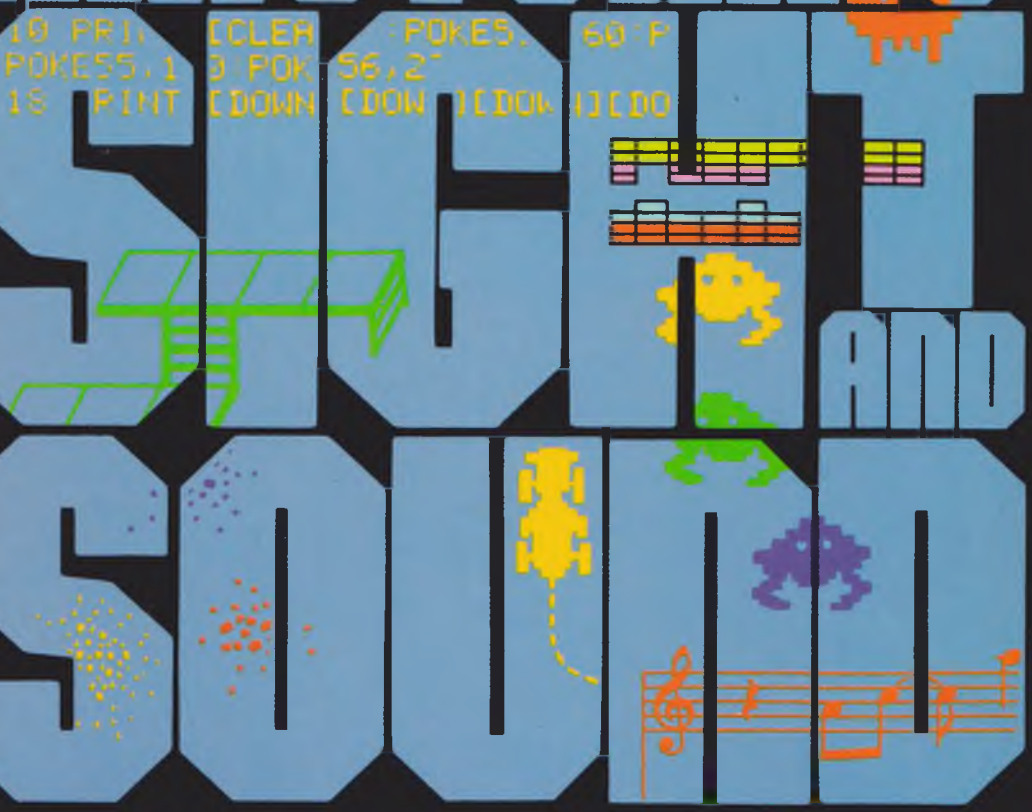
PLUME

MASTERING SIGHT AND SOUND



ON THE ON THE ON THE ON THE

COMMODORE®

64™

HOW TO TAP THE FULL CREATIVE POWER AND PLEASURE OF THE HOME COMPUTER THAT'S TOPS FOR GRAPHICS AND MUSIC

BY KENT PORTER

AUTHOR OF THE NEW AMERICAN COMPUTER DICTIONARY

MAKE YOUR COMMODORE SING AND THE PICTURES DANCE

Ever since humans lived in caves, and perhaps even before, they have stimulated their senses and fulfilled their creative urges by shaping color, form and sound to their desires.

Through the thousands of years since, both professional and amateur artists have used a multitude of media for this purpose.

Now, in the age of the computer, we have a new and exciting way to extend the boundaries of our powers to create–

MASTERING SIGHT AND SOUND ON THE COMMODORE® 64™

KENT PORTER worked for the National Security Administration and was a data processing manager for a major bank before becoming a full-time writer. Among his acclaimed books on computers are *Computers Made Really Simple*, *The New American Computer Dictionary* (available in a Signet edition), and *Beginning With Basic* (available in a Plume edition). Mr. Porter lives in Monterey, California.

MASTERING SIGHT AND SOUND ON THE COMMODORE® 64™

by Kent Porter

A PLUME BOOK
NEW AMERICAN LIBRARY
NEW YORK AND SCARBOROUGH, ONTARIO

Appendixes A and B in this book originally appeared, in a somewhat different form, as Appendixes E and F of *Commodore 64 User's Guide,* published by Commodore Business Machines, Inc. Used by permission of Commodore Electronics Limited.

SIGNET, SIGNET CLASSIC, MENTOR, PLUME, MERIDIAN AND NAL BOOKS are published *in the United States* by New American Library, 1633 Broadway, New York, New York 10019, *in Canada* by The New American Library of Canada Limited, 81 Mack Avenue, Scarborough, Ontario M1L 1M8

Library of Congress Cataloging in Publication Data:

Porter, Kent.
Mastering sight and sound on the Commodore 64.

Includes index.
1. Commodore 64 (Computer) I. Title.
QA76.8.C64P67 1984 001.64'2 84-4900
ISBN 0-452-25490-6

First Printing, June, 1984

1 2 3 4 5 6 7 8 9

PRINTED IN THE UNITED STATES OF AMERICA

TABLE OF CONTENTS

MASTERING SIGHT AND SOUND ON THE COMMODORE® 64™

CHAPTER 1
PRELIMINARIES

The Commodore® 64™ is one of the most remarkable machines ever introduced, a full-function microcomputer (micro) with many powerful features and yet a low price. The Commodore 64 gets its name from its manufacturer (Commodore Business Machines, Inc.) and the fact that it has 64K (65,536 bytes) of memory, which is as much memory as an 8-bit computer can work with at one time and probably more than you'll ever need. It does all the things that we have come to expect of personal computers, and includes some features not found in even the largest mainframe machines or the upper-end micros that sell at many times the price.

The Commodore 64 can be used for business because it does what computers are supposed to do: "number crunching," manipulating data, processing words, and whatnot. The added features, action graphics and a music synthesizer, however, are more for hobbyists and game fans than for accountants and scientists. Thus, while the Commodore 64 can be used for business and professional purposes, in this book I'll focus on the aspects that make it a machine for fun. There are plenty of other sources that can give you ideas for more generalized (and certainly useful) things to do with a computer. Having spent several hundred dollars for one, you probably have some ideas of your own. Here I'll talk about using the graphics and sound capabilities that uniquely qualify the Commodore 64 as an outlet for almost unlimited creativity (yours).

There are also other books that will teach you BASIC, the "mother tongue" of the Commodore 64. An excellent introduction to programming

is *Beginning with BASIC: An Introduction to Computer Programming* (NAL/Plume, 1984), which I also wrote. In the present book, I assume that you already have some familiarity with the language, and consequently I don't introduce fundamentals of programming.

The user's guide that came with your Commodore 64 contains several useful chapters that deal with setting up the machine, operating it, and programming. I recommend that, if you have not done so, you read the first four chapters of the user's guide and work the machine exercises. They should give you enough of a basis for understanding this book.

Throughout the book there are many experiments and programs that illustrate the points made in the text. You should actually enter them into your computer and see their effects, since observation and doing are much more vivid than descriptions and they will teach you, rather than simply tell you, how to do the things that make your 64 such a powerful machine. This book was written on the assumption that you are reading with a powered-on Commodore 64 in front of you.

Direct memory alteration

Most of the Commodore 64's special features are controlled by direct alteration of reserved locations in the computer's memory. Consequently, it's important to become comfortable with the manipulation of memory through the BASIC statements POKE and PEEK and their various options.

General organization of memory. The memory of all computers is organized into a series of consecutively numbered cells called addresses. The lowest address is 0 and the highest is one less than the number of bytes ("cells") in the memory. The Commodore 64 has 64K of memory, or 65,536 bytes, so the range of addresses in its memory is 0 through 65535.

Though not essential to an understanding of the Commodore 64, it is perhaps academically useful to digress long enough to explore this term *K*, which crops up often in computer literature. Sixty-odd thousand memory locations is a lot, and we tend to think of large numbers of anything in terms of thousands. In the binary numbering system, which I'll discuss presently, the nearest round number to 1000 is 1024. Computer people have thus coined the term K, meaning "about a thousand" and specifically with the value 1024, so that 64K is $64 \times 1024 = 65{,}536$. Similarly, 4K is $4 \times 1024 = 4096$. To convert some number of K into an actual amount, multiply that number by 1024. Note, incidentally, that a fractional value is almost never expressed in K; a number such as 8.5K is valid but rare.

At any rate, each location within the memory of the computer has an address between 0 and 65535. The number is as fixed as a house's address, so that address 53281 is an unchanging, known point within memory.

The Commodore 64 memory locations each hold one byte of information. A *byte* consists of eight bits, where a bit is a pulse (represented by the digit 1) or a no-pulse (represented by 0). The meaning of a byte is determined by the pattern of its bits, so that 00110011 has a different meaning from 01010010, just as 3 is distinguishable from R by its pattern. The difference is that a computer readily sees patterns of 1s and 0s, whereas the eye sees meaning in the shapes of letters, numbers, and symbols. The concept, however, is the same.

Ordinarily, as computer users, we don't care how information appears inside the machine. That's why we have keyboards and video displays and printers, to handle the translation between visual shapes and the internal representations of data. As programmers of graphics and sounds, however, we have to create special patterns of 1s and 0s that have no corresponding written symbols, but that convey to the computer information about colors, shapes, sounds, locations, waveforms, and other requisites of the Commodore 64's special features. Much of the rest of this chapter has to do with bit patterns.

Although the Commodore 64 has 64K of memory, or in theory enough memory locations to hold about 40 pages of this book at one time, all of the addresses are not available for general use. The special features of the machine "own" blocks of memory for their own needs, and features such as sound and screen colors are controlled by writing bytes into predetermined memory locations.

As an example, turn on your computer. The screen comes up with some start-up information written in light blue against a dark blue background. Hold down the CTRL key and type 2 to make the lettering on the screen white (which does not affect letters already displayed). Now type the instruction

```
POKE 53281,9
```

(*Note:* After every keyboard entry, you must press the RETURN button to tell the computer the entry is complete.) As you see, the screen turns brown. Now type

```
POKE 53280,0
```

and the border turns black. The POKEs wrote values corresponding to colors (9 = brown, 0 = black) into the memory addresses (53281 and 53280) that control the colors of the screen background and the border, respectively. Since this combination is easier to read than the original

blue-on-blue that the folks at Commodore decided you needed, you might find it worthwhile to follow this process each time you start up your computer.

Leave the computer turned on, because there are more experiments in this chapter.

The anatomy of a byte. The smallest unit of information is a *bit*, which is comparable to a digit in a number. A bit—short for binary digit—can be either 1 or 0 in a computer. This corresponds to a pulse or the absence of a pulse in an electrical circuit. When a bit is 1, it is said to be "on"; when 0, "off."

A byte consists of eight bits and is the smallest unit of information a computer works with. One memory location holds one byte. You can think of a byte as being similar to a written character, such as a letter or a digit or a punctuation mark. As I mentioned earlier, a computer discerns meaning by observing the pattern of 1s and 0s in a byte, just as we see meaning in the shapes of visual symbols.

There is, in fact, a direct one-for-one correlation between bytes and alphanumeric characters that has been established as a standard by the electronics industry. It is called the ASCII code (pronounced "askey" and meaning American Standard Code for Information Interchange). The Commodore 64 uses this code, which states that the letter "A" is represented by the bit pattern 01000001, "a" is 01100001, and all other printable symbols and certain commonly used control functions have preestablished bit patterns within an 8-bit byte.

The analogy between alphabets and codes breaks down, however, when we begin to consider possibilities and limitations. There is no limit to the number of shapes we can create on paper to convey meaning to each other, giving us the Latinic alphabet visible everywhere in this book, the Cyrillic alphabets used by assorted Slavic nations, Sanskrit, Old English and Old German, and Chinese ideographs, to name but a few. In contrast, an 8-bit byte can only have so many possible combinations of 1s and 0s before we use them all up. This finite number happens to come to 256, for reasons that will become apparent shortly.

Just as alphabets are systems of symbols that a group of people agree upon as having prearranged meanings, so are codes such as ASCII, which is why 01000001 always means "A." In creating these systems and the machinery that surrounds them, the computer industry has developed a nomenclature to describe them, and since we're going to work with bits and bytes, we might as well use the proper terminology; hence, the following discussion.

First of all, notice that a byte is written as one group of 1s and 0s just like an ordinary number. And like an ordinary number, the value of each place increases from right to left, a convention known as place-value

Bit number	⟶	7	6	5	4	3	2	1	0
Byte	⟶	0	1	0	0	1	0	1	1
		(MSB)							(LSB)

Figure 1.1 Identification of bits by place number

significance. In the number 1024, the 1 has a greater place-value significance than the 4, even though it has a lower numeric value. In the byte 01111111, the 0 has the greatest place-value significance for the same reason. A decimal number such as 1024, however, can have any number of digits, so that 1 and 10 and 102 are all valid numbers, whereas a byte *always* has eight bits. That's why we wrote 01111111 and not simply seven 1s. In decimal numbers it's fine to write 01024, but others would wonder why we put a meaningless 0 on the front. When writing bytes we have to put on the leading 0 so that others will know the complete pattern of the byte.

The leftmost bit, then, is called the Most Significant Bit (or MSB) and the rightmost is the Least Significant Bit (LSB). To further identify bits, they are numbered in their order of significance, as shown in Fig 1.1. (If you have a background in IBM mainframes, note that in micros the order is opposite what you're used to.)

In decimal numbering, each place value right to left increases by ten times, since our numbering system is based on tens. The binary numbering system is based on twos, in which each digit has one of two possible values (0 or 1), and so each place value right to left increases by two times. Figure 1.2 shows the places values of each bit.

To determine the decimal value of a byte, add the place values of each 1 bit. In the case of Fig. 1.2, bits 6, 3, 1, and 0 are all on, so the value of this byte in decimal is

Bit	*Value*
6	64
3	8
1	2
0	1
	75

Bit number	⟶	7	6	5	4	3	2	1	0
Place value	⟶	128	64	32	16	8	4	2	1
Byte 01001011	⟶	0	1	0	0	1	0	1	1

Figure 1.2 Place value of each bit in a byte

You can use this method to determine the decimal value of any byte. As another example, 10110100 has the decimal value 180, which is the sum of its 1-bit place values 128, 32, 16, and 4.

A two-digit decimal number can have 100 different combinations of digits, because the highest two-digit number is 99 and we could also have the number 00, for a total of 100 combinations. Similarly, an 8-bit byte has the lowest value of 00000000 and the highest possible value of 11111111. Using the method above to find the decimal equivalent, the byte 11111111 has a value of 255. This is, then, the highest number we can represent with one byte, and 255 plus one for the byte 00000000 means there are 256 possible combinations of 1s and 0s in eight bits.

Byte operations. You can get the computer to tell you the decimal equivalent of any memory location with the PEEK instruction. Type the command

```
PRINT  PEEK(31024)
```

The computer will immediately display a number between 0 and 255 that represents the decimal equivalent of the bit pattern stored at memory address 31024. Experiment with this statement, changing the address in the parentheses to other values between 0 and 65535. Note that the computer always displays a number less than 256, since that is the highest value that one byte can hold. Just for practice figure out the bit patterns of some of these numbers.

Creating a bit pattern is nearly the same as figuring one out. As an example, suppose you want to create the byte 01001011. If you write it down, then all you have to do is follow the procedure for determining the decimal value. In this case the value is 75, as we already learned from Fig. 1.2.

You can write bit patterns directly into memory locations with the BASIC instruction POKE, which has the general form

```
POKE address, value
```

The address has to be a number between 0 and 65535, since that's the range of addresses in memory, and the value has to be between 0 and 255 because that's how many possible combinations of 1s and 0s there are in a byte.

Earlier we used this instruction to change the screen's background color by entering

```
POKE  53281,9
```

In this case, 53281 is the address that the Commodore 64 looks at to find out what color to display in the background, and 9 is the byte 00001001 that tells the machine to make it brown.

The Commodore 64 can display any of 16 colors in both the background and the border. These colors are specified by POKEing a byte 00000000 through 00001111 (decimal 0 through 15) into memory addresses 53281 and 53280. Try several combinations with your machine just to get the idea. I'll discuss colors and their numbers at much greater length later.

As one last piece of nomenclature, a byte is often regarded as consisting of two 4-bit groupings called "nibbles" (Commodore spells it "nybbles" instead). Since the colors are identified by the low-order nibble (note that the highest color number is 00001111, and the lowest is 00000000), only the lower four bits matter, while the higher-order nibble remains 0000. This term will become important later, especially in music where the nibbles of certain bytes control different aspects of a sound.

Manipulating individual bits. You can use POKE and PEEK to change one or more bits in a byte stored at a memory location, or to move a byte from one place to another, or both.

PEEK, as you may have noticed, gets the byte from a specified address, while POKE places a byte there. POKE is thus a verb meaning "put into so-and-so address the following byte." PEEK, on the other hand, fetches the byte from a specified address, but you have to tell the computer what to do with it once it's fetched it. PEEK is therefore a *function*, a predefined action that produces a numeric result that must then be disposed of. This means that you can't use PEEK as a self-contained command, because the computer won't know what to do with the value it gets by PEEKing into the memory location. The following are valid uses of PEEK:

```
PRINT  PEEK(53281)      Displays byte at 53281
A = PEEK(53281)         Assigns byte at 53281 to A
```

It is *not* valid to tell the computer simply to

```
PEEK(53281)
```

because there is no destination for the value read from 53281.

