to the value of the lower four bits of the pen value. Which color component is changed depends on the value in the high-order two bits. If those bits are set to 01, the blue component is modified. If they are set to 10, the red bits of the pixel to the left are replaced. And if they are set to 11, the green component is the one that is changed.

You should see now why this mode is called Hold and Modify. It extends the color selection by allowing you to copy (Hold) two of the three color values used by the preceding pixel and to change (Modify) the other value to the new value specified by the lower four bits of the pen number. The following list summarizes the actions taken according to the settings of the top two bits of the pen value for each dot:

00*xxxx*	Use the pen value specified by the bits *xxxx* as you would ordinarily. For example, if these bits had the value 1010, it would mean to use the color values specified in pen (color register) 12.
01*xxxx*	Duplicate the red and green values of the pixel to the left, and use the value *xxxx* for the blue color value.
10*xxxx*	Duplicate the green and blue values of the pixel to the left, and use the value *xxxx* for the red color value.
11*xxxx*	Duplicate the red and blue values of the pixel to the left, and use the value *xxxx* for the green color value.

Setting up and using a Hold and Modify screen is fairly straightforward. To activate Hold and Modify mode, set the ViewModes value in the NewScreen data structure to HAM. Do not set the HIRES or LACE flags. Set the Depth variable of that structure to 6, and open the Screen as you normally would. When drawing in a window that appears on that screen, use the SetAPen statement to set the foreground pen to a value of 0 to 15 to draw using the normal pen colors. If you wish to draw using the Hold and Modify method, add 16 to the color intensity desired if you want to hold the red and green values of the pixel to the left, and modify the blue value. Add 32 if you wish to modify the red value, or add 48 if you wish to modify the green value.

Program 7-2 shows how to use the Hold and Modify mode. It draws six strips, each of which is divided into 16 color segments.

Although HAM mode allows you to display a lot of colors at once, you have limited control over the color selection of any one pixel. Since you can change only one color value per pixel, to change colors entirely takes three pixels.

Another limitation is that when you use HAM mode for a number of pixels in a row, the color of each succeeding pixel depends on that of the pixel to the left. Changing the color value of the first pixel in that row could change all of the pixels that appear to its right. This is demonstrated in Program 7-2. Originally, the bottom three strips of color are identical to the top three. All that is required to change them entirely, however, is to draw a single horizontal line in front of each.

Because of these limitations, HAM mode is somewhat difficult to use for purposes such as freehand drawing. On the other hand, the extended color resolution that it affords can be used very successfully in applications such as digitizing color video images. Pictures produced by this method are amazingly faithful to the original, considering that they use a 320 × 200 display resolution.

Extra Halfbrite Mode

Another special display mode that uses six bit planes is called the Extra Halfbrite mode. This mode is not discussed very much in the Amiga technical literature, because it was a very late addition to the Amiga hardware. In fact, many of the first Amigas do not support this mode at all. At the time of this writing, Commodore-Amiga has not announced a policy concerning updates to the new display chip.

Extra Halfbrite represents a compromise between HAM and the normal display mode which allows the user to extend the color selection beyond the normal 32 colors. In Extra Halfbrite mode, the lower five bits of the pen value are used to select a color register from 0 to 31. If the sixth bit is set to one, however, the red, green, and blue values held in that color register are shifted one place to the right. This effectively halves the luminance value for each color. The resulting display, therefore, is a much darker version of the original color.

Setting up and using an Extra Halfbrite screen is similar to the process used for a HAM screen. Set the ViewModes variable in the NewScreen structure to EXTRA_HALFBRITE. Do not set the LACE or HIRES flags. Set the Depth variable to 6. When drawing, add 32 to the pen value to get the Halfbrite equivalent of the color.

The Extra Halfbrite mode offers a bit of extra flexibility in color selection, but it is not really as handy as having 64 distinct color registers. For one thing, since each Halfbrite color uses only three bits for red, green, and blue values, there are only 512 possible Halfbrite colors, one-eighth of the total Amiga color palette. And, as you may have noticed in Program 7-3, the Halfbrite colors are all fairly dark and are less distinct from one another for that reason as well.