For a practical application of PEEK and POKE, let's say you want to make the border around the screen the same color as the background. To do this, enter the statement

```
POKE  53280,  PEEK(53281)
```

This means "get the byte from location 53281 and put it into address 53280." Nothing is done to change the byte, so it is copied "as is" from one location to the other. Since 53281 holds the screen background color and 53280 the border color, when you hit RETURN the border changes to the same color as the background.

But now let's suppose that for some (strange) reason you want the border to be black if the color of the background is an even number and white if it's odd. The byte for black is 00000000, represented by the number 0, while for white it's 00000001, or decimal 1.

At this point, we need to make an observation about binary numbers. For this we'll count from 1 to 6 in binary.

```
00000001    1
00000010    2
00000011    3
00000100    4
00000101    5
00000110    6
```

Notice that the LSB is a 1 when the number is odd (1, 3, 5) and 0 in even numbers (2, 4, 6). This is true for the entire range of a byte's value, so we can always tell if a number is odd or even by whether or not the LSB is on.

Black is the lowest even number (0), while white is the lowest odd number (1). Since we want a black border for an even-numbered background and a white border for one that is odd, all we have to do is isolate the least significant bit in address 53281 and copy it to 53280.

You can isolate a bit (or a set of bits) with the AND operator in BASIC. AND is a logic operation not to be confused with addition; the processes are similar but not the same. The truth table for AND is shown in Fig. 1.3. The outcome of AND is that if the bits in corresponding positions in two bytes are both 1, the resulting byte will have a 1 bit in that position; otherwise, it will have a 0 bit. Consequently, the following instruction isolates the LSB in the background color and copies it to the border color, producing the desired result:

```
POKE 53280, PEEK(53281) AND 1
```

To see how this works, let's say the background color is light green (13). PEEK fetches the byte 00001101, which is then ANDed with 1 as follows:

```
Light green    0 0 0 0 1 1 0 1
AND 1          0 0 0 0 0 0 0 1

Result         0 0 0 0 0 0 0 1
```

```
0  AND  0    yields    0
0  AND  1    yields    0
1  AND  0    yields    0
1  AND  1    yields    1
```

Figure 1.3 Truth table for AND operation

The result is binary 1, or white, which is POKEd into the border color address satisfying the requirement that an odd-numbered color have a white border.

You can try this by keying the following sequence of commands into your computer:

```
POKE 53281, 13                (Lt. green background)
POKE 53280, PEEK(53281)       (Lt. green border)
POKE 53280, PEEK(53281) AND 1
```

After the second statement, the border is light green, but the last instruction changes it to white by copying the LSB from the background color byte into the border color byte.

Now suppose instead that the background color in 53281 is number 6, blue. You can force this by typing

```
POKE 53281, 6
```

and the background will turn blue. The color number is even, so if you type

```
POKE 53280, PEEK(53281) AND 1
```

the border turns black (color number 0). Here's what happens:

```
Blue      0 0 0 0 0 1 1 0
AND 1     0 0 0 0 0 0 0 1
Result    0 0 0 0 0 0 0 0
```

Because there are no 1 bits in corresponding positions in the ANDed bytes, the resulting byte is all 0s.

The AND operator does not have to be used only with an individual bit; it can set or reset several bits at the same time. Let's suppose that you want to make certain the screen color is in the range 0–7 (see Fig. 3.9 for the Commodore 64's colors and their values). In all values above 7, bit 3 is on, so to force the screen to the desired range of colors we can simply turn this bit off and leave the others on. As an example, let's make the screen medium gray by typing

```
POKE 53281, 12
```

which places the color byte 0000 1100 into address 53281.

To turn off bit 3 but leave the others the same, we can AND with a "mask" containing 1 bits for the bit positions we want to keep and 0s for those we don't. We want to retain the three lowest order bits, so our mask is the byte 0000 0111. The effect of ANDing is

```
Original byte   0 0 0 0  1 1 0 0
AND mask        0 0 0 0  0 1 1 1
Result          0 0 0 0  0 1 0 0
```

The value of the AND mask in decimal is 7 (4 + 2 + 1) and the result of the AND goes back into the same memory location, so we can code this operation in BASIC as

```
POKE 53281, PEEK(53281) AND 7
```

Try it as an experiment. Your screen should have a medium gray background already. Now type the above POKE command and the screen will turn purple. This is because the computer has masked out bit 3 and left only the primary color number 4. The idea is that the 1 bits of the original byte can only "fall through" 1 bits in the ANDing byte, so 0 bits in the ANDing byte mask out their corresponding positions in the original byte. The mask thus acts as a selective filter.

Sometimes, instead of turning off bits, we want to make sure a certain bit or set of bits is turned on (set to 1). To reverse the case just worked, let's say we want to force the screen to a color whose number is greater than 7. We don't care which color, as long as it's in that range. This means bit 3 must be turned *on* regardless of the settings of the other bits.

For this application, we use the OR operator. As with AND, it has a truth table (Fig. 1.4) showing the outcomes of all combinations. Stated in plain English, OR yields a 1 bit if either of the bits it evaluates is a 1; if neither is a 1, it yields a 0 bit.

As a result of our last experiment, the screen is now a color whose value is less than 8 (purple, color 4). To force it to a number above 7, we have to turn on bit 3 with the statement

```
POKE 53281, PEEK(53281) OR 8
```

This can be paraphrased as "we don't care what the other bits are at address 53181, but we want bit 3 turned on." The machine effect of the instruction is

```
Original byte    0 0 0 0  0 1 0 0
OR 8             0 0 0 0  1 0 0 0

Result           0 0 0 0  1 1 0 0
```

This creates the value 12, which is medium gray, and immediately changes the screen background color.

Now enter the same instruction again and see what happens: nothing. This is because 1 OR 1 produces the same result as 1 OR 0. Consequently

```
0  OR  0    yields    0
0  OR  1    yields    1
1  OR  0    yields    1
1  OR  1    yields    1
```

Figure 1.4 Truth table for OR operation

the resulting byte is the same as the original. OR merely turns on a bit if it is not already on.

PEEK/POKE versus BASIC variables

The PEEK/POKE instructions and the fetching and saving of BASIC variables are similar operations, but they differ in one fundamental respect. PEEK and POKE involve the examination and alteration, respectively, of specific memory locations. BASIC variables are memory locations, too, but they are nonspecific.

BASIC maintains a pool of memory locations as workspace, and much of its activity has to do with management of this memory. When you issue the statement

```
C = PEEK(53281)
```

BASIC fetches the value stored at 53281 and assigns a memory address somewhere in its pool, noting in a table that C refers to that address. It places the value read from 53281 at the assigned location, which might be anywhere; the programmer has no control over where BASIC puts variables within its pool. The only way to retrieve the value associated with C is to refer to it by its variable name. BASIC then looks up C in the table to find it. Thus, BASIC variables are useful for saving and recalling information, but not for directly altering memory locations.

Conversely, POKE and PEEK are used to alter memory, but they are not good for saving and fetching information. For one thing, POKE/PEEK only work with values in the range 0–255. For another, it's dangerous to go about indiscriminately POKEing values into memory locations. This is an excellent way to alter control sections of memory or variables, causing the computer to go haywire and produce bizarre results. You can PEEK wherever you want without causing harm, but you should only POKE into locations whose purposes are known and fixed. These locations will be discussed in great detail in succeeding chapters.

You can exploit the fundamental differences between variables and fixed locations to advantage in your programs. As an example, suppose you have a combination of border and background colors that you want to save and later restore after going through some other combinations. You can save the present values with the instructions

```
BG = PEEK(53281)
BO = PEEK(53280)
```

which holds the background color in the BASIC integer variables BG and the border color in BO. Now you can alter the screen by POKEing other numbers into the control addresses. No matter how many times you change the screen colors, you can always restore them to their original

state with

```
POKE 53281, BG
POKE 53280, BO
```

It's possible to expand this idea for saving and restoring complex setups. For example, the music synthesizer depends chiefly on 25 memory locations beginning at address 54272. If you have a chord you play frequently in a composition, you can set it up with 25 POKEs the first time and then, while it's playing, execute the following loop:

```
400     FOR P = 0 to 24
410       C1(P) = PEEK(54272 + P)
420     NEXT P
```

This loop assumes that you have previously issued the statement DIM C1(25) to tell BASIC to allocate space for the array. Later, each time you want to play this chord, instead of going to all the trouble of setting it back up with POKEs, you can write

```
800     FOR P = 0 to 24
810       POKE(54272 + P), C1(P)
820     NEXT P
```

This is obviously a great deal simpler than repeating 25 POKEs.

Now that we've reviewed the rudiments of direct memory alteration essential to controlling the special features of the Commodore 64, let's start using them to have a little fun.

CHAPTER 2

ALL ABOUT BOUNCING BALLS

Turn on your Commodore 64 and set up the screen for a comfortable color combination. We're about to launch bouncing balls and with them our first journey into computer action graphics.

Whenever you start entering a program on the 64, type the word NEW. The machine then responds READY, indicating that it has cleared away any other program lines currently in memory. Enter the program in Fig. 2.1.

```
100 REM  ** BOUNCING BALL #1
110 PRINT CHR$(147)
120 POKE 53280, 0: POKE 53281, 2
130 X = 1: Y = 1: XV = 1: YV = 1
140 P = X + (Y * 40)
150 POKE (P + 1024), 81
160 POKE (P + 55296), 1
170 FOR T = 1 TO 10: NEXT T
180 POKE (P + 55296), 2
190 X = X + XV: Y = Y + YV
200 IF X <= 0 OR X >= 39 THEN XV = -XV
210 IF Y <= 0 OR Y >= 24 THEN YV = -YV
220 GOTO 140
RUN
```

Figure 2.1 BOUNCING BALL #1

This program sends a ball bouncing aimlessly about the screen, ricocheting off the "walls" of the border. It's an endless program that will bounce the ball all week unless you press the RUN/STOP button. If you want to restart it, type RUN again.

It's a very simple program, and as we go along we'll get fancier and also find ways to keep the action figure—the ball in this case—from flickering as it does here. But though simple and a little crude, BOUNCING BALL #1 embodies some important features of graphics programming on the Commodore 64. Let's dissect it line by line in Fig. 2.2, and then I'll discuss graphics programming in more detail.

Line	*What it does*
100	A REMark line that identifies the program name.
110	Clears the screen.
120	Sets the screen border to black and the background to red.
130	Initializes the position and vector (direction) values of the bouncing ball. X and Y are the horizontal and vertical positions, respectively; XV and YV are the number of distance units to move horizontally and vertically, and the sign of each (plus or minus) indicates direction.
140	Calculates the screen position of the X and Y coordinates.
150	Places the ball (character 81) into screen memory at the position corresponding to the current X and Y.
160	Sets color #1 (white) in color memory at the position coinciding with the ball's location in screen memory. This makes the ball white.
170	A "do-nothing" loop that creates a delay long enough to see the ball. This is called a *timing loop*.
180	Resets the color memory location corresponding to the ball's position to the same color as the background, thus making the ball vanish.
190	Calculates the next position of the ball by adding a horizontal vector unit (XV) to the X coordinate and a vertical vector unit (YV) to the Y coordinate.
200	Checks to see whether the ball has collided with one of the walls. If it has, it reverses the sign of the vector so that subsequently the ball will move in the opposite horizontal direction.
210	Checks to find out if the ball has hit the floor or the ceiling and, if it has, changes the sign of the vertical vector to reverse direction up or down.
220	Repeats the process from line 140.

Figure 2.2 Blow-by-blow of BOUNCING BALL #1

Screen coordinates

In low-resolution graphics (or character) mode, which this program uses, the screen of the Commodore 64 consists of 25 rows of 40 columns each, for a total of 1000 positions. You can think of the screen as a grid in which

the horizontal positions are numbered 0 through 39 and the lines (rows) are numbered 0 through 24 (Fig. 2.3). In BOUNCING BALL #1, as in most of the graphics programs in this book (and in general), the column is called X and the row is called Y. Thus you can describe any point on the screen with two numbers: when X = 0 and Y = 0, the position is the upper left corner of the screen; when X = 39 and Y = 24, the lower right. All other points are somewhere in between. Mathematicians should note that, contrary to the usual convention in Cartesian geometry, the Y coordinates (rows) are numbered downward, so that the zero Y is at the top of the screen and the highest Y is at the bottom.

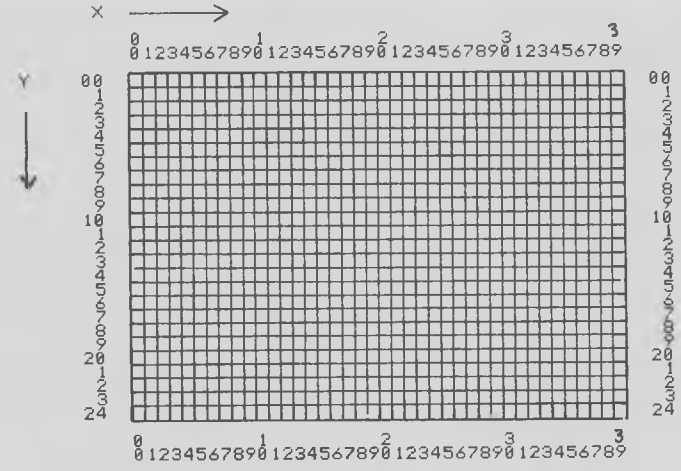


Figure 2.3 Screen layout for low-resolution graphics (or character) mode.

Screen and color memories

To produce a display, the Commodore 64 uses two areas of memory called the *screen memory* and the *color memory*, whose names suggest their purposes. The screen memory is 1000 locations starting at address 1024 and containing, in each address, a byte representing the visual character that appears at the corresponding location on the screen. The color memory is also 1000 locations. It starts at address 55296 and contains, for each position on the screen, a number that indicates the color of the character at that position.

Consequently, if we want the character "*" to appear in blue at the second column of the top row, we place an asterisk in screen memory location 1025 and color #6 (blue) in color memory at address 55297. When

it comes time to display that position on the screen, the Commodore 64 looks at the screen memory to get the character and at the corresponding color memory address to find the color, and then it puts a blue asterisk on the screen.

The Commodore 64 can have several screen memories (though only one is visible at any given moment) and switch around among them. There is only one color memory, however, and you may not use memory locations 55296 through 56295 for anything but color.

Calculating the screen position

Line 140 in BOUNCING BALL #1 contains a calculation that will soon become familiar. It uses the column coordinate (X) and the row coordinate (Y) to compute the corresponding relative address in the screen and color memories.

For example, if the screen position is the upper left corner, X = 0 and Y = 0, so the relative memory address is zero [0 + (0 * 40) = 0]. Line 150 adds this relative address to the start of screen memory (1024) to put character 81, which is the ball, into memory at location 1024 + 0 = 1024, and that is the memory address for the upper left corner of the screen. On the other hand, if the ball is to appear at the start of the second line (Y = 1), the position P is 0 + (40 * 1) = 40, so line 150 computes the screen memory address as 1024 + 40 = 1064. This makes sense since the screen consists of 40 column lines, which are strung end to end in memory.

Similarly, to find the corresponding address in color memory, line 160 adds the relative address P to the start of color memory at 55296. This computation saves a lot of agony and errors by having the computer figure out the actual addresses. As programmers, we need only to remember the starting points of the screen and color memories. For any X and Y coordinates on the screen, the computer will offset properly from the address we give it through calculations such as those in lines 150, 160, and 180.

Timing loops

Line 170 in BOUNCING BALL #1 contains a timing loop, a programming trick also sometimes called a *delay loop*. Its only purpose is to stop the action momentarily by distracting the computer. In this case, the timing loop keeps the ball visible for a little longer than it would be if there were no delay between the time the ball is displayed and the time it is erased.

The use of timing loops is somewhat discretionary. They do nothing productive but instead create a state of suspended animation, thus "freezing" the display for their duration. In this case, line 170 holds down the flicker of the ball. To remove it, simply type 170 and hit the RETURN key and then RUN the program again. As you see, the ball remains visible at each position for a little shorter time, thus making it seem to flash. You can put the delay loop back in by stopping the program and retyping line 170.

As you begin to develop your own graphics and sound programs, you'll have to experiment with timing loops. The loop

```
FOR T = 1 TO 738: NEXT T
```

stops action for exactly one second (the time it takes the computer to perform the loop control functions 738 times). You must, however, consider other actions in setting the duration of the loop. In this program, the timing loop is part of a larger loop (lines 140–210) that does several POKEs and a couple of IFs, actions that usually take more time than the pointless cycling of a timing loop. The more things a loop does, the shorter a timing loop has to be to stop action for a specified duration. The whole matter is largely one of trial and error. As an experiment that demonstrates this point graphically, replace the timing loop with

```
170  FOR T = 1 TO 369: NEXT T
```

and rerun the program. The ball now moves very slowly, advancing about once every second. The timing loop is holding it in place for half a second, and the other activities in the loop hold it there for another half-second.