Program 7-2. Hold and Modify

```
/* Include the definitions we need */

#include <exec/types.h>
#include <intuition/intuition.h>

/* Structures needed for libraries */

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

/* Structures required for graphics */

struct Screen *CustScr;
struct Window *Wdw;
struct ViewPort *WVP;

/********** Program Constants *********************/

#define Rp   Wdw->RPort   /* to shorten this up */

#define BLACK  Øx000
#define BGRP     Ø

#define MODBLU Øx1Ø
#define MODRED Øx2Ø
#define MODGRN Øx3Ø


/* * Pre-initialized NewScreen and NewWindow Structures * */
```

```
    struct NewScreen NewCustScr =
       {
       Ø,Ø,                 /* LeftEdge (always=Ø),TopEdge */
       32Ø,2ØØ,6,           /* Width, Height, Depth */
       1,Ø,                 /* DetailPen and BlockPen */
       HAM,                 /* special display modes */
       CUSTOMSCREEN,        /* Screen Type */
       NULL,                /* Pointer to Custom font*/
       NULL,                /* Pointer to title text */
       NULL,                /* Pointer to Screen Gadgets */
       NULL,                /* Pointer to CustomBitMap */
       };


struct NewWindow NewWdw =
      {
      Ø,Ø,              /* Left Edge, Top Edge */
      32Ø,2ØØ,          /* Width, Height */
      NULL,NULL,        /* Block Pen, Detail Pen */
      CLOSEWINDOW,      /* IDCMP Flags */
      SMART_REFRESH | ACTIVATE
      | BORDERLESS | WINDOWCLOSE,   /* Flags */
      NULL,             /* Pointer to First Gadget */
      NULL,             /* Pointer to Check Mark image */
      NULL,             /* Title */
      NULL,             /* Pointer to Screen structure, dummy */
      NULL,             /* Pointer to custom Bit Map */
      Ø,Ø,              /* Minimum Width, Height */
      Ø,Ø,              /* Maximum Width, Height */
      CUSTOMSCREEN      /* Type of Screen it resides on */
      };

/* **************** Program Begins Here *************** */

main()
{
```

```
/* Open the Intuition and Graphics libraries.
 * Get pointer to WCS routines,
 * and if = Ø, libraries aren't available.
 */

   IntuitionBase = (struct IntuitionBase *)
         OpenLibrary("intuition.library", LIBRARY_VERSION);
   if (IntuitionBase == NULL) exit(FALSE);

   GfxBase = (struct GfxBase *)
         OpenLibrary("graphics.library", LIBRARY_VERSION);
   if (GfxBase == NULL) exit(FALSE);

/* Open the Screen and Windows.
*  If their pointers = Ø, they weren't opened. */

  if ((NewWdw.Screen = CustScr =
 (struct Screen *)OpenScreen(&NewCustScr)) == NULL)
      exit(FALSE);

  if (( Wdw = (struct Window *)OpenWindow(&NewWdw)) == NULL)
      exit(FALSE);

 /* find the viewport and load color map*/

WVP = (struct ViewPort *)ViewPortAddress(Wdw);
SetRGB4(WVP,Ø,Ø,Ø,Ø);     /*set background to black */
SetRGB4(WVP,1,Ø,8,8);     /*set color 1 */
SetRGB4(WVP,2,1Ø,Ø,4);    /*set color 2 */
SetRGB4(WVP,3,3,9,Ø);     /*set color 2 */

demo();

Wait(1<<Wdw->UserPort->mp_SigBit);
```

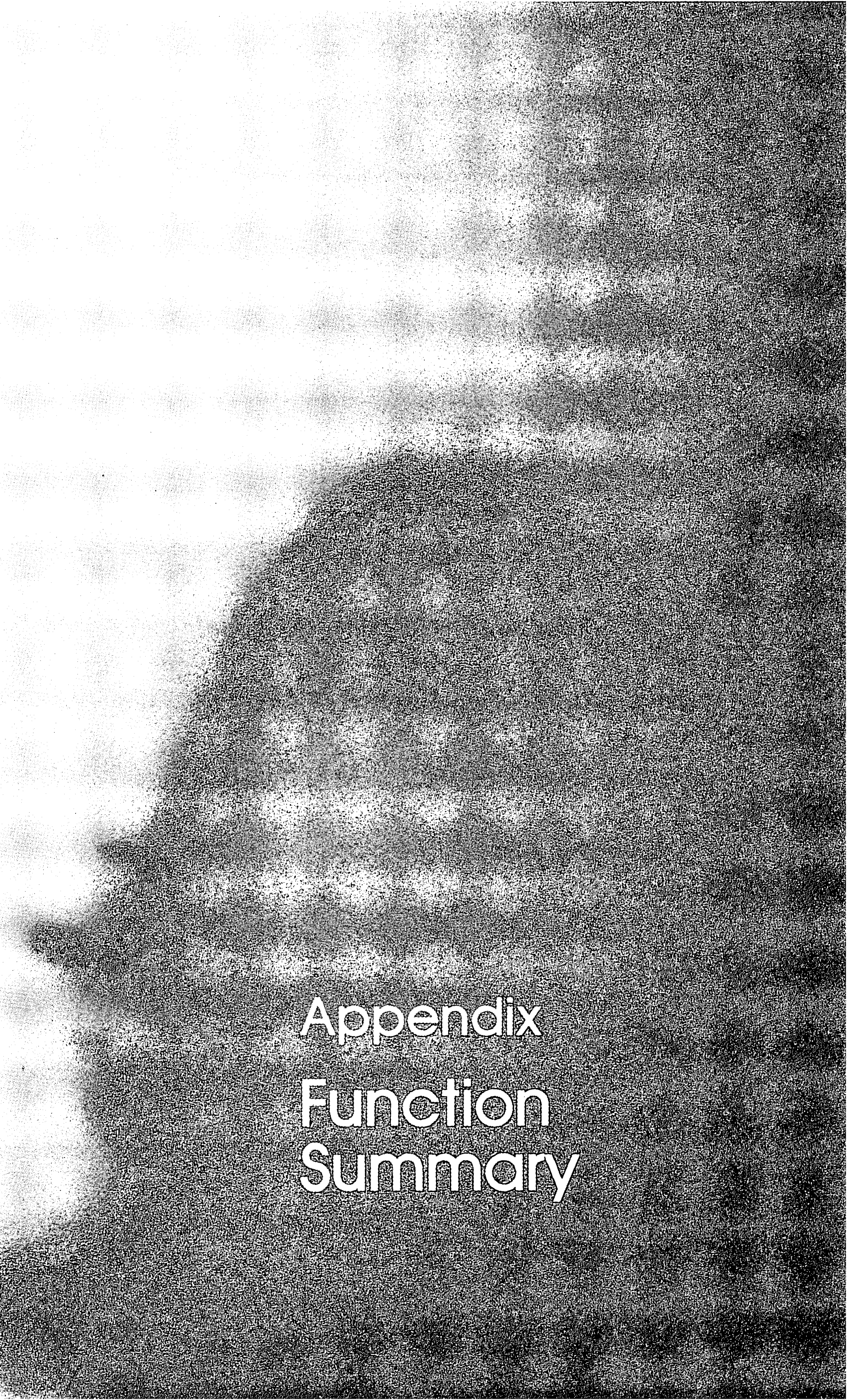
```
   CloseWindow(Wdw);
   CloseScreen(CustScr);
   CloseLibrary(GfxBase);
   CloseLibrary(IntuitionBase);
}

demo()
{
int c;

for (c=1;c<16;c++)
   {
   SetAPen(Rp,MODRED+c);
   RectFill(Rp,18*c,2Ø,18*c+17,4Ø);
   RectFill(Rp,18*c,11Ø,18*c+17,13Ø);

   SetAPen(Rp,MODGRN+c);
   RectFill(Rp,18*c,5Ø,18*c+17,7Ø);
   RectFill(Rp,18*c,14Ø,18*c+17,16Ø);

   SetAPen(Rp,MODBLU+c);
   RectFill(Rp,18*c,8Ø,18*c+17,1ØØ);
   RectFill(Rp,18*c,17Ø,18*c+17,19Ø);
   }

for (c=1;c<4;c++)
   {
   SetAPen(Rp,c);
   Move(Rp,18,3Ø*c+8Ø);
   Draw(Rp,18,3Ø*c+1ØØ);
   }

}
```

Program 7-3. Extra Halfbrite Mode

```
/* Include the definitions we need */

#include <exec/types.h>
#include <intuition/intuition.h>

/* Structures needed for libraries */

struct IntuitionBase *IntuitionBase;
struct GfxBase *GfxBase;

/* Structures required for graphics */

struct Screen *CustScr;
struct Window *Wdw;
struct ViewPort *WVP;

/********** Program Constants ***********************/

#define Rp    Wdw->RPort      /* to shorten this up */

UWORD colormap [32] =         /* colors for pen Ø-31 */
{
  Øx888, ØxFØØ, ØxØFØ, ØxØØF,
  ØxFFØ, ØxØFF, ØxFØF, ØxFFF,
  ØxAØ4, ØxØA4, ØxA4Ø, Øx4ØA,
  Øx4AØ, ØxØ4a, ØxAAØ, ØxAØA,
  Øx953, Øx369, Øx84A, ØxC3B,
  Øx8E4, Øx5C8, Øx678, Øx983,
  Øx77A, Øx9B4, ØxD4D, ØxCF7,
  Øx3B9, Øx95E, ØxDD5, Øx4F7
};
```

```
/* * Pre-initialized NewScreen and NewWindow Structures * */

   struct NewScreen NewCustScr =
     {
     Ø,Ø,                  /* LeftEdge (always=Ø),TopEdge */
     32Ø,2ØØ,6,            /* Width, Height, Depth */
     1,Ø,                  /* DetailPen and BlockPen */
     EXTRA_HALFBRITE,      /* special display modes */
     CUSTOMSCREEN,         /* Screen Type */
     NULL,                 /* Pointer to Custom font*/
     NULL,                 /* Pointer to title text */
     NULL,                 /* Pointer to Screen Gadgets */
     NULL,                 /* Pointer to CustomBitMap */
     };


struct NewWindow NewWdw =
     {
     Ø,Ø,             /* Left Edge, Top Edge */
     32Ø,2ØØ,         /* Width, Height */
     NULL,NULL,       /* Block Pen, Detail Pen */
     CLOSEWINDOW,     /* IDCMP Flags */
     SMART_REFRESH | ACTIVATE
      | BORDERLESS | WINDOWCLOSE,    /* Flags */
     NULL,            /* Pointer to First Gadget */
     NULL,            /* Pointer to Check Mark image */
     NULL,            /* Title */
     NULL,            /* Pointer to Screen structure, dummy */
     NULL,            /* Pointer to custom Bit Map */
     Ø,Ø,             /* Minimum Width, Height */
     Ø,Ø,             /* Maximum Width, Height */
     CUSTOMSCREEN     /* Type of Screen it resides on */
     };
```

```
/* ************* Program Begins Here ************** */

main()
{

/* Open the Intuition and Graphics libraries.
 * Get pointer to WCS routines,
 * and if = Ø, libraries aren't available.
 */

   IntuitionBase = (struct IntuitionBase *)
         OpenLibrary("intuition.library", LIBRARY_VERSION);
   if (IntuitionBase == NULL) exit(FALSE);

   GfxBase = (struct GfxBase *)
         OpenLibrary("graphics.library", LIBRARY_VERSION);
   if (GfxBase == NULL) exit(FALSE);

/* Open the Screen and Windows.
*  If their pointers = Ø, they weren't opened. */

  if ((NewWdw.Screen = CustScr =
 (struct Screen *)OpenScreen(&NewCustScr)) == NULL)
      exit(FALSE);

  if (( Wdw = (struct Window *)OpenWindow(&NewWdw)) == NULL)
      exit(FALSE);

 /* find the viewport and load color map*/

WVP = (struct ViewPort *)ViewPortAddress(Wdw);
LoadRGB4(WVP,&colormap,32);    /*load color registers */
demo();

Wait(1<<Wdw->UserPort->mp_SigBit);

     CloseWindow(Wdw);
     CloseScreen(CustScr);
     CloseLibrary(GfxBase);
     CloseLibrary(IntuitionBase);
}

demo()
{
int c;

for (c=Ø;c<16;c++)
   {
   SetAPen(Rp,c);    /* color bars for colors Ø-15 */
   RectFill(Rp,18*c,2Ø,18*c+17,4Ø);

   SetAPen(Rp,c+32); /* and the halfbrite versions */
   RectFill(Rp,18*c,5Ø,18*c+17,7Ø);

   SetAPen(Rp,c+16);    /* color bars for colors 16-31 */
   RectFill(Rp,18*c,11Ø,18*c+17,13Ø);

   SetAPen(Rp,c+48);  /* and the halfbrite versions */
   RectFill(Rp,18*c,14Ø,18*c+17,16Ø);

   }


}/* end HalfBrite.c */
```