Vectoring

Those who have studied physics know that a vector is the composite movement of an object acted upon by forces that try to move it in various directions. In this case, we have two dimensions, horizontal and vertical, that are represented by the X and Y coordinates, respectively.

As you run BOUNCING BALL #1, you'll observe that the ball always moves on the diagonal. This is because it is "acted on" by the variables XV and YV, the X and Y vectors, that move it one unit in each dimension. The ball starts out by moving southeast, because of XV = 1 (moving it right each step) and YV = 1 (moving it down each step). This is established in line 130 and continues in line 190.

Eventually the ball hits the "floor" and line 200 prevails; Y becomes equal to 39, so the sign of XY is reversed to negative. When the computer again reaches line 190, it "adds" YV to Y, and since YV is now −1, in fact it subtracts one vertical unit from the position of Y and moves it

upward. The ball thus rebounds from the floor and continues to move toward the top. When Y = 0, the ball hits the "ceiling," and again line 200 reverses the sign of YV to the negative of a negative (in other words to a positive value), so that the ball once more moves downward until Y = 39.

The X vector works the same way. The screen has 40 horizontal units and only 25 vertical positions, so moving 45° downward the ball has to hit the floor before it hits the wall. On the rebound, though, it strikes the right wall next, and line 210 reverses the sign of XV to −1. Line 190 then moves the ball left until it reaches the left wall, when once more it reverses the sign of the horizontal vector XV and begins moving the ball to the right.

You can see the impact of vectoring by retyping line 130 so that XV = 2 and then RUNning the program. (Be sure to change the delay loop at line 170 back to a value of 10 or you'll be up all night.) The ball now moves right two places for every one unit of vertical motion, so its angle of incidence with the sides is 22.5° instead of 45°. (This program was not written with such vectors in mind, so it behaves a little strangely and it will "crash" if you monkey with the YV.)

Solid objects

Very soon I'll add an obstacle for the ball, but first let's discuss the creation of solid objects on the screen. We now know that we can make a ball by POKEing the number 81 into screen memory and giving it a color. The Commodore 64 can also create many other shapes within a character cell; I'll get into all those possibilities presently. For the moment, I'll deal with shape #224.

This figure is a solid block that fills the entire character position. By POKEing several of them into a line, and making several lines of them in the same columns, we can make a box on the screen. Of course, we have to give them a color by POKEing a number into the corresponding positions in color memory. To see this in action, enter the program in Fig. 2.4.

```
NEW
120 T = 9: B = 18
130 L = 23: R = 32
140 REM  ** SET UP SCREEN
150 PRINT CHR$(147): BG = 2: BX = 0
160 POKE 53280,BX: POKE 53281,BG
170 REM  ** DRAW THE BOX
```

```
180 FOR Y = T TO B
190 FOR X = L TO R
200 P = X + (Y * 40)
210 POKE 1024 + P, 224
220 POKE 55296 + P, BX
230 NEXT X
240 NEXT Y
RUN
```

Figure 2.4 Solid box on the screen

This program makes the border turn black and the screen turn red. It then draws a black box toward the lower right corner of the screen. Although right now it doesn't do anything else, we'll soon incorporate it into a large bouncing ball program. For the moment, let's go line by line and see what Fig. 2.4 does.

Line	*What it does*
120	Sets top (T) and bottom (B) lines of box.
130	Sets left (L) and right (R) sides of box.
140	Describes the next two lines.
150	Clears screen, sets the background color (BG), and the box and border color (BX).
160	Changes the screen characteristics per line 150.
170	Describes the next block of instructions.
180	Starts the loop for the top-to-bottom lines.
190	Starts the loop for the left-to-right columns.
200	Calculates the memory offset for the column/row.
210	Puts the block character 224 into screen memory for the column and row.
220	Turns on the box color for that screen position.
230	Ends the column loop.
240	Ends the row loop.

Figure 2.5 Blow-by-blow of Fig. 2.4.

You can change the position of the box and its dimensions by altering the values of T, B, L, and R in lines 120 and 130, subject to the restrictions that (1) T must be less than B and both must be within the range 0–24, and (2) L must be less than R and both within the range 0–39, since those are the dimensions of the screen. You can also alter the colors of the border, box, and background by changing the values of BX and BG in line 150.

Adding an obstacle for the ball

A ball that bounces around an empty screen is okay for a while, but it becomes much more interesting when it has to make its way around an obstacle of some sort. BOUNCING BALL #2 uses the solid object developed in Fig. 2.4 to create this obstruction. To get Fig. 2.6, you can save yourself some rekeying by typing lines 100–110 and from line 250 onward. The Commodore will insert them where they belong in the program.

```
100 REM  **  BOUNCING BALL #2
110 REM  **  LOCATION OF BOX
120 T = 9: B = 18
130 L = 23: R = 32
140 REM  **  SET UP SCREEN
150 PRINT CHR$(147): BG = 2: BX = 0
160 POKE 53280,BX: POKE 53281,BG
170 REM  **  DRAW THE BOX
180 FOR Y = T TO B
190 FOR X = L TO R
200 P = X + (Y * 40)
210 POKE 1024 + P, 224
220 POKE 55296 + P, BX
230 NEXT X
240 NEXT Y
250 REM ----------
260 REM  **  SET UP THE BALL
270 X = 1: Y = 1: P = 0: XV = 1: YV = 1
280 REM  **  LOOP
290 POKE 1024 + P, 81: POKE 55296 + P, 1
300 REM  **  HIT BORDER?
310 IF X <= 0 OR X >= 39 THEN XV = -XV
320 IF Y <= 0 OR Y >= 24 THEN YV = -YV
330 REM  **  HORIZONTAL VECTOR
340 P1 = (X + XV) + (40 * Y)
350 IF PEEK(55296 + P1) <> BX THEN X = X + XV: GOTO 370
360 XV = -XV: GOTO 330
370 REM  **  VERTICAL VECTOR
380 P1 = X + (40 * (Y + YV))
390 IF PEEK(55296 + P1) <> BX THEN Y = Y + YV: GOTO 410
400 YV = -YV: GOTO 370
410 POKE 55296 + P, BG
420 P = P1: GOTO 280
```

Figure 2.6 BOUNCING BALL with a box as an obstacle

We've already covered the creation of a solid object, so in Fig. 2.7 we'll examine this program from line 260 onward.

Line	*What it does*
260	Explains the purpose of the next line.
270	Initializes the starting position of the ball and the vectors. *Note:* P = 0, which establishes a value for the offset required in line 290.
280	The start of the main loop for bouncing the ball.
290	Places the ball and a white color at the current X and Y coordinates.
300	Explains the next two lines.
310	Reverses the vector of horizontal motion if the ball has struck the border walls.
320	Reverses the vertical vector if the ball has hit the ceiling or floor.
330	Explains the next two lines.
340	P1 is the tentative next horizontal position of the ball, given the current direction XV.
350	If the tentative next horizontal position of the ball is not a box color, advance the ball to it and go to 370.
360	(If the tentative next horizontal position is a box color) reverse the horizontal vector and try again at 350.
370	Explains the next two lines.
380	P1 is the tentative next vertical position of the ball, given the current X and Y vectors.
390	If the tentative vertical position is not occupied by a box color, advance the ball and go to line 410.
400	(If the tentative vertical position is a box color) reverse the vertical vector and try again from line 370.
410	Clear the old displayed position of the ball by POKEing the background color so that it becomes invisible.
420	The current position P becomes the tentative position P1, and then repeats the bounce loop from line 280.

Figure 2.7 Blow-by-blow of BOUNCING BALL #2

There are a few general notes we need to make about this program. First of all, the variable P1 gives the tentative position of the ball in lines 340 and 380, given the current vector(s). The actual current position P takes the tentative position only when the program has determined that the ball isn't trying to invade the box (at 420). Second, there is no timing loop in this program. What with computing tentative positions and re-trying and so forth, the program has enough to do, making a forced delay unnecessary.

Finally, line 410 clears the current position of the ball before displaying it in its new place by changing it to the background color. The ball is still there: you just can't see it because it's become the same color as the background. After it has disappeared, the program loops back to show the ball at its new location. You can, if you wish, trace the motion of the

ball by changing this line to read

```
410  REM
```

The old ball positions will then remain on the screen, giving a complete record of the ball's travels. Sooner or later, the program will seem to go to sleep if you do this. That's because the ball is retracing previously taken paths. Ordinarily, this doesn't happen until the screen is almost entirely filled with used balls.

You can change the location and size of the box by altering the variables in lines 120 and 130, subject to the limits described earlier. The ball takes different paths, depending on the obstacle's location and dimensions.

As in the case of BOUNCING BALL #1, the only way to stop the program is to hit the RUN/STOP key. After stopping the program, clear the screen by holding down SHIFT while you press the CLR/HOME key in the upper right corner of the keyboard.

Randomly placed obstacles

To add further interest to the action of the bouncing ball, you can use the Commodore 64's random number generator to place "pegs" at assorted points around the screen. A random number generator, as its name suggests, creates random numbers. Every time you use this function it returns a different number with an unpredictable value somewhere between 0 and 1. Try it by typing

```
PRINT RND(1)
```

and the computer will display a number preceded by a decimal point. Type it several more times and you'll get a different result each time. The 1 in RND(1) tells the computer to give you the next random number; you can also use 0 or any other positive number to get the next random value out of the generator. If you enter a series of negative arguments, such as RND(−1), the generator always returns to the same number.

The raw results of the random number generator can't be used in this case, because (1) they are fractional numbers and not the integers necessary for specifying columns and rows, and (2) they are always less than 1. As a consequence, we have to do something to the random numbers to get them into an acceptable form and range for our purposes.

BASIC has another function called INT that converts a fractional value into the next lower whole number (integer) by truncating—chopping off—at the decimal point. For example, INT(3.14159) yields the integer 3.

That's fine, except that INT(RND(1)) always returns the value 0.

Why? Because RND(1) returns a number less than 1, and if we truncate at the decimal point the result has to be zero. Therefore, to raise the random numbers into a desired range, we have to multiply by the upper limit of the range. In this case, we want random X coordinates (column positions) somewhere between 0 and 39, so the upper limit of the range is 40. To get a random integer in the range, we use the instruction

X = INT(40 * RND(1))

As a test case, let's say the random number RND(1) is 0.993413. This number times 40 is 39.73652, and INT(39.73652) is 39, which is the highest permissible number in the range of X coordinates. The formula can never return a number greater than 39 because the random number generator always furnishes a number less than 1. At the other extreme, suppose the next random number is 0.003121. Multiplied by 40, it comes to 0.12484, which then truncates to 0. The calculation thus provides random integers between 0 and 39. It never produces negative numbers, since the generator's output is always greater than 0.

Similarly, we can generate random Y coordinates (row numbers) in the range 0 to 24 with the statement

Y = INT(25 * RND(1))

and the value 25 serves as an upper boundary for the results.

To put this into practice, add Fig. 2.8 to BOUNCING BALL #2. With the program already in the computer, you can simply type these lines and the Commodore 64 will insert them at the proper places.

```
100 REM  ** BOUNCING BALL #3
241 REM  ** RANDOM PEGS
242 FOR P1 = 1 TO 15
243 X = INT(40 * RND(1))
244 Y = INT(25 * RND(1))
245 P = X + (Y * 40)
246 POKE 1024 + P, 224: POKE 55296 + P, BX
247 NEXT P1
RUN
```

Figure 2.8 Adding random pegs

With this addition, scattered pegs appear on the screen and the ball bounces off them as well as off the border and the box. Let it bounce for a while, then hit the RUN/STOP button and type RUN again. Each time you run the program, the pattern of pegs is different since they are placed by the random number generator. You can also vary the number of pegs by changing the upper limit of the loop counter in line 242. The statement

```
242  FOR PI = 1 TO 7
```

creates seven pegs instead of 15 as originally programmed. If you try a larger number, say 25, you'll find that the ball tends to get locked into an inescapable pattern. It can at 15, too, but the likelihood is lower with a smaller number of pegs.

Creating sound effects

Balls don't bounce soundlessly, as anyone knows who's ever played Ping-Pong. To add a dimension of realism to our bouncing ball programs, then, we need to create a sound when the ball ricochets off an obstacle or the border.

Later in this book we'll investigate soundmaking with the Commodore 64's synthesizer. For now I'll keep it simple by modifying the program to make a banging sound whenever the ball hits something and reverses one of its vectors. This can be accomplished by changing two program statements and adding 13.

The overall strategy of these changes is to turn on the sound whenever a vector reverses. This happens in four places in the program: when the ball hits the border (lines 310 and 320), and when the ball strikes an obstacle (330–360 and 370–400). We'll insert a GOSUB instruction at those points to call the subroutine that turns on the synthesizer. The subroutine itself will be tacked on the end of the program in lines 430–450.

The GOSUB instruction interrupts the normal one-after-another flow of a program by telling the computer to GO to the SUBroutine at line so-and-so and execute the instructions there. When the computer encounters a RETURN statement in the subroutine, it goes back and resumes running at the instruction following the GOSUB. This is the same idea as saying, "Stop reading this book and get a drink of water, then pick up where you left off." In this case, we'll tell the computer, whenever it reverses a vector, to "go turn on the synthesizer, and then resume."

The synthesizer, of course, has to know what to do before we turn it on. We're going to insert six new lines starting at 250 that will tell it. The computer's Sound Interface Device (SID) produces an astonishing variety of sounds with as many as three voices simultaneously. To do this, it looks at 24 memory locations starting at address 54272 for instructions, which are numeric values POKEd there by a program. These *SID registers*, as the 24 locations are called, each have a specific purpose, and the values POKEd in them tell the synthesizer what sorts of sounds to produce. In this case, we want the synthesizer to produce only two sounds: a banging noise or silence. Consequently, we can set up the SID

registers ahead of time and then simply turn the synthesizer on and off. Lines 250–255 do the setting up, lines 430–450 turn the synthesizer on (when called by a GOSUB), and line 295 turns it off.

Two of the existing program lines need to be modified. These are lines 310 and 320, which reverse one or the other vector when the ball is at the border. The easiest way is to type

LIST 310

which displays line 310 on the screen. Use the cursor keys to move the cursor to the end of the line as displayed, then type

: GOSUB 430

The colon is very important. The line should now read

310 IF X <= 0 OR X >= 39 THEN XV = −XV: GOSUB 430

Make exactly the same change to line 320. Because the GOSUB is on the same line as an IF, it is only executed if the condition is true, that is, if the ball is at the right or left wall and the vector changes: no vector change, no GOSUB. The rest of the program changes and additions are shown in Fig. 2.9.

```
100 REM  **  BOUNCING BALL #4
250 REM  **  SET UP SOUND
251 N = 54272
252 FOR A = 0 TO 24: POKE N+A, 0: NEXT A
253 POKE N+0, 48: POKE N+1, 4
254 POKE N+5, 8
255 POKE N+24, 15
295 POKE N+4, 0
355 GOSUB 430
395 GOSUB 430
430 REM  **  TURN ON SOUND OF BALL HITTING
440 POKE N+4, 17
450 RETURN
```

Figure 2.9 Adding sound effects

Line 100 changes the existing program name line to read BOUNCING BALL #4, since this is the fourth version of this program. The other lines are new and the computer will insert them into the appropriate places according to their numbers. Figure 2.10 gives a line-by-line description of what we've done.

Line	*What it does*
(Lines 250–255 are a unit that sets up the SID registers.)	
250	Identifies the following four lines.
251	Gives the starting address of the SID registers.
252	Clears all 24 registers to eliminate garbage that might cause the synthesizer to produce unwanted sounds.
253	Sets a note value to be played (C below middle C).
254	Sets up the attack/decay value for a sharp (staccato).
255	Turns on the volume control.
295	After advancing the ball to its new position (line 290), this instruction turns off the synthesizer. It is always executed, even when the sound is already off, and has no effect in that case.
355 & 395	These GOSUBs are inserted into the routines that try the tentative horizontal and vertical positions. When the program discovers that it's trying to move the ball into an obstacle, it turns on the bounce sound with these instructions before reversing the vector and trying again.
(430 through 450 are a subroutine.)	
430	Identifies the subroutine.
440	Turns on the synthesizer by telling it to produce a triangular waveform. The sound remains on until line 295 squelches the synthesizer by POKEing a "no waveform" value into the same location (N + 4).
450	Reverts back to the instruction following the GOSUB that called this subroutine.

Figure 2.10 Narrative of sound-effects changes

Now that you know what the modifications do, type RUN and listen. The ball sounds as though it's bouncing off a metal garbage can or a tin roof. We've deliberately made it not very loud; you can add volume if you want by cranking up the sound level of the TV set. If you want to give it a more distinctive sound, read on and after we've covered the synthesizer in more detail you can come back and tinker with this program. By then you'll know a lot of other neat graphics tricks you can add, too.

CHAPTER 3
CHARACTER GRAPHICS

The Commodore 64 has (appropriately) 64 special graphics characters that we can combine in various ways to make pictures, graphs, outlines, and other attractive displays. You can see them by tilting the front of the computer up and examining the fronts of the keycaps. The character on the right is obtained by typing the key while holding down SHIFT, and the one on the left by typing while holding down the Commodore key (the one below RUN/STOP with the Commodore logo on it, which I'll abbreviate as C= key). This is easy to remember because of the position of the SHIFT and C= keys. To see the effect, pick any letter key and type it three times, once with no other key depressed, then with SHIFT held down, and finally with the C= down. These two keys change the high-order bit setting of the character, forcing the Commodore 64 to produce graphics.