Appendix Function Summary

Function Summary

AddBob

Location	graphics library
Function	Adds a bob to the current GEL list
Syntax	AddBob(Bob, RastPort); (a0) (a1)
Input Parameters	Bob = Pointer to the Bob data structure to be added to the list RastPort = Pointer to the RastPort structure to which the GEL list is linked

AddVSprite

Location	graphics library
Function	Adds a vsprite to the current GEL list
Syntax	AddVSprite(VSprite, RastPort); (a0) (a1)
Input Parameters	VSprite = Pointer to the VSPrite data structure to be added to the list RastPort = Pointer to the RastPort structure to which the GEL list is linked

AllocRaster

Location	graphics library
Function	Allocates free memory from the heap for use as display memory
Syntax	Raster = AllocRaster(Width, Height); (d0) (d0) (d1)

Input Parameters	Width = Width of the bit plane in pixels Height = Height of the bit plane in lines
Results	If the function is able to allocate the memory requested, Raster will contain a pointer to the memory area. If not, it will contain a zero.

AreaDraw

Location	graphics library
Function	Adds a point to the list of points used to fill an area
Syntax	Error = AreaDraw(RastPort, X, Y); (a1) (d0) (d1)
Input Parameters	RastPort = Pointer to the RastPort data structure X = Horizontal coordinate of the point Y = Vertical coordinate of the point
Results	Error is set to −1 if there is no room left in the list of points; otherwise, 0.

AreaEnd

Location	graphics library
Function	Fills an area using a list of vertices
Syntax	AreaEnd(RastPort); (a1)
Input Parameters	RastPort = Pointer to the RastPort data structure

AreaMove

Location	graphics library
Function	Closes the current polygon described by a list of points and defines the starting point for a new polygon

Syntax	Error = AreaMove(RastPort, X, Y); (a1) (d0) (d1)
Input Parameters	RastPort = Pointer to the RastPort data structure X = Horizontal coordinate of the point Y = Vertical coordinate of the point
Results	Error is set to −1 if there is no room left in the list of points; otherwise, 0.

AskFont

Location	graphics library
Function	Moves the text attributes of the current font to a TextAttr data structure
Syntax	AskFont(RastPort, TextAttr); (a1) (a0)
Input Parameters	RastPort = Pointer to the RastPort data structure TextAttr = Pointer to the TextAttr data structure to be filled

AskSoftStyle

Location	graphics library
Function	Returns the style bits for the font styles that can be generated by the operating system software for the current font. The value returned may be used as the Enable mask for SetSoftStyle.
Syntax	Enable = AskSoftStyle(RastPort); (d0) (a1)
Input Parameters	RastPort = Pointer to the RastPort data structure
Results	The valid style bits are returned in Enable.

AvailFonts

Location — diskfont library

Function — Builds an array of information on all of the fonts available on disk or in memory or both

Syntax

```
Bytes_short = AvailFonts(Buf_ptr, Buf_size,
   (d0)                    (a0)      (d0)
Type);
 (d1)
```

Input Parameters — Buf_ptr = Pointer to the memory buffer used to hold the array
Buf_size = Size of the buffer in bytes
Type = The type of font to search for. AFF_MEMORY is used to search for fonts in memory, and AFF_DISK is set to search for disk-residents fonts. Both can be used.

Results — If the buffer does not have enough room to contain all of the font information, Bytes_short contains the number of additional bytes that must be added to the buffer size so that it can contain all of the font information.

BltClear

Location — graphics library

Function — Fills a section of chip memory with zero bytes

Syntax

```
BltClear(Memory, Bytes, Flags);
           (a1)   (d0)   (d1)
```

Input Parameters — Memory = Pointer to memory to clear (must start on a word boundary)
Bytes = Amount of memory to clear (usually an even number of bytes)
Flags = Set bit 0 to force the function to wait until the memory is cleared before resuming. Bit 1 is used to determine if Bytes is interpreted as an even number of bytes (0) or as a number of rows and bytes per row to clear.

BltPattern

Location	graphics library
Function	Draws through as stencil, using the standard drawing rules
Syntax	BltPattern(RastPort, Pattern, X1, Y1, X2, Y2, Width); RastPort (a1), Pattern (a0), X1 (d0), Y1 (d1), X2 (d2), Y2 (d3), Width (d4)
Input Parameters	RastPort = Pointer to the RastPort data structure Pattern = Pointer to the two-dimensional stencil pattern X1 = Horizontal coordinate for the upper left corner of the destination in the RastPort Y1 = Vertical coordinate for the upper left corner of the destination in the RastPort X2 = Horizontal coordinate for the lower right corner of the destination in the RastPort Y2 = Vertical coordinate for the lower right corner of the destination in the RastPort Width = Width of the two-dimensional stencil pattern in bytes

BNDRYOFF

Location	include/graphics/gfxmacros.h (graphics macro)
Function	Turns off outlining of filled figures
Syntax	BNDRYOFF(RastPort);
Input Parameters	RastPort = Pointer to the RastPort data structure