To see all the built-in graphics characters of the Commodore 64, along with a practical application of some of them, enter Fig. 3.1 into your computer.

```
NEW
100 REM  **  GRAPHICS SAMPLER
110 PRINT CHR$(147)
120 FOR P = 1 TO 6: PRINT: NEXT P
130 PRINT TAB(13) "GRAPHICS SAMPLER"
```

(Continued)

```
140 PRINT CHR$(19);
150 T = 5  : B = 22
160 L = 3  : R = 37
170 S = 93 : H = 64
180 SM = 1024: CM = 55296
190 P = L + T * 40: POKE SM+P, 112: POKE CM+P,1
200 P = R + T * 40: POKE SM+P, 110: POKE CM+P,1
210 P = R + B * 40: POKE SM+P, 125: POKE CM+P,1
220 P = L + B * 40: POKE SM+P, 109: POKE CM+P,1
230 FOR X = (L+1) TO (R-1)
240 FOR Y = T TO B STEP (B-T)
250 P = X = Y * 40: POKE SM+P, H: POKE CM+P,1
260 NEXT Y: NEXT X
270 FOR Y = (T+1) TO (B-1)
280 FOR X = L TO R STEP (R-L)
290 P = X + Y * 40: POKE SM+P,S: POKE CM+P,1
300 NEXT X: NEXT Y
310 REM  **  DISPLAY CHARS 64 THRU 127
320 CH = 64: X = L+2: Y = T+5
330 REM  **  LOOP STARTS HERE
340 P = X + Y * 40: POKE SM+P, CH: POKE CM+P,1
350 CH = CH + 1
360 IF CH = 128 THEN 999
370 X = X + 2
380 IF X > (R-2) THEN L = L+2: Y = Y+3
390 GOTO 330
999 END
RUN
```

Figure 3.1 Graphics character sampler

This program draws a box enclosing an explanatory legend and displays each of the special graphics characters in a neat and professional-looking format against the background color of your screen. To see how it does it, let's dissect the program (Fig. 3.2).

Line	*What it does*
110	Clears the screen, homes the cursor.
120	Moves the cursor down 6 lines.
130	Moves the cursor right 13 spaces and displays the legend.
140	Homes the cursor without clearing the screen.
150	Sets the row numbers for the top (T) and the bottom (B) of the box.
160	Sets the column numbers for the left (L) and the right (R) of the box.
170	Gives the graphics character numbers for the vertical sides (S) and the horizontal sides (H).

180 Furnishes the addresses for the screen memory (SM) and color memory (CM) of the computer.
190 Computes the offset position (P) for the upper left corner of the box and POKEs the graphics character and the color white into memory.
200 Places the upper right corner of the box, as in line 190.
210 Places the lower right corner of the box, as in line 190.
220 Places the lower left corner of the box, as in line 190.
(230–260 form a nested loop that draws the top and bottom of the box.)
230 Steps across the screen between the corner angles.
240 Flips alternately between the top and bottom of the box.
250 Computes the offset for the current X and Y and places the horizontal side in white into memory. This causes it to be displayed, and gives the impression that the lines are being drawn.
260 Closes the loops.
(270–300 form a nested loop that draws the vertical sides of the box.)
270 Steps down the box between the upper and lower corners.
280 Flips alternately between the left and right sides.
290 Computes the offset for the current X and Y and places the vertical sides in white into memory. This causes it to be displayed, and gives the impression that the lines are being drawn.
300 Closes the loops.
320 Sets up values for the sampler loop that follows:
CH is the graphics character number;
X is the current horizontal position, starting two spaces from the left side of the box;
Y is the current row, starting five rows below the top of the box.
(330–390 display all the graphics characters.)
340 Computes the offset for the current X and Y and places the current character in white into memory. This displays the character.
350 Advances the current character number to the next higher.
360 If the new current character number is 128, end the program (graphics characters stop at 127).
370 Advances the current horizontal position two to the right.
380 If the new horizontal position is less than two spaces from the right side of the box, move back to the left side plus two and move down three rows.
390 Repeat the loop from line 330.
999 End the program.

Figure 3.2 Dissection of GRAPHICS SAMPLER

Appendix A shows all the graphics characters and their corresponding "POKE" codes (64 through 127). Note that these diagrams are enclosed in boxes. The outlines don't appear on the display. Instead, they indicate the position of the graphics character within an area of 8×8 dots, which is the Commodore 64's character cell (the amount of space occupied by one character on the screen).

You can obtain the reverse images of these special graphics characters and also see the full cell occupied by each one if you increase the character number by 128. There is a note at the end of Appendix A that makes

reference to this fact. To see the effect, type the following replacement line in GRAPHICS SAMPLER and run it again:

```
320  CH = 64 + 128: X = L+2: Y = T + 5
```

The sampler now appears with a white background and the graphics characters in the screen color (which is what a reverse image is). When programming with these graphics characters, you can save some trouble by using the base character number plus 128 in programs, instead of adding 128 yourself. For a reverse heart, use

```
POKE SM + P, 83 + 128
```

This will make it easier to go to the appendix to see what character you meant when you read the program later on.

Note the use of the homing function in line 140. This command, PRINT CHR$(19);—the trailing semicolon tells BASIC not to go to the next line after completing the operation—returns the cursor to the upper left corner of the screen, a convenient point of reference. In this case, it homes the cursor so that when the program ends, the messages output by BASIC will not interfere with the display.

Lines 120 and 130 use standard PRINT statements to write to the screen. Had we used POKEs, it would have been necessary to POKE each character in the legend "GRAPHICS SAMPLER" individually. Conversely, we could have used the CHR$ codes for the graphics characters in the sampler, which are shown in Appendix B. The CHR$ codes for graphics characters are different from the POKE codes, which adds a clumsy complication to the production of graphics on the Commodore 64. Also, the CHR$ codes are not continuous, as you can see in Appendix B. It's thus best to stick with one method or the other for producing graphics, and the POKE codes are more convenient. For that reason I'll seldom use the CHR$ codes for producing graphics in this book.

Checkerboards

Checkerboards offer some interesting but still easily understood applications of graphics. We'll build a few here to focus on some graphics programming techniques and to see some of the characters on the screen.

Even though 64 graphics characters seem to offer more patterns than you'll ever need, they have an immediate glaring lack: there is no entirely solid square that fills a character cell. For that we have to go into reverse image and use the space character. In Appendix A, you'll find that 96 is a space. Consequently, 96 + 128 is a reversed space. In other words, a normal space is, on the screen, an area occupied completely by the screen's background color, so a reversed space is a character cell occupied entirely by some color *other than* that of the background. We can make it any

color we want by POKEing a value into the corresponding location in color memory.

As a demonstration, press the CLR/HOME button to wipe out the sampler display and then type

```
POKE 53281,9
```

to turn the screen brown. Next type the command

```
POKE 1024 + 500, 96 + 128
```

which causes a reversed space to appear in the center of the screen. Do you see it? Of course not: we haven't assigned a color to it yet, so it takes the same color as the background. It's there, but it blends in. To give it a color, type

```
POKE 55296 + 500, 0
```

This assigns the color black to the character in screen memory + 500. As soon as you hit RETURN, the solid square in the center of the screen turns black.

Now let's use this block to make a checkerboard. Enter the program in Fig. 3.3

```
NEW
100 REM  ** CHECKERBOARD #1
110 PRINT CHR$(147)
120 BR = PEEK(53280): BG = PEEK(53281)
130 POKE 53280,0: POKE 53281,0
140 SM = 1024: CM = 55296
150 FOR Y = 0 TO 24 STEP 2
160 FOR X = 0 TO 38 STEP 2
170 GOSUB 260
180 NEXT X: NEXT Y
190 FOR Y = 1 TO 23 STEP 2
200 FOR X = 1 TO 39 STEP 2
210 GOSUB 260
220 NEXT X: NEXT Y
230 FOR T = 0 TO 5535: NEXT T
240 PRINT CHR$(147)
250 POKE 53280,BR: POKE 53281,BG: END
260 REM  ** SUBRTN TO DISPLAY SQUARE
270 P = X + Y * 40
280 POKE SM+P, 96+128: POKE CM+P,2
290 RETURN
RUN
```

Figure 3.3 Checkerboard using reversed spaces

The program builds and displays a black-and-red checkerboard pattern that fills the screen. Once the pattern is complete, it remains on the display for 7.5 seconds, and then the screen returns to its original border and background colors. It's a simple display. We'll see how the program does it in Fig. 3.4.

Line	*What it does*
110	Clears the screen and homes the cursor.
120	Reads and saves the current border (BR) and background (BG) colors.
130	Sets the border and background to black.
140	Establishes the base memory addresses for the screen memory (SM) and color memory (CM).
(150–180 are a nested loop that places red squares in the even-numbered columns of the even-numbered rows.)	
150	Starts the row loop, stepping by twos through the even numbers 0–24.
160	Starts the column loop, stepping by twos through the even numbers 0–38.
170	Calls the subroutine to display a red square.
180	Ends the two nested loops.
(190–220 are a nested loop that places red squares in the odd-numbered columns of the odd-numbered rows.)	
190	Starts the row loop, stepping by twos through the odd numbers 1–23.
200	Starts the column loop, stepping by twos through the odd numbers 1–39.
210	Calls the subroutine to display a red square.
220	Closes the two nested loops.
230	Timing loop to freeze the completed display for 7.5 seconds (@ 738 loops per second).
240	Clears the screen and homes the cursor.
250	Restores the border (BR) and background (BG) colors, and ends the program run.
(260–290 are a subroutine to display a red square at the current X and Y on the display.)	
270	Computes the memory offset position (P) based on the current X and Y coordinates on the screen.
280	Places a reverse space at the current position and turns on the color red for that position.
290	Returns execution to the statement following the GOSUB that called this subroutine.

Figure 3.4 Dissection of CHECKERBOARD #1

The alternating nature of a checkerboard pattern introduces a bit of complication into this program, since we need two separate sets of nested loops to handle the even- and odd-numbered rows and columns. It simplifies the program to use a higher resolution check, such as character 109 from Appendix A. Because it already contains two sets of squares that alternate, we can simply fill the screen with this character and set up the color of the checks. Figure 3.5 does this.

```
NEW
100 REM  **  CHECKERBOARD #2
110 PRINT CHR$(147)
120 BR = PEEK(53280): BG = PEEK(53281)
130 POKE 53280,0: POKE 53281,0
140 SM = 1024: CM = 55296
150 FOR Y = 0 TO 24
160 FOR X = 0 TO 39
170 GOSUB 260
180 NEXT X: NEXT Y
230 FOR T = 0 TO 5535: NEXT T
240 PRINT CHR$(147)
250 POKE 53280,BR: POKE 53281,BG: END
260 REM  **  SUBRTN TO DISPLAY SQUARE
270 P = X + Y * 40
280 POKE SM+P,127: POKE CM+P,2
290 RETURN
RUN
```

Figure 3.5 Higher resolution checkerboard

Notice that we borrowed most of CHECKERBOARD #1 for this program, replacing lines 150–220 with the new lines 150–180. These lines form a pair of nested loops that scan across each row, filling the entire screen. Also, we changed the character number in line 280 from 96 + 128 to 127, which is the double checkerboard character. Otherwise, the new program is the same as CHECKERBOARD #1, and the resulting display has twice as many checks.

You can bring this down to the finest check pattern possible on the Commodore 64—character 102, a pattern of 8 × 8 dots of alternating color—by retyping line 280 to read

```
280  POKE SM+P,102: POKE CM+P,2
```

and running it. As you see, it creates a very subtle check.

Alternate character sets

If you've examined Appendix A, you'll realize we've missed one level of resolution in our experiments—character 94, contained in Set 2. Perhaps you've wondered what is meant by Set 1 and Set 2 in the appendix. Well, wonder no more. The Commodore 64 has two standard character sets, that is, two different sets of images it can draw upon to construct character shapes on the screen. When you power up your Commodore 64, you automatically get Set 1. That's called a *default* in computerese, a

default being what you get unless you specify otherwise. To activate Set 2, press the Commodore (C=) key and the SHIFT key at the same time. Type something and you'll see that the output is displayed in lower case; the only way to get upper case is to hold down SHIFT while typing, as on a typewriter. To return to Set 1, press the C= and SHIFT keys again. These keys function as a latch/unlatch mechanism, flipping between the two character sets each time you press them simultaneously.

This two-key operation is fine while typing on the keyboard, but what if you want to change character sets within a program? The designers of the Commodore 64 anticipated this question by furnishing a pair of instructions that select the character sets. This is even better than the latch/unlatch, since the computer can't look at the screen to see which character set is active. To select Set 1, type

```
POKE 53272,21
```

If Set 1 is already active, this instruction has no effect, but if you're in Set 2, the computer switches to it. You can get Set 2 by typing the instruction

```
POKE 53272,23
```

These instructions work both from the keyboard and from within a BASIC program, allowing you to make a positive character set selection without worrying about the computer getting confused as to which set it's using at the moment.

Armed with this information, let's use the medium-resolution checkerboard given by character 94, Set 2. We can patch this into the program by adding the statements:

```
125  POKE 53272,23
245  POKE 53272,21
```

and changing line 280 by retyping it as

```
280  POKE SM+P,94: POKE CM+P,2
```

Run the program to see the effect. Line 125 switches to Set 2 before any output occurs, and line 245 switches back to the primary set after the program is finished. The change in line 280 merely specifies the appropriate character 94, and with Set 2 active it produces the medium-resolution checkerboard pattern.

One final note on alternate character sets: Only one set can be displayed on the screen at a time. You can't use one set for part of the display and then switch to the other. This causes instant and astonishing changes, because if you have a screen full of medium checks (character 94) and you switch back to Set 1, all the checks immediately become *pi* symbols (also character 94, but in the primary set). To see it actually happen, add the line

```
225 POKE 53272,21
```

and run the program. The display is built using checkerboards, but before it freezes with the timing loop, it switches back to Set 1. Just watch what happens.

Probably the vast majority of your graphics will use Set 1 inasmuch as Set 2 contains only four graphics symbols. It's a pity that Commodore couldn't fit them all into one set, but at any rate, switching character sets is not something you'll do often.

Mixing characters and graphics

The graphics sampler program mixes character data (the "legend") with graphics to create a complete, self-explanatory display. When mixing the two, you have to make sure a graphics character and a text character don't try to occupy the same position. Whichever gets there last prevails by overwriting the previous occupant of the cell. I'll discuss means of avoiding such territorial conflicts in the next section.

The methods for placing text and graphics on the display differ. We use POKEs to write graphics, and PRINTs to write text. A POKE does not move the cursor, whereas a PRINT does. That being the case, while programming it's easy to lose track of where the cursor is, hence of where the next PRINT will write on the screen.

To make it simpler, you should write all the text in one part of the program and all the graphics in another. The order doesn't matter, as long as the two different types of output are segregated. In effect, using this approach you either write out the text and build the display around it, or you build the display and then fill in the text. In either case, it's good practice to rehome the cursor after text output with the statement

```
PRINT CHR$(19);
```

That way you'll be absolutely sure that the cursor always comes to rest in the upper left corner of the screen.

You can also use "home" as a point of reference for vertical spacing of the text. Every time you issue a PRINT that is not followed by a semicolon, the cursor moves to the start of the next line. Conversely, when a PRINT (or the information it produces) *is* followed by a semicolon, the cursor stops on the current line. This is why the homing statement ends with a semicolon; it means "put the cursor in the upper left corner of the screen and leave it there," which provides a reference point during output. If your display is busy—with a label here, a legend there, a heading someplace else—it's easy to lose track of where you are and then your vertical spacing will come out wrong. You can simplify matters by re-homing the cursor and then issuing a PRINT for each line you want to

move it down. A simple PRINT does not disturb text that already exists in a line it passes through.

Don't forget to rehome the cursor when the text output is all done. The flashing cursor, incidentally, automatically disappears while a program is running on the Commodore 64, except that it turns back on during an INPUT instruction. When it reappears after the program is finished, it is in the position where it was last placed.

Planning the display

Because the Commodore 64 screen is a rectangle of 40 columns by 25 rows, an indispensable tool for planning displays is quadrille-ruled graph paper with five squares per inch. You can buy it where school supplies are sold. Lay out the sheet as shown in Fig. 3.6, labeling the columns 0 through 39 and the rows 0 through 24. This gives you a properly proportioned worksheet on which to draft displays. You can then program from it, using the column and row numbers as X and Y coordinates.