ChangeSprite

Location	graphics library
Function	Links a table of sprite shape data to a SimpleSprite data structure, thus determining the shape of that simple sprite
Syntax	ChangeSprite (ViewPort, SimpleSprite, Sprite_data); ViewPort (a0), SimpleSprite (a1), Sprite_data (a2)
Input Parameters	ViewPort = Pointer to the ViewPort data structure (a zero may be used if the sprite is positioned relative to the view) SimpleSprite = Pointer to an initialized SimpleSprite data structure Sprite_data = Pointer to a table of sprite shape data

ClearPointer

Location	intuition library
Function	Returns the Intuition mouse pointer to its default shape
Syntax	ClearPointer(Window); Window (a0)
Input Parameters	Window = Pointer to the Window data structure

ClipBlit

Location	graphics library
Function	Transfers (and possibly manipulates) bitmap data from a rectangular area in one rastport to another rastport or to a different portion of the same rastport

Syntax

```
ClipBlit
(SrcRp, SrcX, SrcY, DestRp, DestX, DestY,
 (a0)   (d0)  (d1)  (a1)    (d2)   (d3)
Width, Height, Minterm);
 (d4)   (d5)    (d6)
```

Input Parameters

SrcRp = Pointer to the source RastPort data structure
SrcX = Horizontal coordinate for the upper left corner of the source rectangle
SrcY = Vertical coordinate for the upper left corner of the source rectangle
DestRp = Pointer to the destination RastPort data structure
DestX = Horizontal coordinate for the upper left corner of the destination rectangle
DestY = Vertical coordinate for the upper left corner of the destination rectangle
Width = Width of the rectangle (in bits)
Height = Height of the rectangle (in lines)
Minterm = The blitter logic minterm used to transfer and/or manipulate the graphics image data

CloseFont

Location

graphics library

Function

Indicates to the system that a font opened with the OpenFont call is no longer in use

Syntax

```
CloseFont(FontPtr);
          (a1)
```

Input Parameters

FontPtr = Pointer to a font descriptor (obtained from OpenFont call)

CloseLibrary

Location

exec library

Function

Indicates to the system that a library opened with OpenFont call is no longer in use

Syntax	CloseLibrary(LibraryBase); (a1)
Input Parameters	LibraryBase = Pointer to the base address of the library (obtained from OpenLibrary call)

CloseWindow

Location	intuition library
Function	Closes an Intuition window, unlinks it from the system, and deallocates its memory
Syntax	CloseWindow(Window); (a0)
Input Parameters	Window = Pointer to the Window data structure

CloseWorkBench

Location	intuition library
Function	Attempts to close the Workbench screen
Syntax	Results = CloseWorkBench( ); (d0)
Results	If any applications have opened windows on the Workbench screen, it can't be closed, and Results will be set to false (0). If the screen was closed, Results is set to true (1).

Draw

Location	graphics library
Function	Draws a line from the current pen position to a specified point, using the current pens, line pattern, and drawing mode
Syntax	Draw(RastPort, X, Y); (a1) (d0)(d1)

Input Parameters

RastPort = Pointer to the RastPort data structure
X = Horizontal position of the line's endpoint
Y = Vertical position of the line's endpoint

DrawGList

Location

graphics library

Function

Processes the GEL list, drawing bobs and constructing a copper list for vsprites

Syntax

```
DrawGList(RastPort, ViewPort);
            (a1)      (a0)
```

Input Parameters

RastPort = Pointer to the RastPort data structure
ViewPort = Pointer to the ViewPort data structure

DrawImage

Location

intuition library

Function

Draws an Intuition image into the rastport

Syntax

```
DrawImage (RastPort, Image, LeftOffset,
             (a0)     (a1)     (d1)
TopOffset);
  (d2)
```

Input Parameters

RastPort = Pointer to the RastPort data structure
Image = Pointer to the Image data structure
LeftOffset = Horizontal placement offset for the image
TopOffset = Vertical placement offset for the image

FreeRaster

Location	graphics library
Function	Releases graphics memory back to the system memory pool
Syntax	FreeRaster(Raster, Width, Height); (a0) (d0) (d1)
Input Parameters	Raster = Pointer to the beginning of memory allocation (obtained from AllocRaster call) Width = Width of the bit plane in bits (must be the same value used in AllocRaster) Height = Height of the bit plane in lines (must be the same as used in AllocRaster)

FreeSprite

Location	graphics library
Function	Deallocates a hardware sprite so that your application no longer has exclusive use of that sprite
Syntax	FreeSprite (Sprite_number); (d0)
Input Parameters	Sprite_number = The number of the hardware sprite to be released

GetSprite

Location	graphics library
Function	Reserves a hardware sprite for your exclusive use

Syntax	Sprite_got = GetSprite (SimpleSprite,
	(d0) (a0)
	Sprite_number);
	(d0)

Input Parameters

SimpleSprite = A pointer to the SimpleSprite structure to be used with the hardware sprite that is allocated.
Sprite_number = The number of the hardware sprite (0–7) that you are requesting. If you wish to use the first available sprite, pass a value of −1, and read Sprite_got to find out which sprite was reserved.

Results

Sprite_got contains the number of the hardware sprite (0–7) that was actually allocated. If none could be allocated, its value is −1.

InitBitMap

Location

graphics library

Function

Initializes a BitMap data structure to default values

Syntax

```
InitBitMap(BitMap, Depth, Width, Height);
             (a0)   (d0)   (d1)    (d2)
```

Input Parameters

BitMap = Pointer to the BitMap data structure
Depth = Number of bit planes to be used
Width = Width of each bit plane (in bits)
Height = Height of each bit plane (in lines)

InitGels

Location

graphics library

Function

Initializes a GEL list

Syntax

```
InitGels(VSprite1, VSprite2, GelsInfo);
           (a0)      (a1)      (a2)
```

Input Parameters

VSprite1 = Pointer to a dummy VSprite data structure to be used as the head of the GEL list
VSprite2 = Pointer to a dummy VSprite data structure to be used as the tail of the GEL list
GelsInfo = Pointer to the GelsInfo data structure to be initialized

InitMasks

Location	graphics library
Function	Initializes the BorderLine and CollMask values used by the VSprite data structure
Syntax	InitMasks(VSprite); (a0)
Input Parameters	VSprite = Pointer to the VSprite data structure

LoadRGB4

Location	graphics library
Function	Loads a list of color register values from a data table
Syntax	LoadRGB4(ViewPort, Colormap, Pens); (a0) (a1) (d0)
Input Parameters	ViewPort = Pointer to the ViewPort data structure. Colormap = Pointer to the table of color values for the registers. This table is arranged as an array of 16-bit data words, where the first nybble is zero, the second contains the red color value, the third the green, and the fourth the blue. Pens = The number of consecutive color registers to load, starting with register 0.

LoadView

Location	graphics library
Function	Creates a display using a new copper list
Syntax	LoadView(View); (a1)
Input Parameters	View = Pointer to the View data structure

Move

Location	graphics library
Function	Moves the drawing pen from its current location to the specified position without drawing anything
Syntax	Move (RastPort, X, Y); (a1) (d0) (d1)

Input Parameters	RastPort = Pointer to the RastPort data structure
	X = New horizontal coordinate
	Y = New vertical coordinate

MoveScreen

Location	intuition library
Function	Drags an Intuition screen up or down the display
Syntax	MoveScreen (Screen, DeltaX, DeltaY); (a0) (d0) (d1)
Input Parameters	Screen = Pointer to the Screen data structure DeltaX = Offset by which to move the screen horizontally (ignored by current version of Intuition) DeltaY = Offset by which to move the screen vertically (current version of Intuition requires the bottom of the screen to stay at or below the bottom of the display)

MoveSprite

Location	graphics library
Function	Changes the display position of a simple sprite to the specified location
Syntax	MoveSprite(ViewPort, SimpleSprite, X, Y); (a0) (a1) (d0) (d1)

Input Parameters

ViewPort = Pointer to the ViewPort data structure
SimpleSprite = Pointer to the SimpleSprite data structure
X = New horizontal position of the sprite
Y = New vertical position of the sprite

MoveWindow

Location

intuition library

Function

Moves an Intuition window under software control

Syntax

```
MoveWindow(Window, DeltaX, DeltaY);
             (a0)    (d0)    (d1)
```

Input Parameters

Window = Pointer to the Window data structure
DeltaX = Horizontal offset by which to move the window
DeltaY = Vertical offset by which to move the window

MrgCop

Location

graphics library

Function

Merges together coprocessor instructions to form one instruction list

Syntax

```
MrgCop(View);
       (a1)
```

Input Parameters

View = Pointer to the View data structure

OpenDiskFont

Location

diskfont library

Function

Obtains a pointer to the font descriptor for a disk-resident font and indicates that the font is being used by your application

Syntax

```
FontPtr = OpenDiskFont(TextAttr);
  (d0)                    (a0)
```

Input Parameters

TextAttr = Pointer to a TextAttr data structure that describes the font you wish to open

OpenFont

Location

graphics library

Function

Obtains a pointer to the font descriptor for a memory-resident font and indicates that the font is being used by your application

Syntax

```
FontPtr = OpenFont(TextAttr);
  (d0)                (a0)
```

Input Parameters

TextAttr = Pointer to a TextAttr data structure that describes the font you wish to open

OpenLibrary

Location

exec library

Function

Obtains a pointer to the base address of a library and indicates that the library is being used by your application

Syntax

```
Library_base_address =
          (d0)
OpenLibrary("name.library",Version);
                 (a1)        (d0)
```

Input Parameters

"name.library" = Pointer to a string of ASCII characters that names the library. This name must be in the form "name.library", all in lowercase letters, ending with an ASCII 0 (for example, "graphics.library").

Version = Version number of the library. OpenLibrary will successfully open the library only if the version of the library that the system is using is greater than or equal to this number.

Results — If the library can be successfully opened, its base address is returned in Library_base_address. If not, a value of 0 is returned.

OpenScreen

Location — intuition library

Function — Sets up a Screen data structure and displays the Intuition screen described by a NewScreen data structure

Syntax

```
Screen = OpenScreen(NewScreen);
 (d0)                  (a0)
```

Input Parameters — NewScreen = Pointer to the NewScreen data structure that describes the characteristics of the screen.

Results — If the screen is successfully opened, Screen contains a pointer to the Screen data structure. If not, Screen is set to zero.

OpenWindow

Location — intuition library

Function — Sets up a Window data structure and displays the Intuition window described by a NewWindow data structure

Syntax

```
Window = OpenWindow(NewWindow);
 (d0)                  (a0)
```

Input Parameters — NewWindow = Pointer to the NewWindow data structure that describes the characteristics of the window.

Results — If the window is successfully opened, Window contains a pointer to the Window data structure. If not, Window is set to zero.

OpenWorkBench

Location	intuition library
Function	Attempts to open the Workbench screen
Syntax	Result = OpenWorkBench( ); (d0)
Results	If the screen is open, Result is set to true (1). If not, it is set to false (0).