If you want to be spiffy about display planning, you can use markers of the appropriate colors to portray an exact image of the screen. It works just as well, however, to block out and darken filled-in areas, perhaps

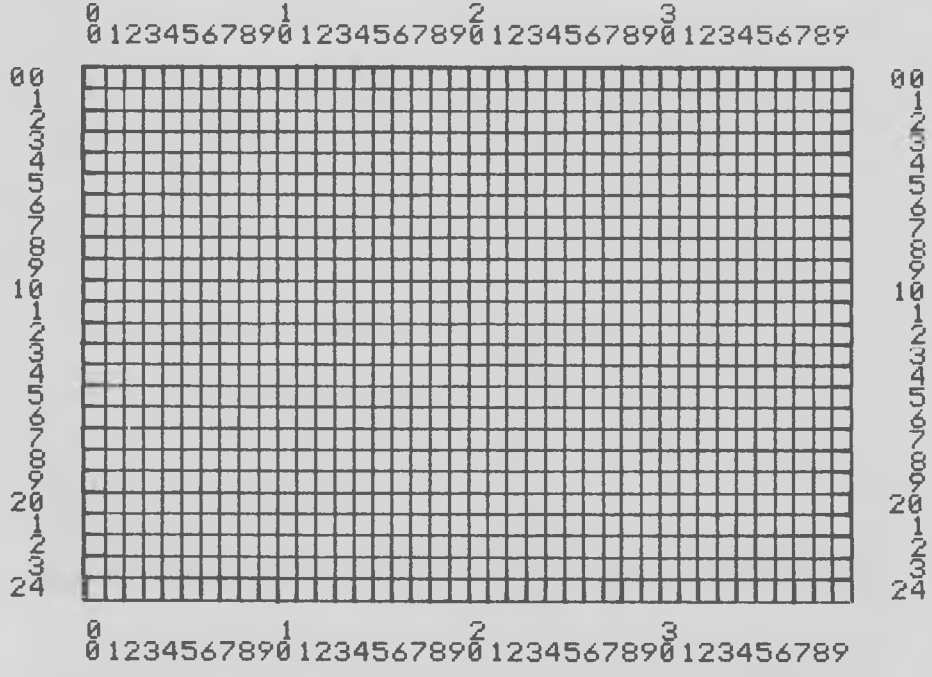
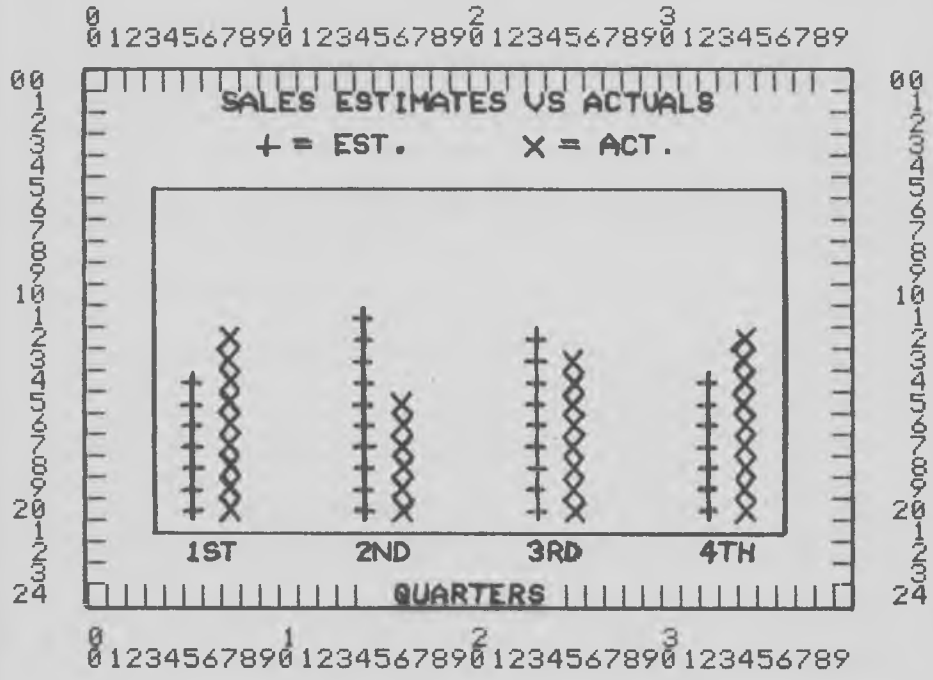


Figure 3.6 Blank screen worksheet

using rough patterns of varying kinds to convey different colors as in needlepoint charts, and jot notes to yourself on the drawing. The idea is to make a sketch from which to program, not a work of art.

As a case study in display planning, and also to illustrate the mixing of text and graphics, let's play "impress the boss" and make a colored bar chart that shows projected and actual sales volumes for each quarter of the year. A green bar will show the estimated sales, a red bar the actual sales, by quarter. This is an important presentation, since not only are sales good, but we want to show the boss what accomplished programmers we are, and it's also time for raises and promotions. Consequently, the display has to be professional looking.

The layout of the display is shown in Fig. 3.7. Note that we can't really show the actual height of each bar, since that depends on sales. Of course,



Key to symbols: + Red square
X Green square

Notes: Black background and border.
Vertical bar height varies with data.
Box around bars is white.

Figure 3.7 Sales chart for the boss

we know the figures for this year, but we want to generalize the program to work with other years' sales as well, so the heights of the bars can vary widely and we don't want to have to rewrite the program for each year. We *do* know that the maximum sales volume for any quarter is $750,000, so we can apply a scale factor to figure out the maximum height for any possible bar on the display.

Before we write the program, let's digress for a moment to consider the scaling of data on displays. As the worksheet shows, the box outlining the area for the bars has its top at row 5 and its bottom at row 21. The bars rise from row 20 to a height approximately determined by the values they represent—in this case, sales estimated or actual. Since the top of the box is at row 5, the highest the bar can go within the box is row 6. Thus, we have 15 increments by which to measure up to $750,000. Each increment therefore represents $750,000 / 15 = $50,000.

In this program, we'll use the usual BASIC rounding procedure to find the nearest whole-number increment. The second part of lines 430 and 510 show how this is done. The result is admittedly approximate, since each square in a bar stands for $50,000 or a majority of that amount. Chapters 5 and 6, dealing with high-resolution graphics, will show how to hone this approximation down more finely.

Enter the program in Fig. 3.8 and, after you run it, we'll analyze its workings.

```
NEW
100 REM  ** SALES CHART
110 PRINT CHR$(147);
120 POKE 53280,0: POKE 53281,0
130 SM = 1024: CM = 55296
140 PRINT CHR$(5);
150 REM  ** WRITE TEXT TO DISPLAY
160 PRINT: PRINT TAB(7) "SALES ESTIMATES VS ACTUALS"
170 PRINT: PRINT TAB(11) "= EST." TAB(25) "= ACT."
180 PRINT CHR$(19);
190 FOR L = 1 TO 22: PRINT: NEXT L
200 PRINT TAB(5) "1ST" TAB(14) "2ND";
210 PRINT TAB(23) "3RD" TAB(32) "4TH"
220 PRINT: PRINT TAB(16) "QUARTERS";
230 PRINT CHR$(19);
240 REM  ** COMPLETE THE KEY
250 P = 9 + 3 * 40: POKE SM+P, 96+128: POKE CM+P,2
260 P = 23+ 3 * 40: POKE SM+P, 96+128: POKE CM+P,5
270 REM  ** DRAW BOX
280 T = 5: B = 21: L = 3: R = 36
290 P = L + T * 40: POKE SM+P,112: POKE CM+P,1
```

```
300 P = R + T * 40: POKE SM+P,110: POKE CM+P,1
310 P = R + B * 40: POKE SM+P,125: POKE CM+P,1
320 P = L + B * 40: POKE SM+P,109: POKE CM+P,1
330 FOR Y = T TO B STEP (B-T)
340 FOR X = (L+1) TO (R-1)
350 P = X + Y * 40: POKE SM+P,67: POKE CM+P,1
360 NEXT X: NEXT Y
370 FOR X = L TO R STEP (R-L)
380 FOR Y = (T+1) TO (B-1)
390 P = X + Y * 40: POKE SM+P,93: POKE CM+P,1
400 NEXT Y: NEXT X
410 REM  **  BARS FOR EST SALES (RED)
420 FOR X = 5 TO 32 STEP 9
430 READ A: T = INT((A / 50000) + .5)
440 IF A = 0 THEN 480
450 FOR Y = 20 TO (20-T) STEP -1
460 GOSUB 590: POKE CM+P,2
470 NEXT Y
480 NEXT X
490 REM  **  BARS FOR ACTUAL SALES
500 FOR X = 7 TO 34 STEP 9
510 READ A: T = INT((A / 50000) + .5)
520 IF A = 0 THEN 560
530 FOR Y = 20 TO (20-T) STEP -1
540 GOSUB 590: POKE CM+P,5
550 NEXT Y
560 NEXT X
570 REM  **  ENDLESS LOOP TO FREEZE DISPLAY
580 GOTO 580
590 REM  **  SUBRTN FOR SQUARE
600 P = X + Y * 40: POKE SM+P, 96+128
610 RETURN
620 REM  **  EST SALES DATA BY QUARTER
630 DATA 300000, 400000, 200000, 500000
640 REM  **  ACT SALES DATA BY QUARTER
650 DATA 375000, 460000, 300000, 700000
RUN
```

Figure 3.8 Sales volumes bar-chart program

This program freezes the display by entering an endless loop at line 580. We can talk to the boss all afternoon and the display won't alter a bit, because the computer stays busy going in circles. To stop it, hit the RUN/STOP button on the left side of the keyboard.

Note line 140, which says to PRINT CHR$(5);. This instruction has the

same effect as when you hold down the CTRL key and press 2 to force white letters on the screen. Unfortunately, the Commodore 64 does not have a one-for-one correlation between CHR$ codes for colors and their numbers on the keyboard, so that CHR$(5) is equivalent to CTRL-2 for white, CHR$(28) is the same as CTRL-3 for red, etc. Appendix B contains all the CHR$ codes for colors of lettering. Since the keyboard color numbers, the POKE codes, and the CHR$ codes are all different, you'll probably have to look up the CHR$ codes each time you use one. The table in Fig. 3.9 consolidates the various color codes.

Color	*Keyboard*	*POKE*	*CHR$*
Black	CNTL 1	0	144
White	CNTL 2	1	5
Red	CNTL 3	2	28
Cyan	CNTL 4	3	159
Purple	CNTL 5	4	156
Green	CNTL 6	5	30
Blue	CNTL 7	6	31
Yellow	CNTL 8	7	158
Orange	C= 0	8	129
Brown	C= 1	9	149
Light red	C= 2	10	150
Dark gray	C= 3	11	151
Medium gray	C= 4	12	152
Light green	C= 5	13	153
Light blue	C= 6	14	154
Light gray	C= 7	15	155

Figure 3.9 Color codes for the Commodore 64

Examine lines 200–220 in the program. Line 200 ends with a semicolon, telling BASIC not to advance the cursor to the next line of output. As a result, when the PRINT in line 210 is executed, the output appears on the same row as that from line 200. The value in parentheses following TAB tells BASIC how many print positions from the start of the line to begin the output (where 1 is the first position in the line). Line 210, then, properly spaces the output so that it appears that all the quarter designations were produced by a single PRINT statement.

The semicolon at the end of line 220 deserves an explanation. It's the last legend of the display, so why have a semicolon? The answer is that we don't want BASIC to send a new-line signal to the screen because that would shift the entire text display upward one row, thereby wrecking all our carefully planned vertical spacing. To keep BASIC from doing this, we have to put a semicolon at the end of the instruction.

Finally, note lines 630 and 650, which are DATA statements. It was our intent to generalize this program so that it could be used with data

from other years. The DATA statements enable us to do that. All we have to do is retype these two lines with any other years' estimated and actual sales figures, and the program will change its output accordingly.

We'll analyze the SALES CHART program in Fig. 3.10.

Line *What it does*

110 Clears screen, homes cursor.
120 Resets border and background to black.
130 Establishes the screen and color memory base addresses.
140 Sets letter coloring to white.
(150–260 write text and the color-code key.)
160 Writes the chart heading in row 1.
170 Writes the text portions of the color-code key in row 3.
180 Rehomes the cursor.
190 Spaces down to row 22.
(200 & 210 write the quarter legends under the bars.)
220 Writes the QUARTERS literal in the last line (semicolon inhibits a new line that would shift the display up).
230 Rehomes the cursor.
(240–260 Place squares of appropriate colors to complete the color-code key.)
(270–400 draw a white box around the bars area.)
280 Sets the top, bottom, left, and right boundaries of box.
(290–320 set up the corners of the box.)
(330–360 are a pair of nested loops that draw the top and bottom edges of the box.)
330 Starts a loop that flips the Y between the top and the bottom edges of the box.
340 Starts a loop that draws the top and bottom edges from left to right.
350 Places the horizontal line segment and white color into memories.
360 Ends the nested loops for the top and bottom.
(370–400 are a pair of nested loops that draw the left and right sides of the box.)
370 Starts a loop that flips the X between the left and right sides.
380 Starts a loop that draws the sides of the box from top to bottom.
390 Places the vertical line segment and white color into memories.
400 Ends the nested loops that draw the sides.
(410–480 use data to display the red bars for estimated sales.)
420 Starts a loop that moves across the display, building each bar for estimated sales.
430 Reads the estimated sales for the quarter being worked on and reduces the amount to the number of vertical increments it represents on the display.
440 Skips to the end of the loop (producing no bar) if the number of scale increments is 0.
450 Starts a nested loop that steps upward from the bottom of the bar for the computed number of scale increments.
460 Calls a subroutine to figure the current position memory offset and place a square in screen memory. Upon regaining control, writes the

(Continued)

color red to the corresponding color memory address, displaying the square.
470 Ends the nested loop to build the bar.
480 Ends the outer loop for the estimated sales bars.
(490–560 perform the same way for actual sales, making the bars green.)
580 Locks the program into an endless loop, freezing the display until you press the RUN/STOP button.
(590–610 are a subroutine that services the bar builders.)
600 Computes the memory offset for the current X and Y, and POKEs a square at that place in screen memory.
610 Restores control to the instruction following the one that called this subroutine.
(620–650 contain sales data used by the program.)
630 Estimated sales volumes by quarter in order.
650 Actual sales volumes by quarter in order.

Figure 3.10 Dissection of SALES CHART program

Graphics without POKES

So far almost all our graphics have been created through the use of POKE instructions. It is possible, but more difficult, to create graphics on the Commodore 64 with PRINT CHR$ statements. The CHR$ codes are listed in Appendix B, and for the most part there are corresponding POKE and CHR$ codes for functions.

As an example of graphics with CHR$ codes, let's write a sampler program that displays the entire spectrum of Commodore 64 colors, with each color labeled and the "rainbow" in a rectangle. This program is useful for adjusting the color and distortion of the picture tube.

When we use POKEs to write graphics figures to the screen, we alter the screen and color memories. This gives us absolute individual control over each of the 1000 display cells. With PRINT CHR$'s, on the other hand, we issue codes that alter the way the display operates until the next CHR$ is issued. Consequently, PRINT CHR$'s are faster than POKEs, but for this greater speed we sacrifice much of our control over the screen. There are some things we can do with POKEs that simply aren't possible with PRINT CHR$'s. The CHR$ codes, then, are useful for simple graphics, but not for more complex displays.

Enter and run the program in Fig. 3.11 and then we'll discuss its nuances in detail.

```
NEW
100 REM  ** COLOR SAMPLER
110 X$ = CHR$(18) + "                                    ":REM  20 SPACES
120 PRINT CHR$(147) CHR$(5)
```

```
130 POKE 53280,0: POKE 53281,0
140 FOR X = 1 TO 15
150 READ C$, C
160 PRINT C$ TAB(10) CHR$(C) X$ CHR$(5)
170 NEXT X
180 END
190 DATA "WHITE",5,"RED",28
200 DATA "CYAN",159,"PURPLE",156
210 DATA "GREEN",30,"BLUE",31
220 DATA "YELLOW",158,"ORANGE",129
230 DATA "BROWN",149,"LT RED",150
240 DATA "DK GRAY",151,"MED GRAY",152
250 DATA "LT GREEN",153,"LT BLUE",154
260 DATA "LT GRAY",155
RUN
```

Figure 3.11 A color sampler for adjusting your display

At first glance this seems like a simple little program. Of its 17 lines, the instructions occupy only seven (110 through 170), and most of the work is done by a four-line loop (140 through 170). On closer examination, however, we find a number of technical points worth discussion.

The first two are in line 110. The code CHR$(18) turns on the reverse video feature, which forces all succeeding characters to appear as negative images. This has the same effect as the POKE convention of adding 128 to the character number. Reverse video remains in effect until:

1. The end of the current output line, or;
2. The code CHR$(146) is issued to cancel reverse video.

The second feature of line 110 is what programmers call *string concatenation*: joining two or more units of nonnumeric data together to form a single unit. In this case, we're attaching CHR$(18) to the start of a string of 20 spaces and calling the result by the name X$. The effect of the statement PRINT X$ is to print a bar 20 spaces long in whatever color the display is currently producing.

Lines 120 and 130 set up the screen for output by clearing it [(CHR$(147)], setting the text color to white [CHR$(5)], and making the border and background black (line 130). The semicolon was deliberately left off line 120 to palce the cursor on the second row of the screen. Now we're ready to begin output.

Line 140 starts the FOR/NEXT loop. Note that the loop counter X does not include color 0, which is black. This is because a black bar won't show up against a black background. Line 150 is executed each time the loop repeats, reading the next color name (C$) and color CHR$ code (C)

from the DATA statements. The color numbers were obtained from Fig. 3.9.