PolyDraw

Location	graphics library
Function	Draws a series of connected lines from the current pen position to the points specified by a table of *(x,y)* coordinate pairs, using the current drawing modes, pen colors, and line pattern.
Syntax	PolyDraw(RastPort, Coordinate_pairs, (a1) (d0) Array_address); (a0)
Input Parameters	RastPort = Pointer to the RastPort data structure Coordinate_pairs = Number of coordinate pairs in the data table Array_address = Pointer to the data table of coordinate pairs

ReadPixel

Location	graphics library
Function	Finds the color register (pen) used to color the point at a specified location on the display
Syntax	Pen = ReadPixel(RastPort, X, Y); (d0) (a1) (d0) (d1)

Input Parameters	RastPort = Pointer to the RastPort data structure X = Horizontal coordinate for the point Y = Vertical coordinate for the point
Results	If the point lies within the boundaries of the rastport, Pen is set to the pen number used to color the point. If not, Pen is set to −1.

RemIBob

Location	graphics library
Function	Removes a bob from the GEL list and erases it from the rastport display
Syntax	RemIBob(Bob, RastPort); (a0) (a1)

Input Parameters	Bob = Pointer to the Bob data structure
	RastPort = Pointer to the RastPort data structure

ScreenToBack

Location	intuition library
Function	Moves the specified screen to the back of the display
Syntax	ScreenToBack(Screen); (a0)
Input Parameters	Screen = Pointer to the Screen data structure

ScreenToFront

Location	intuition library
Function	Moves the specified screen to the front of the display

Syntax

```
ScreenToFront(Screen);
              (a0)
```

Input Parameters

Screen = Pointer to the Screen data structure

ScrollLayer

Location

layers library

Function

Copies data to a layer from a SuperBitMap so as to reposition the display over the bitmap

Syntax

```
ScrollLayer(Layer_Info, Layer, DeltaX, DeltaY);
            (a0)        (a1)   (d0)    (d1)
```

Input Parameters

Layer_Info = Pointer to the Layer_Info data structure
Layer = Pointer to the Layer data structure
DeltaX = Horizontal offset by which to move the layer
DeltaY = Vertical offset by which to move the layer

ScrollRaster

Location

graphics library

Function

Scrolls the contents of a rectangular area of a rastport

Syntax

```
ScrollRaster(RastPort, Dx,  Dy,  X1,  Y1,
             (a1)      (d0) (d1) (d2) (d3)
X2,  Y2);
(d4) (d5)
```

Input Parameters

RastPort = Pointer to the RastPort data structure
Dx = Horizontal offset by which to scroll the rectangle
Dy = Vertical offset by which to scroll the rectangle
X1 = Horizontal position of the left edge of the rectangle
X2 = Horizontal position of the right edge of the rectangle
Y1 = Vertical position of the top edge of the rectangle
Y2 = Vertical position of the bottom edge of the rectangle

SetAFPt

Location

include/graphics/gfxmacros.h (graphics macro)

Function

Sets a pattern for area fills

Syntax

SetAFPt(RastPort, Pattern, Size);

Input Parameters

RastPort = Pointer to the RastPort data structure
Pattern = Pointer to a table of 16-bit pattern data words
Size = Number of data words in pattern (must equal a power of 2)

SetAPen

Location

graphics library

Function

Sets the color register used by the foreground drawing pen

Syntax

```
SetAPen(RastPort, Pen);
          (a1)    (d0)
```

Input Parameters

RastPort = Pointer to the RastPort data structure
Pen = Color register used for the pen

SetBPen

Location	graphics library
Function	Sets the color register used for the background drawing pen
Syntax	SetBPen(RastPort, Pen); (a1) (d0)
Input Parameters	RastPort = Pointer to the RastPort data structure Pen = Color register used for the pen

SetDrMd

Location	graphics library
Function	Sets a drawing mode for drawing routines
Syntax	SetDrMd(RastPort, Mode); (a1) (d0)

Input Parameters	RastPort = Pointer to the RastPort data structure
	Mode = Drawing mode (JAM1, JAM2, COMPLEMENT, INVERSVID)

SetDrPt

Location	include/graphics/gfxmacros.h (graphics macro)
Function	Sets a pattern to use for line drawing
Syntax	SetDrPt(RastPort, Pattern);

Input Parameters	RastPort = Pointer to the RastPort data structure
	Pattern = A 16-bit drawing pattern

SetFont

Location graphics library

Function Sets the font to be used for drawing text in a rastport

Syntax
```
SetFont(RastPort, FontPtr);
             (a1)       (a0)
```

Input Parameters RastPort = Pointer to the RastPort data structure
FontPtr = Pointer to a font descriptor (obtained from OpenFont)

SetOPen

Location include/graphics/gfxmacros.h (graphics macro)

Function Sets the pen used for outlining filled figures and turns on outlining

Syntax
```
SetOPen(RastPort, Pen);
```

Input Parameters RastPort = Pointer to the RastPort data structure
Pen = Color register used for the pen

SetPointer

Location intuition library

Function Sets the shape of the Intuition mouse pointer in a window

Syntax
```
SetPointer
 (Window, Sprite_data, Height, Width, XOffset,
    (a0)        (a1)       (d0)    (d1)     (d2)
YOffset);
   (d3)
```

Input Parameters

Window = Pointer to the Window data structure
Sprite_data = Pointer to a table of sprite shape data
Height = Height of the pointer sprite in lines
Width = Width of the sprite in pixels (must be less than or equal to 16)
XOffset = Horizontal offset of the hot spot
YOffset = Vertical offset of the hot spot

SetRast

Location

graphics library

Function

Sets the entire RastPort to a specified color

Syntax

```
SetRast(RastPort, Pen);
             (a1)    (d0)
```

Input Parameters

RastPort = Pointer to the RastPort data structure
Pen = Color register used to color the rastport

SetRGB4

Location

graphics library

Function

Sets the red, green, and blue color values for a color register

Syntax

```
SetRGB4(ViewPort, Pen, Red, Green, Blue)
          (a0)     (d0) (d1)  (d2)   (d3)
```

Input Parameters

ViewPort = Pointer to the ViewPort data structure
Pen = Color register to set
Red = Red color level (0–15)
Green = Green color level (0–15)
Blue = Blue color level (0–15)

SetSoftStyle

Location graphics library

Function Sets the software-generated font style

Syntax
```
Result = SetSoftStyle(RastPort, Style, Enable);
 (d0)                    (a1)     (d0)   (d1)
```

Input Parameters RastPort = Pointer to the RastPort data structure
Style = The software-generated style requested
Enable = A mask that determines which style bit can be changed; can be derived from AskSoftStyle

Results The resulting style is returned in Result.

ShowTitle

Location intuition library

Function Determines whether the screen title bar will be displayed in front of a backdrop window or not

Syntax
```
ShowTitle(Screen, ShowIt);
           (a0)    (d0)
```

Input Parameters Screen = Pointer to the Screen data structure.
ShowIt = A flag that indicates whether or not to display the title bar in front of a backdrop window. A value of true (1) means show the title bar, while a value of false (0) means hide it.

SizeWindow

Location intuition library

Function Changes the size of an Intuition window under program control

Syntax	SizeWindow(Window, DeltaX, DeltaY); (a0) (d0) (d1)
Input Parameters	Window = Pointer to the Window data structure DeltaX = Change to the width of the window DeltaY = Change to the height of the window

SortGList

Location	graphics library
Function	Sorts the GEL list by vertical position of each element, prior to displaying the GELs
Syntax	SortGList(RastPort); (a1)
Input Parameters	RastPort = Pointer to the RastPort data structure

Text

Location	graphics library
Function	Draws text in a rastport using the current font
Syntax	Text(RastPort,Text_string,Chars); (a1) (a0) (d0)
Input Parameters	RastPort = Pointer to the RastPort data structure Text_string = Pointer to a string of ASCII characters Chars = Number of characters to print

TextLength

Location	graphics library
Function	Finds the length (in bits) that a string of characters would occupy if printed to a rastport using the current text font

Syntax

```
Length = TextLength(RastPort,Text_string,
 (d0)                        (a1)     (a0)
Chars);
 (d0)
```

Input Parameters

RastPort = Pointer to the RastPort data structure
Text_string = Pointer to a string of ASCII characters
Chars = Number of characters to be printed

ViewPortAddress

Location

intuition library

Function

Finds the address of a window's viewport

Syntax

```
ViewPort = ViewPortAddress(Window);
  (d0)                      (a0)
```

Input Parameters

Window = Pointer to the Window data structure

Results

The address of the viewport is returned in viewport

WBenchToBack

Location

intuition library

Function

Moves the Workbench screen to the back of the display

Syntax

```
Results = WBenchToBack( );
  (d0)
```

Results

If the Workbench was opened, Results is set to true (1). If not, it is set to false (0).

WBenchToFront

Location	intuition library
Function	Moves the Workbench screen to the front of the display
Syntax	Results = WBenchToFront( ); (d0)
Results	If the Workbench was opened, Results is set to true (1). If not, it is set to false (0).

WindowLimits

Location	intuition library
Function	Sets new limits to which a window may be sized

Syntax	status = WindowLimits(Window, MinWidth,
	(d0) (a0) (d0)
	MinHeight, MaxWidth, MaxHeight);
	(d1) (d2) (d3)

Input Parameters	Window = Pointer to the Window data structure MinWidth = New minimum width of the window (in pixels) MinHeight = New minimum height of the window (in lines) MaxWidth = New maximum width of the window (in pixels) MaxHeight = New maximum height of the window (in lines)

WindowToBack

Location	intuition library
Function	Moves the specified window to the back of the display
Syntax	WindowToBack(Window); (a0)
Input Parameters	Window = Pointer to the Window data structure

WindowToFront

Location	intuition library
Function	Moves the specified window to the front of the display
Syntax	WindowToFront(Window); (a0)
Input Parameters	Window = Pointer to the Window data structure

WritePixel

Location	graphics library
Function	Colors a single pixel with the current foreground drawing pen
Syntax	Result = WritePixel(RastPort, X, Y); (d0) (a1) (d0) (d1)
Input Parameters	RastPort = Pointer to the RastPort data structure X = Horizontal position of the dot Y = Vertical position of the dot

Index

Amiga Graphics

The Amiga has the most advanced graphics capabilities of any consumer computer. And *Inside Amiga Graphics* is the comprehensive guide to the power of Amiga graphics. Whether you program in BASIC, C, or machine language, you'll find here a wealth of information you can use to add impressive graphics to your own programs.

Here's a sample of what you'll find inside:

- Clear explanations of how to access and use Intuition routines.
- How to create and open custom screens.
- Loading and changing text fonts from BASIC and C language.
- Using the graphics.library from BASIC and C.
- Drawing and manipulating image blocks.
- How to create and use sprites and bobs.
- Dozens of program examples you can learn from and use.

Written clearly and concisely, *Inside Amiga Graphics* is *the* guide to programming graphics on the Amiga. Whether you want to set up a display screen or program sprites—in BASIC or C—this book has the information you need to take your Amiga to its limits.

ISBN 0-87455-040-8

Inside Amiga Graphics

COMPUTE! Books