The sophistication occurs in line 160, which PRINTs:

1. The color name C$.
2. Skips over to column 10 of the line [TAB(10)].
3. The CHR$ code for the named color (C).
4. A 20-space bar in color C (X$).
5. CHR$(5) to reset the text color to white for the next output (usually the name of the next color).

The loop repeats from line 170 until all colors 1 through 15 have been printed, and then the program ends at line 180. The READY message is in white because of the last code issued in line 160.

Save this program on your tape or disk to use when you need to adjust your display.

To see why PRINT CHR$'s are not acceptable for higher graphics, let's say we want to create a midwestern horizon with a level green field against a blue sky. This is a very simple picture, but it illustrates the point. Enter Fig. 3.12 into your computer and we'll discuss it.

```
NEW
100 PRINT CHR$(147);
110 POKE 53280,0: POKE 53281,6
120 FOR Y = 1 TO 20: PRINT: NEXT Y
130 PRINT CHR$(30) CHR$(18);
140 FOR Y = 1 TO 199: PRINT " ";: NEXT Y
150 PRINT CHR$(19) CHR$(5);
RUN
```

Figure 3.12 Midwestern horizon

Lines 100 and 110 should be familiar by now. The one-line loop in 120 moves the cursor down to row 21, leaving the blue sky unchanged. Line 130 changes the output color to green [CHR$(30)] and turns on reverse video [CHR$(18)] with a trailing semicolon to tell BASIC that the line doesn't end here.

The crux of the matter is in line 140, which prints a sequence of 199 reversed spaces in green. There are 40 characters in a line, so this output fills rows 21 through 25 less the very last cell on the screen. Line 150 then rehomes the cursor and changes the text color to white.

Now, why leave the last cell unchanged? There it sits, one tiny square of blue sky somewhere in a midwestern field and clearly out of place. Let's try fixing it by increasing the number of green blocks in line 140 from 199 to 200. Do it and rerun the program.

Oh-oh! What happened? Now we've got a whole strip of blue sky *under* the field. This phenomenon, known as *scrolling*, occurs when the cursor moves through the last column of the bottom line and jumps to a new row, shoving the entire display upward. Problems of this sort make it simpler to use POKEs for graphics.

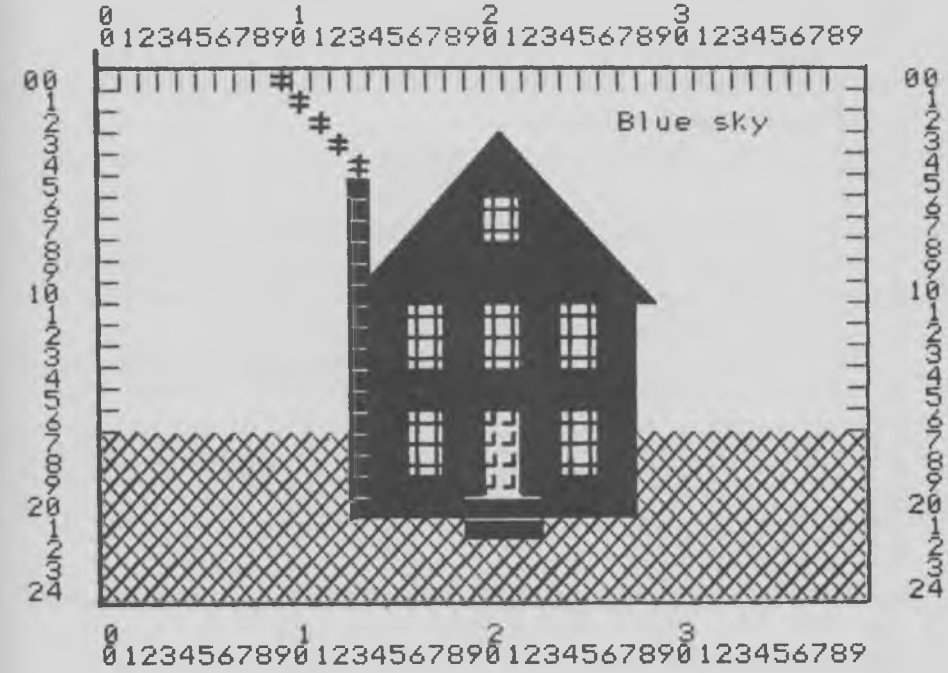


Figure 3.13 A house built of character graphics

Building a house

Grandma Moses, eat your heart out. Character graphics and a little imagination can create primitive art on the Commodore 64 in a fraction of the time it takes to paint a picture. As an exercise in quickie primitive art,

we'll build the house shown in Fig. 3.13. Type each of the listings that follow into your computer to build the program as we go along.

The first thing we need to do is prepare the screen and set up some constant information for the program in Fig. 3.14.

```
NEW
100 REM  **  BUILDING A HOUSE
110 PRINT CHR$(147);
120 POKE 53280,0: POKE 53281,14
130 B = 96+128: GOTO 200
```

Figure 3.14 Heading for the house-building program

Line 120 sets the border to black and the background to light blue, the color of the sky. Line 130 declares the variable B as a reverse space (POKE code 96 plus 128 to make it a negative image). The purpose of the GOTO is to bypass the subroutine we'll discuss next.

This program requires a lot of POKEs into screen and color memories, so we'll set up a common subroutine to compute POKE addresses from the row coordinate Y and the column coordinate X (Fig. 3.15). This subroutine takes a different approach from what we've used before. It returns the complete addresses of the screen-memory location (S) and the corresponding color-memory location (C) so that we don't have to calculate them each time.

```
140 REM  **  SUBRTN FOR MEMORY LOCATIONS
150 P = X + (Y * 40)
160 X = 1024 + P          :REM  SCREEN MEM
170 Y = 55296 + P         :REM  COLOR MEM
180 RETURN
190 REM  ----------
```

Figure 3.15 Memory locations from XY coordinates

Now we're ready to start in earnest. If you ran this program, it would turn the screen blue and report an undefined line error since BASIC can't execute GOTO 200 (so don't run it). A blue screen is just sky—space—so we need Fig. 3.16 to computerize Genesis by creating the Earth.

```
200 REM  **  LET THERE BE EARTH
210 FOR Y=24 TO 17 STEP-1
220 FOR X=0 TO 39: GOSUB 140
230 POKE S,B: POKE C,13
```

```
240 NEXT X: NEXT Y
RUN
```

Figure 3.16 Laying the ground

Our pioneer ancestors had a tradition known as a raisin', in which everybody got together and put up a house or a barn in no time flat. Like Grandma Moses, they'd be madly jealous if they could see how quickly and easily Fig. 3.17 does it.

```
250 REM  **  RAISE THE HOUSE
260 FOR Y=20 TO 11 STEP-1
270 FOR X=14 TO 27: GOSUB 140
280 POKE S,B: POKE C,1
290 NEXT X: NEXT Y
RUN
```

Figure 3.17 The raisin'

This portion of the program builds a solid white block sitting on the green lawn somewhere this side of the horizon. It's not a house yet, but at least the basic structure is there. As in reality, the house builds from the ground up in line 260, which steps backwards (bottom to top) through the Y coordinates.

```
300 REM  **  PUT UP THE GABLES
310 L = 14: R = 27
320 FOR Y=10 TO 4 STEP-1
330 FOR X=L TO R: GOSUB 140
340 POKE S,B: POKE C,1
350 NEXT X
360 L = L + 1: R = R - 1
370 NEXT Y
RUN
```

Figure 3.18 Adding a peaked roof

Figure 3.18 stairsteps the gables until they join at the center of the house. Lines 310 and 360 show how this is done. Line 310 sets the left and right boundaries of the gables. Line 330 loops between those boundaries, filling (between 330 and 350) the space with solid white blocks. Line 360 moves the left boundary right and the right boundary left by one space each, so when the Y loop repeats the next time, the X loop at line 330 has narrower boundaries.

```
380 REM  **  PUT ON THE ROOF
390 L = 13: R = 28
400 FOR Y=10 TO 3 STEP-1
410 X=L: GOSUB 140: POKE S,105+128: POKE C,1
420 X=R: GOSUB 140: POKE S,95+128: POKE C,1
430 L = L + 1: R = R - 1
440 NEXT Y
RUN
```

Figure 3.19 Raising the roof

In this case (Fig. 3.19), we don't have an X loop because we don't need to fill in between the left (L) and right (R) bounds. Instead, those boundaries are where we place a single 45° slope. Lines 410 and 420 do that, using the reverse of POKE codes 105 on the left slope and 95 on the right. The handling of the L and R variables is just the same as in building the gables; in other words, shifting them at each Y level until we reach the top.

The trouble with this house is that there's no entrance. Before work can progress, it has to have a door. To break up the blandness of a white house, we'll make the door red (see Fig. 3.20).

```
450 REM  **  INSTALL THE DOOR
460 FOR Y=19 TO 16 STEP-1
470 FOR X=20 TO 21: GOSUB 140
480 POKE S,B: POKE C,10
490 NEXT X: NEXT Y
RUN
```

Figure 3.20 Adding a red door

Now that they are able to gain access, our builders can proceed to install five windows, two on the first floor and three on the second. Since all five windows are the same except for placement, they'll use a subroutine to install the windows and call it with a loop that specifies where to put them. Figure 3.21 is the most complicated part of building the house.

```
500 REM  **  INSTALL 5 WINDOWS
510 V = 18: H = 16: G = 91 + 128
520 FOR W=1 TO 6
530 IF W = 2 THEN 560
540 IF W = 4 THEN V = 13: H = 16
550 GOSUB 590
```

```
560 H = H + 4
570 NEXT W
580 GOTO 650
590 REM   WINDOW-MAKER
600 FOR Y=V TO V-2 STEP-1
610 FOR X=H TO H+1: GOSUB 140
620 POKE S,G: POKE C,0
630 NEXT X: NEXT Y
640 RETURN
RUN
```

Figure 3.21 Putting in the windows

The windowbuilder subroutine (590–640) builds the window starting at the lower left corner. Consequently, we always have to tell it where the lower left corner is. Line 510 does this by setting up the vertical (V) and horizontal (H) position of the first window. It also specifies the material for the window, which is the reverse of POKE code 91 (the cross character). Lines 520–570 define the window (W) loop. The loop limit in 520 calls for six windows, but line 530 omits the second one, which the loop would try to put where the door already is. It's necessary to do it this way, since there is a window directly above the door on the second floor. Line 550 calls the windowmaker, and 560 advances to the next window to the right. After the installation of the third window, the loop value W is 4 and line 540 sends the windowmaker to the left window position upstairs; that's what "V = 13: H = 16" mean. Line 580 prevents the program from wandering into the subroutine after the last window is installed.

As for the windowmaker itself, line 600 steps upward by three rows, starting at the current vertical (V) level, corresponding to the floor it's working on. Line 610 uses the current horizontal position as a reference, so that these two loops are confined to an area two columns wide by three rows high, beginning at the lower left corner. The rest of the subroutine is old stuff.

A small window high in the gables will add a pleasing touch. Because this window is smaller than the others, we can't use the same windowmaker (see Fig. 3.22).

```
650 REM  **  ATTIC WINDOW
660 FOR Y=7 TO 6 STEP-1
670 FOR X=20 TO 21: GOSUB 140
680 POKE S,G: POKE C,0
690 NEXT X: NEXT Y
RUN
```

Figure 3.22 Making a small attic window

In the interest of energy-consciousness, we'll call in a mason to build a chimney up the left side of the house. Like the other builders, he's a fast worker and needs only the few instructions in Fig. 3.23 to do his job.

```
700 REM  **  PUT IN A CHIMNEY
710 M = 99 + 128
720 FOR Y=20 TO 5 STEP-1
730 X = 13: GOSUB 140
740 POKE S, M: POKE C,9
750 NEXT X: NEXT Y
RUN
```

Figure 3.23 Electronic stonemasonry

Line 710 specifies the material, which is reverse code 99. Line 720 tells the chimney mason how high to make it, and line 730 says where to put it with respect to the house (on the left) and calls the position subroutine for exact locations of each layer.

The house is nearly complete. The last thing to build is a front step leading to the door, which we'll have the mason lay while he's here. He can use the same material as for the chimney (M). Again, his instructions are sparse (see Fig. 3.24).

```
760 REM  **  BUILD THE FRONT STEP
770 FOR Y=21 TO 20 STEP-1
780 FOR X=19 TO 22: GOSUB 140
790 POKE S,M: POKE C,9
800 NEXT X: NEXT Y
RUN
```

Figure 3.24 More masonry

The house is now ready to welcome its new occupants. To give it a homey air for them, let's build a fire on the hearth. We're looking at it from the outside, of course, but so will they when they arrive, and smoke from the chimney is hospitable. The smoke rises and flows from the flue in gray puffs that trail off to the left because of line 850 (see Fig. 3.25).

```
810 REM  **  SMOKE FROM THE CHIMNEY
820 X = 13: G = 87
830 FOR Y = 4 TO 0 STEP-1: GOSUB 140
840 POKE S,G: POKE C,12
```

```
850 X = X - 1
860 NEXT Y
870 REM  ** HOLD THE PICTURE STEADY
880 GOTO 880
RUN
```

Figure 3.25 A fire's on the hearth

We could continue with this, adding trees and flower beds and picket fences and all the other trappings of a domestic establishment, but you get the picture. This is, alas, not as enduring as one of Grandma Moses' paintings, but it will stay on your screen until you hit the RUN/STOP button. Feel free to add other things (starting at line 870, not 890!), and experiment with the things we've learned in this chapter about character graphics. Also, make a special point of saving this program under the name HOUSE, because we'll revisit the neighborhood in a later chapter.

CHAPTER 4

THE CAST OF CHARACTERS, AND ADDING SOME

Computers only give the appearance of being intelligent; in reality they're profoundly stupid, just a bunch of electronic circuits that respond blindly to combinations of 1s and 0s with no forethought, no intuition, no insights, and no ability to learn from their experience. A good example of this "know-nothing" idiocy has to do with characters.

When you push the keyboard button inscribed with "W", the same character image appears on the screen in what seems an automatic cause-and-effect sequence. If you bother to think about it at all, you probably assume that there's a linkage similar to a typewriter's between the key and the display image, or at least some mechanistic wisdom that associates the two. In fact, however, it isn't that way at all. The computer makes no connection between the symbol on the keycap and the image it displays, and it doesn't care. When you push the W key, you close a switch contact that sends a number into electrical circuitry where it is sensed by software. The software looks up this number in a table of images to find the pattern specified for the button. Other software and circuitry then translate that image into dots on the screen to form the pattern that you recognize as a W. The computer makes no correlation between the W legend on the keycap and the pattern of bits it takes from the table; if the image happens to form a dollar sign, the computer blithely displays a dollar sign whenever you type W, and never wonders why.

This fundamental imbecility is, of course, not bad, but simply a fact. It's probably not much different than the way our minds work, since we too had to be taught this particular shape. You look at it and you say,

"Oh, yeah, that's a W." You've looked it up in your mental table of images. What differentiates human intelligence from machine "intelligence" is that we can take W and all its siblings and combine them to form written and spoken language, literature, art, mathematics, heritage, whereas a computer can only look it up in a table and display it.

Because the computer has such a narrow view of its cast of characters and doesn't care if they coincide with the keycaps, we can reprogram its character set and manipulate it in various ways to suit our purposes, and the computer will happily go right along with us. That's what this chapter is about.

The face of a character

A character cell consists of an 8 × 8 dot matrix that has a corresponding 8-byte image in memory, that is, 64 dots mapped to 64 bits. This image takes eight sequential memory addresses that are generally referred to by their offsets from the first byte; the first byte of the cell is byte 0 and the last byte is 7. It's useful to think of them as being stacked as they are on the display screen, with byte 0 on top and 7 on the bottom. Byte 8 is the top byte of the next character cell to the right, which is, of course, byte 0 for that cell. So it goes, in an up-and-down zigzag across the 40 cells that make a line. It takes, then, 320 (8 × 40) bytes to form a line. With 320 bytes per line and 25 lines on the display, a screen represents 8000 bytes of memory. A little later I'll show how these 8000 bytes are "summarized" in a screen memory of 1000 bytes.

Two colors normally appear within any given cell. When a bit is set to 0, the background color is displayed; when set to 1, the foreground color is shown. (We say "normally" because in multicolor mode, which we'll discuss later, a cell can contain up to four colors.) The control circuitry of the Commodore 64 takes care of mapping these bit settings to their corresponding screen locations and turning on the appropriate colors, so as programmers our only concern is the bit settings themselves.

I remember that face from somewhere . . .

Gather 'round and I'll tell you a tale of deceit and trickery entitled "The Bank-Switched ROM." It's about a shell game perpetrated on a poor unsuspecting computer, and W. C. Fields would have adored it.

A ROM is a Read-Only Memory, a special kind of memory that the computer can read but can't modify. Also, a ROM—unlike the RAM (Random-Access Memory), that you use for programs and data—doesn't lose its contents when the power is off. The Commodore 64 is full of ROMs for sundry purposes: they hold the machine's operating system (the "Ker-

nal"), the BASIC interpreter, input/output drivers that communicate with the outside world, the synthesizer, and—relevant to this tale— the character images. There are other ROMs as well, but that's another story.

An 8-bit microcomputer, which the Commodore 64 is, uses two bytes to indicate memory addresses. Since one byte represents any of 256 values, two of them together will represent that number squared, or 65,536. This means that the maximum memory size of the computer is 65,536 bytes, or 64K (one K is 1024), which is how much RAM the Commodore 64 has.

Well and good, but if the machine's memory addresses are all committed to RAM, that leaves no addresses free for the ROMs that hold all the computer's basic knowledge. After all, if the CPU is expected to read these ROMs and learn how to do its thing, it has to have a means of finding that information, and that's what addresses are for.

The shell game begins. Bank switching involves turning off a hunk of RAM and simultaneously turning on a ROM with the same addresses. The RAM isn't really powered down, but instead made inaccessible so that the CPU can see what's in the ROM that has those addresses. Afterwards, it bank-switches the ROM off and the RAM back on, and the RAM holds exactly the same information it had before all this deception began.

One of these ROMs holds all the images—8-byte blocks—that comprise the Commodore 64's cast of characters. It begins at address 53248 in ROM. There are 256 potential characters, one for each possible byte value, and since an image has 8 bytes, it takes $256 \times 8 = 2048$ bytes to hold the complete cast. That's 2K extending from 53248 through 55295. Back in Chapter 3 we monkeyed around with alternate character sets: Set 1 lives in this addressing range. Set 2 occupies another 2K between 55296 and 57343. The command POKE 53272,23 switched to Set 2 by telling the Commodore 64 that you wanted to access the character ROM's second half. POKE 53272,21 pointed it at the first part.

Notice, incidentally, that the POKE that controls the selection of character sets goes squarely into the very RAM that gets bank-switched out when it comes time to read the character ROM. If we had only one set of memory addresses in the range 53248–57343 and it held character images, the POKE would change one of the characters. But of course, there are two apartments at that address, and bank switching decides whether we go to the one upstairs or the one down.

Appendix A shows the screen display codes. These are numbers by which character images are known internally to the computer. Good old W, according to this chart, is 23. This means that the image for W is the 24th in the ROM table (the first being the zeroth). The computer, when presented with a character number 23, promptly multiplies this number by 8, for the eight bytes in each image, to find where Image 23 begins in the table. The answer is 184, so it adds 184 to the starting address of

the character set currently in use—usually Set 1 at 53248—to get the exact address of the image for character 23. It then bank-switches to the ROM and reads the eight bytes starting at 53248 + 184, and then bank-switches back to the RAM. And that's how the Commodore 64 gets the image for W.

Well, almost. We've deceived *you* by leaving out part of this story about deceiving the computer. The fellow who does the memory-to-screen management in the Commodore 64 is named VIC-II, and he lives in a ROM somewhere down in the innards. VIC is very, very busy, so he limits the scope of his view to one 16K hunk of memory at a time. Later I'll get into that in detail, when I talk about multiple-screen applications. For now, the implication of VIC's narrow view is that, if he can only see 16K at a time, the only place we could ever have screen memories with characters is in an area that encompasses the character ROM addresses, that is, above 49152, which is the top 16K of memory. But we know from the last chapter that screen memory is at 1024–2023. Where did the characters come from?

The plot thickens with deception number two. Sneaky elements within the Commodore 64 make VIC *think* the character ROM is within his field of view. When the power comes on, VIC is absolutely convinced that the character ROM starts at address 4096, which is within the low memory bank (addresses 0–16383) where he does business until directed otherwise. When VIC gets hold of a W and calculates its image address as 184, he adds that offset to 4096, which he thinks is the start of character ROM. But then, when he sends out for the image at that address, somebody along the line intervenes and converts it into the real address of the 24th image in character ROM, and then bank-switches to the ROM to get the image. VIC is consistently fooled by this, and since it works so well he has no reason not to be.

The automatic character-finder

Now it so happens—and here we get to the crux of this chapter—that VIC doesn't *have* to get his character images from the false address at 4096. He can get them from any of seven other addresses within the 16K bank he's working with, and when he does, those are *real* addresses. If VIC hasn't specified 4096 as the starting point of the character images, nobody meddles by converting the address and bank switching the ROM. Thus, he gets his images directly from RAM.

You can make him do this right now by typing the command

```
POKE 53272,(PEEK(53272) AND 240) OR 12
```

Bingo! Suddenly the characters on the screen change into a bunch of strange blobs. Why? Because VIC is getting his character images from

address 12288 and upward, and there aren't any character images there. What you see are the random bits that happen to be turned on in that part of memory right now. It's a dirty trick, but as we've already said, the images mean nothing to VIC. We can point him back to the character ROM with

```
POKE 53272,(PEEK(53272) AND 240) OR 4
```

If you made a typo doing that, there will still be garbage on the screen, and in that case, retype it. When you've got it right, the letters will reappear as though nothing were ever amiss.

Had there been character images stored from 12288 upward VIC would have displayed them on the screen instead of garbage. More on placing images there in a moment, but first let's talk about address 53272. This important byte contains two pointers that tell VIC where his screen is, and where to find the character images. Mess up this byte and you've bought the farm. The only recovery is to hold down RUN/STOP while you press RESTORE. Among other things, that resets the screen pointer to 1024 and the character pointer to ROM, which is its condition at power-up.

When we get to multiple screens I'll discuss this byte again. Its upper four bits govern the location of the screen memory. For now, simply keep in mind that you must always protect that upper nibble from accidental alteration because of something you're doing with the character pointer. The lower nibble holds the character pointer in its upper three bits. VIC ignores the low-order bit. Whether it's 1 or 0 makes no difference.

VIC can get his character images from any of eight addresses, of which seven are in real RAM memory. These addresses correspond to 2K boundaries, progressing 0, 2048, 4096, 6144 . . . 14436. Location 4096 is the phantom address for character ROM, so we can't expect VIC to find characters if we put them into the real 4096. Also, you shouldn't use address 0 in the low memory bank (below 16384) where we normally operate, because the Kernal uses that area for system status information, and 2048 and 6144 are out because they belong to BASIC. These constraints don't apply in the other banks, as we'll discover when we study multiple-screen operations. This leaves four areas for character memory in the upper 8K of bank 0.

Figure 4.1 shows the character memory locations, the bit settings related to them (X's represent bits not relevant to the character pointer), and the POKE values for those settings.

You can develop the value of PV mathematically from the actual address of the character memory with the statement

```
PV = (CM - (BN * 16384)) / 1024
```

where CM is the character memory address and BN is the number of the memory bank. If character memory is at location 12288 in bank 0, this

Command to change the pointer in BASIC:
POKE 53272, (PEEK(53272) AND 240) OR PV

Location	*Bit setting*	*PV*
0	X X X X 0 0 0 X	0 (Note 1)
2048	X X X X 0 0 1 X	2 (Note 2)
4096	X X X X 0 1 0 X	4 (Note 3)
6144	X X X X 0 1 1 X	6 (Note 2)
8192	X X X X 1 0 0 X	8
10240	X X X X 1 0 1 X	10
12288	X X X X 1 1 0 X	12
14336	X X X X 1 1 1 X	14

Notes:
1. Do not use in bank 0. Belongs to Kernal.
2. Do not use in bank 0. Belongs to BASIC.
3. Phantom (default) address for character ROM.

Figure 4.1 Selecting character memory locations

works out as:

```
PV = (12288 - (0 * 16384)) / 1024
PV = 12
```

which is given for that address in Fig. 4.1. When you are in bank 2 and use 8192 for your character memory, CM is 8192 plus the bank address 32768, for an actual address of 40960. Thus,

```
PV = (40960 - (2 * 16384)) / 1024
PV = (40960 - 32768) / 1024 = 8
```

and that, too, is correct for the location 8192 bytes from the start of the bank.

The instant the POKE is executed, VIC begins getting his character images from the new location, and if the images are different for codes already on the screen, the display changes to the new images. That's why, when we did the POKE a while ago, the entire screen was affected. You can restore normal (character ROM) operation by POKEing the PV of 4.

Gittin' them characters

If you want to add your own special characters or modify existing ones, you can't do it while the images remain in ROM. That's because ROM is—consistent with its name—a read-only memory, and modifications of necessity alter images by changing their bit patterns. Consequently, you'll have to move the whole character set (or the parts of it that you want)

out of the ROM and into RAM. Once it's in RAM, you can make changes to your heart's content, and then POKE 53272 to point VIC-II at your new RAM character memory.

Copying a block of memory from one place to another is simple. You just set up a FOR/NEXT loop that POKEs into the destination what it PEEKed from the source. Copying from the character ROM, however, involves some extra precautions and steps.

The address range of the character ROM coincides with the area of memory where the Commodore 64 keeps track of peripheral devices, such as printers, data storage units, and modems. This is also where it maintains status information concerning the keyboard. Such devices generate *interrupts*—requests for service from the computer—by setting bits in this area. We thus have to make sure no interrupts occur while the ROM is switched on during the copy operation. The ROM won't be damaged, but the interrupts will go unanswered and other bizarre things can happen. This calls for a planned procedure that we can incorporate into a program that copies the character ROM into RAM:

1. Disable interrupts with POKE 56334,PEEK(56334) AND 254
2. Bank-switch the ROM in with POKE 1,PEEK(1) AND 251
3. Copy from ROM to RAM
4. Bank-switch the ROM out with POKE 1,PEEK(1) OR 4
5. Enable interrupts with POKE 56334,PEEK(56334) OR 1

During the copy operation, the keyboard is inoperative, as are all computer-attached devices, including the display screen. For this reason, you must be absolutely certain that your copy loop works right, because the only way to recover from a hang-up is to turn the computer off and back on.

Let's try it by copying the entire Set 1 from the ROM into the memory area starting at 12288. Figure 4.2 shows the program to enter into your computer.

```
NEW
100 REM  **  ROM COPY
110 S = 53248: D = 12288
120 POKE 56334, PEEK(56334) AND 254
130 POKE 1, PEEK(1) AND 251
140 FOR M = 0 TO 2047
150 POKE D+M, PEEK(S+M)
160 NEXT M
170 POKE 56334, PEEK(56334) OR 1
180 POKE 1, PEEK(1) OR 4
190 END
RUN
```

Figure 4.2 Copying character ROM Set 1 into RAM

It takes about 15 seconds for this program to complete, so don't have an anxiety attack unless it's obviously taking far longer than that. If the copy operation is successful, we should be able to switch the character pointer in 53272 and see absolutely no change on the screen. Type the command

```
POKE 53272, (PEEK(53272) AND 240) OR 12
```

If the characters on the screen changed in any way, the program didn't run right. This command could have been included in the program at line 190.

You now have a complete copy of character Set 1 in RAM, and the VIC-II is using it for the display instead of ROM. Since you can't see any difference, you might doubt that the program accomplished anything. To put your doubts to rest, and also to show how to do a partial copy from ROM to RAM, let's replace the letter W in RAM with A from ROM.

The same program we just ran will do this with only three changes. The source letter A is the second character in ROM, so we have to change the value of S in line 110 to 53248 + 8. The letter W is the 23rd image in both ROM and our RAM copy, which is 184 bytes from the start, so change D in line 110 to 12288 + 184. Finally, we're copying only the eight bytes of this one character image, so change the high value of the FOR/NEXT loop in line 140 from 2047 to 7.

Before you run the program, type a bunch of W's. BASIC will scold you for making a syntax error, but no matter. While the program runs (remember, VIC is now looking at RAM for his character images), watch what happens to those W's.

They all changed to A's! Not only that, but when you hit the W key, it produces an A. If you type AWAY IN THE WOODS, you get AAAY IN THE AOODS. There's nothing wrong with the computer; we've just changed the image produced by the W key.

Let's fix the poor busted image and quell your fears that I've wrecked your computer. Change the value of S in line 110 to read 53248 + 184 and run the program again. *Voilà!* Before your very eyes, the A's change into perfect W's.

A character-building experience

I'm an American and for me, having a whole key assigned to the British pound sterling symbol wastes valuable space. Suppose I want it to be a cent sign, which the Commodore 64 doesn't have. (British readers can use the exercise that follows to make something more useful out of the dollar sign.)

According to Appendix A, the pound sterling sign is screen code 28, which means we'll find its image starting at 12288 + 224, or 12512. *Hint:*

If you dislike hunting through tables as much as I do, you can find the screen code for a character with the following procedure:

1. Hold down SHIFT and hit CLR/HOME to place the cursor in the top left corner of the screen.
2. Type the character whose code you want to know.
3. Hit RETURN (you'll get a syntax error, but so what?).
4. Type PRINT PEEK(1024).
5. The number that appears is the screen code of the character.

Anyway, back to our character-building exercise. A character image is, as we've said, an 8 × 8 dot matrix in which the 1 bits define its shape. The 1 bits are set by POKEing values that equal the sum of the 1 bits' positions.

To make a character image, use some of that graph paper you got for planning graphics screens. Outline on it an area eight squares high by eight wide, and within that area sketch out the shape of the image you want to construct. (It's wise, by the way, to make all vertical lines two dots wide to avoid color distortion.) Use a lead pencil so you can make erasures. Once you've worked out a satisfactory shape, make X's in the squares that approximate the sketch.

Now you're ready to define the character numerically. A calculator helps, because, for each row, you have to add up the sum of the bit positions in which 1 bits appear. Remember, going from left to right the values of bit positions are 128, 64, 32, 16, 8, 4, 2, and 1. If it helps, label the columns. To the right, jot down each byte's value, as we have in Fig. 4.3.

	128	64	32	16	8	4	2	1	
0	·	·	·	X	X	·	·	·	24
1	·	X	X	X	X	X	X	·	126
2	X	X	·	X	X	·	X	X	219
3	X	X	·	X	X	·	·	·	216
4	X	X	·	X	X	·	·	·	216
5	X	X	·	X	X	·	X	X	219
6	·	X	X	X	X	X	X	·	126
7	·	·	·	X	X	·	·	·	24

Figure 4.3 Building a character image for the cent sign

The numbers on the left represent the relative byte number within the image. Across the top are the values of each bit position, and on the right are the sums of the 1 bits in each byte. We can now use these sums to make a new character image which, in this case, overlays the pound sterling sign at 12512.

The program in Fig. 4.4 is a general-purpose imagemaker. You can use it to make any new character image simply by changing the destination address in line 110 and the DATA statement to include the byte values developed for the image. Before you run this program, type a pound sterling sign so that you can see it change into a cent sign before your eyes.

```
100 REM  ** BUILDS A CHARACTER IMAGE IN RAM AT LOCATION "D"
110 D = 12512
120 FOR M = 0 TO 7
130 READ B
140 POKE D + M, B
150 NEXT M
160 END
170 DATA 24, 126, 219, 216, 216, 219, 126, 24
```

Figure 4.4 A character-building program

Unfortunately, character changes of this sort do not last. When the computer is turned off, the change is lost, and there is no way to store it in ROM as a permanent part of the machine's character set. Therefore character-building is only temporary as far as the Commodore 64 is concerned.

One solution to the perishability of custom character sets is available to those who have a disk drive. You can construct a modified character set in RAM and then make a program to read it with PEEKs and write it to a disk file. The companion program to this is executed each time you power up the system: read the character set from disk, POKE it into the desired memory area, and switch the pointer in 53272 to it.

Colorful characters

There are several ways to control the colors of the characters on the display. The simplest is by a direct keyboard command, holding down either the CTRL or the C= key and typing any number between 1 and 8. Thereafter, all characters on the display appear in the color you set through that command, until you either change it or reset the display by holding down RUN/STOP while hitting the RESTORE key. Fig. 3.9 and the accompanying discussion showed how to perform these keyboard commands from a program with PRINT CHR$ statements. Also, as you know from Chapter 3, if you POKE characters into screen memory, you can control their colors with the corresponding location in color memory. Here I'll discuss two other ways to alter character colors.

The first of these involves changing the color of things already on the screen. To do this, you simply pass through color memory for the area corresponding to the cells you want to change and POKE the code for the desired color. Whatever is on the screen (even though you didn't POKE screen codes to put it there) immediately changes to the new color.

As an example, hold down SHIFT and hit CLR/HOME to clear the screen, and set the text color to white and the screen background to black (POKE 53281,0). Now enter the following command:

```
FOR X = 0 TO 319: PRINT "X";: NEXT X
```

This produces eight rows of X's near the top of the screen (right under the command line you entered). They're all white against a black background. Now type the command

```
FOR X = 0 TO 319: POKE 55296 + X, 2: NEXT X
```

This puts the code for red into the first 320 bytes of color memory, which correspond to the top eight lines of the display. And behold, the command line and all the X's except the last row turn red. Why not the last row of X's? Because it's on the ninth line of the display. It therefore remains white, as does the other text below it. We have changed the color memory only behind the first eight rows.

Now suppose we want to change the middle ten X's in the still-white row. They begin at column 15 and end at column 24. As we know from XY coordinate plotting, we can find that location with (Y * 40) + X. In this case, X is 15 and Y is 8, so the address for the first X in the group is 335 bytes into color memory. We can modify these X's, then, with the command

```
FOR X = 0 TO 10: POKE 55296 + 335 + X, 3: NEXT X
```

Sure enough, the middle X's in that row become cyan (greenish blue).

It would seem, since we've changed color memory, that as the text scrolls upward, the lines would change as they moved into the spaces "in front of" the color memory; in other words, that the line of white X's with cyan in the middle, as it shifted upward, would turn red, and when the word POKE farther down arrived where the cyan X's are now, it would turn cyan. Right? Let's try it and see. Hit RETURN several times until the cursor gets to the bottom of the screen and starts pushing lines up off the top and watch what happens.

Hey! The colors stay with the characters they belong to. That's because, as part of the scrolling mechanism, color memory is shifted forward 40 bytes each time a line vanishes off the top of the screen. That way, once you've assigned a color to a location on the display, whatever is there at the time of assignment retains that color until it leaves the screen.

It does *not* work, incidentally, to alter the color memory and then

PRINT characters into the corresponding screen locations. Color changes can only be made after the text is already visible.

The second (exotic) technique is what Commodore calls *extended background color mode*, which enables you to select any of four different colors for the background field of a character. For instance, you might have white letters on a black screen, but the background field of the character itself could be green or red or gray. In this mode, the 1 bits of the image take their color from color memory, and the Ø bits assume any of the four colors you select.

Character background colors are specified by POKEing color numbers (Ø–15) into the four consecutive memory addresses starting at 53281. You'll recall that 53281 itself determines the screen's color. In this case, it also determines one of the possible background field colors for a character, thus giving the character an "invisible" background that matches the screen. The other three memory locations (53282–53284) hold any color numbers you want. Obviously, you have to exercise a little care in selecting background colors that contrast with the lettering; red characters don't show up against a red background.

The whole process, in fact, requires planning and setting up. You have to load color memory with the character color and the four memory locations 53281–53284 with the background color options. Then, the way you select a background color for a specific character is by setting the uppermost two bits in the screen code, as shown in Fig. 4.5.

Background number	*Memory location*	*Character code bit 7*	*bit 6*	*Screen code range*	*Add to base*
Ø	53281	Ø	Ø	Ø – 63	Ø
1	53282	Ø	1	64 – 127	64
2	53283	1	Ø	128 – 191	128
3	53284	1	1	192 – 255	192

Figure 4.5 Extended color background selection

To see how to use this chart, let's say you want to display the letter A at some location on the screen called XY. The screen code for A is 1, as Appendix A tells us. Background number Ø (in 53281) is the color of the screen. To display an A with the same background as the screen, you would instruct the computer to

```
POKE XY, 1
```

and an A would appear with no discernible background field. But now let's say you want the character to have a surrounding field of background color 1. In that case, you have to set bits 7 and 6 of the screen code to Ø1. Since bit 6 has the binary value 64, you can do this by adding 64 (the

"add to base" value) to the screen code, giving 65 in this case. When you POKE this number into XY, the A shows up with its image in one color and its background in another against a screen of a third color. Similarly, to surround the character with the color indicated in background number 2 (address 53283), you must set the high-order bits to 10, which is done by adding 128 to the screen code; for background number 3, add 192.

The consequence of committing the two high-order bits of the screen code to the background color selection is that you can only use the first 64 screen codes. This gives you all the standard alphanumerics, punctuation marks, and math symbols, and none of the graphics characters. Extended background mode, then, is useful chiefly for displaying highlighted messages that you want to stand out from the rest of your text display.

The computer, lacking a glimmer of intuition, needs to be told that you want to enter or leave extended background mode. To turn the mode on, use the instruction

```
POKE 53265, PEEK(53265) OR 64
```

To turn extended background mode off and reenter normal display operations, use

```
POKE 53265, PEEK(53265) AND 191
```

Note that if you have graphics characters on the screen when you enter extended background mode, they will instantly change to text letters, numbers, or symbols. Also, if you switch into extended background mode, display a message using the mode, and then switch back to normal operation, the benefit of extended mode vanishes immediately (the message using it becomes ordinary text or garbles into a mess of graphics characters). *Extended background mode and character graphics do not peacefully coexist, and extended background mode must remain in effect as long as any text using it stays on the screen.*

Figure 4.6 is a demonstration program that shows extended background mode in action by displaying four rows of A's with two empty rows between each. You can exit the program and restore normal screen operation by striking the SPACE bar.

```
NEW
100 REM  **  EBC DEMO
110 PRINT CHR$(147):PRINT:PRINT
120 PRINT TAB(5) "EXTENDED BACKGROUND COLOR DEMO"
130 REM  **  WHITE CHARACTERS INTO COLOR MEM
140 FOR X = 0 TO 999: POKE 55296 + X, 1: NEXT X
150 REM  **  SET UP COLOR SELECTIONS
```

```
160 POKE 53281, 0        :REM BLACK SCREEN
170 POKE 53282, 2        :REM RED IS COLOR 1
180 POKE 53283, 6        :REM BLUE IS COLOR 2
190 POKE 53284, 12       :REM MED GRAY IS COLOR 3
200 REM  **  ENTER EBC MODE
210 POKE 53265, PEEK(53265) OR 64
220 REM  **  DEMO DISPLAY LOOP
230 Y = 120
240 FOR BC = 0 TO 192 STEP 64
250 FOR X = 0 TO 39
260 POKE 1024 + X + Y, 1 + BC
270 NEXT X
280 Y = Y + 120
290 NEXT BC
300 REM  **  FREEZE DISPLAY UNTIL KBD SIGNAL
310 GET X$: IF X$ = "" THEN 310
320 PRINT CHR$(147);
330 POKE 53265, PEEK(53265) AND 191
340 END
RUN
```

Figure 4.6 Demonstration of extended background color mode

The operative statement of this program is line 260. The address it POKEs into is the sum of 1024 (screen memory) plus the column (X) plus the row (Y), and the value it POKEs is the sum of the screen code for A plus the value BC, which determines the setting of the upper two bits of the screen code. Note that BC increases by 64 for each repetition of the outer loop, and that Y—the row—increases by 120 for each BC, advancing the display three rows.

Extended background color mode seems terrific, but it is so touchy and fraught with limitations that it is difficult to conceive of many uses for it, and except that it is a complex subject, it probably doesn't deserve as much attention as we've given it.

ASCII versus the screen codes

Among the things you could ask the Commodore 64's design engineer if you ever meet him, I hope you'll include the question: "Why does the machine have different ASCII and screen codes?" Surely there is some good reason, but it won't be found in these pages. Instead, I'll be pragmatic by dealing with the facts.

Appendix B shows the ASCII codes for the symbols used by the Commodore 64. ASCII, which stands for the American Standard Code for Information Interchange and is pronounced "askey," provides a means of representing letters, numbers, and symbols that is universally observed by microcomputer manufacturers. In ASCII, the letter A is always represented by the bit pattern that has the value 65, a comma is 44, and so forth. There are 256 possible values in an 8-bit byte, and ASCII establishes the relationship of written symbols to bit patterns for everything from 32 (SPACE) to 90 (Z). The rest are available for whatever purposes the machine's manufacturer sees fit. Commodore has seen fit to use most of them for graphics symbols, color numbers, and other control functions.

As you type programs and instructions and data into the Commodore 64, your keystrokes are converted into ASCII and stored in the computer's memory. The computer doesn't recognize A as a visual pattern, but it does recognize 65 as a data object, as I pointed out earlier in this chapter. Therefore, it works on 65 and leaves it to some other part of the computer to make visual sense out of 65.

If you look up code 65 in Appendix A (screen codes), however, you find that it's an ace of spades, not an A. The letter A in the screen codes is a 1. The screen codes are what we POKE into screen memory to get a character on the display; the ASCII codes are what we PRINT or manipulate internally or whatever. In fact, what the screen codes represent is the order of images in the character generator ROM, and we can translate any screen code into its relative ROM address if we multiply the screen code by 8.

Okay, you say, then to convert an ASCII value into a screen code, all you have to do is subtract 64, the difference between the ASCII A and the screen code A. But what if you have the digit 2, which has the ASCII code 50? Subtract 64 and you get a negative number, and there aren't any negative screen codes. Aha! For that, we discover, the screen code is the same as the ASCII code. And then what about the ace of spades? In ASCII it's 97 and in the screen codes it's 65, a difference of 32. What a mess!

Fortunately, however, there's a relatively easy conversion from one code to the other. You might be wondering why we should even care. The answer is that sometimes, if you've got a busy game or elaborate graphics that depend on a lot of POKEs, it's easier to POKE text than to space around on the screen with a lot of PRINTs, as well as being more efficient programming to use a consistent means for getting things onto the display. So what is this easy conversion? (the author asked rhetorically).

Well, (he continued), there are mathematical relationships among *blocks* of ASCII and screen codes. If you know the ASCII value, you can determine the block it belongs to and adjust that value to compute the

corresponding screen code. It happens that BASIC, anticipating this very problem, provides the function ASC to return the value of a character. As an example, type

```
PRINT ASC("A")
```

The computer responds with 65, the ASCII value for A. ASC has a counterpart function CHR$, which produces the character associated with a numeric value. Type the command

```
PRINT CHR$(65)
```

and you get an A. The one has the opposite effect of the other.

ASCII values	*Screen codes*	*Factor*
0 – 31	(none)	(none)
32 – 63	32 – 63	0
64 – 95	0 – 31	−64
96 – 127	64 – 95	−32
128 – 159	(none)	(none)
160 – 191	96 – 127	−64

Figure 4.7 Conversion of ASCII to screen codes

The blocks and conversion factors are shown in Fig. 4.7. Apply this table to the code conversion as follows:

1. Use the ASC function to obtain the ASCII value of the character you want to POKE.
2. Isolate it to the block of values and apply the factor shown to find the screen code.
3. If there is no factor, ignore the character (you can't convert it to a screen code).
4. POKE the resulting code into screen memory.

Before we reduce this algorithm to BASIC programming statements, we need to consider one other problem. Suppose you have 27 characters in a row that you want to place on the screen. If you had to repeat the instructions 27 times, once for each character, it would be easier to PRINT them. Instead, you can use a subroutine and provide it with a string variable (CS$) that contains the characters you want to display and the X and Y coordinates where the string is to begin. The subroutine can then pick apart the string and process one character at a time in a loop.

BASIC has the string function MID$ that extracts individual characters. MID$ takes three arguments (an *argument* is a piece of information a function needs to do its job), which are, in order, the name of

the string, the position within the string, and the number of characters to extract. For the string "MY NAME IS JOE", if you execute the statement

N$ = MID$(CS$, 12, 3)

N$ will be JOE. To help with the picking apart, BASIC also furnishes the LEN function, which returns the number of characters in the string. In this case, the statement

LS = LEN(CS$)

gives LS the value 15, which is how many characters there are in the string called CS$. Figure 4.8 puts these functions to work.

```
NEW
8000 REM  ** CONVERT AND DISPLAY ASCII TEXT
8010 LS = LEN(CS$)       :REM  LENGTH OF STRING
8020 FOR P = 1 TO LS
8030 CH$ = MID$(CS$, P, 1): AV = ASC(CH$)
8040 REM  ** CONVERT TO SCREEN CODE
8050 IF AV <= 31 THEN 8150
8060 IF AV <= 63 THEN SC = AV: GOTO 8110
8070 IF AV <= 95 THEN SC = AV - 64: GOTO 8110
8080 IF AV <= 127 THEN SC = AV - 32: GOTO 8110
8090 IF AV <= 159 THEN 8150
8100 SC = AV - 64
8110 REM  ** POKE INTO SCREEN MEMORY
8120 SL = X + Y * 40
8130 POKE 1024 + SL, SC
8140 X = X + 1
8150 NEXT P
8160 RETURN
```

Figure 4.8 Code conversion and display subroutine

The statements should be self-explanatory based on the preceding discussion. Besides having the string CS$ and the X and Y coordinates of the starting character cell provided to it, the subroutine assumes that color memory is already set up for the area where the text is to appear.

Let's test the subroutine with a program that calls it. The program in Fig. 4.9 prints "SAMPLE STRING" in the center of the screen (at row 12, column 13).

```
100 REM  ** TEST PROGRAM
110 PRINT CHR$(147);
120 POKE 53280, 0: POKE 53281, 0
130 FOR P = 0 TO 999: POKE 55296 + P, 1: NEXT P
140 CS$ = "SAMPLE STRING": X = 12: Y = 13
150 GOSUB 8000
160 END
RUN
```

Figure 4.9 Test of code conversion subroutine

This program begins by clearing the screen. There is then an alarming delay during which nothing seems to happen; in reality the color memory is being loaded with white (line 130). About the time you become convinced that the computer is hung up, SAMPLE STRING pops reassuringly into the middle of the screen. In a real-life situation we'd probably shorten the delay to an eyewink by using the machine-language routine, presented in the next chapter, to load color memory.

The Commodore 64 gives the programmer—you—more control over its character set than almost any other computer. No computer is really smart, and people are (or should be) always able to outwit it. This chapter has provided you with the means for doing so, and for exploiting the rich possibilities of its cast of characters and your own creativity.

CHAPTER 5

HIGH RESOLUTIONS TO THE 64,000 DOT QUESTION

Up until now, we've made graphics by combining the shapes that Commodore provided with their computer. That gives us a lot of flexibility, but it limits the horizons of our creativity. What if we want to do things such as plot curves and circles, make detailed pictures (less primitive than Grandma Moses'), and draw complex diagrams with lines running hither and yon? For those kinds of applications, we need High-Resolution Graphics (HRG).

With character graphics, we conceived of the screen as a grid measuring 40 columns wide (the X dimension) by 25 rows deep (the Y dimension) for a total of 1000 positions, any one of which could be described by its X and Y coordinates. We told the Commodore 64 where to find the byte corresponding to the coordinates with the formula:

X + (Y * 40) + base address

We can think of the screen in the same XY fashion with HRG, but we have to rescale the dimensions. A character cell is a square 8 dots wide by 8 high, so across the 40 columns of the screen there are $8 \times 40 = 320$ dots, and across 25 rows there are $8 \times 25 = 200$ dots. HRG controls each dot individually, making the X dimension grow from 40 to 320 and the Y dimension from 25 to 200, for a total of $320 \times 200 = 64{,}000$ positions on the screen, each of which has an X and Y coordinate.

To display a figure at a given XY location in character mode, we POKE into the corresponding screen memory address a number that refers to the desired character. The Commodore 64 uses this number to index the

character generator ROM and find the 8 × 8 dot pattern required to form the shape on the screen. We thus have control of the cells, but not of the dots within a cell.

In HRG, on the other hand, we don't have access to the character ROM. Instead, we control *every* dot on the screen as an individual entity. For each dot there is a corresponding bit in memory. If that bit is "off" (set to 0), the dot displays the screen background color; if "on" (set to 1), the dot takes the foreground color. To illustrate this, imagine an area of 8 × 8 dots occupied by the letter A. In the image portion of Fig. 5.1, a period represents a dot of background color and an X represents a foreground dot. The bit pattern portion of the figure shows the eight corresponding bytes to effect this shape and their values.

Image	*Bit pattern*	*Value*
. . . X X . . .	0 0 0 1 1 0 0 0	24
. . X X X X . .	0 0 1 1 1 1 0 0	60
. X X . . X X .	0 1 1 0 0 1 1 0	102
. X X X X X X .	0 1 1 1 1 1 1 0	126
. X X . . X X .	0 1 1 0 0 1 1 0	102
. X X . . X X .	0 1 1 0 0 1 1 0	102
. X X . . X X .	0 1 1 0 0 1 1 0	102

Figure 5.1 Relationship of display dots to memory bits

With 64,000 dots on the screen, we need 64,000 bits to represent them, and with 8 bits in a byte that comes to 64,000/8 = 8000 bytes, which is a whole bunch more memory than we needed for a character graphics screen. Well, you might ask, so what? The very name of the Commodore 64 suggests that there are great gobs of paid-for memory out there that we haven't touched yet, so let's just take another 7000 bytes. In fact, that's what we're going to do, but it's not quite as simple as that.

Screen memory for character mode lies from addresses 1024 to 2023. The memory beginning at 2024, however, is already committed as the work area for BASIC programs, so if you grab 7000 more bytes above screen memory and start messing with bits, you'll clobber your own program and crash the computer. To get around this obstacle, you have to change the place where the machine looks for its screen image. Conceptually, the screen is a window that shows part of memory. How much it shows depends on whether the machine is in character mode or in HRG mode (1000 or 8000 bytes, respectively). The Commodore 64 can move this window from place to place through program instructions that Chapter 7 discusses in detail. For now, we'll simply shift the window to a place well above the BASIC program area, where there are 8000 uncommitted bytes. This area begins at address 8192. We can tell the computer to shift

shift the display window to 8192 with

```
POKE 53272, PEEK(53272) OR 8
```

Don't key the command into your computer, though, until we're ready to use that screen. If you already did, or if at any time you want to restore normal operation (e.g., at the end of a program that used 8192 for screen memory), you can undo the instruction above with the command

```
POKE 53272, PEEK(53272) AND 247
```

These instructions control the bits that point to screen memory.

After the window is shifted, the Commodore 64 looks there for its screen image but, thinking it's still in character graphics mode, it looks at only 1000 bytes. You have to tell the computer to go into HRG mode with

```
POKE 53265, PEEK(53265) OR 32
```

With bit 5 of this byte turned on, the machine knows it has to look at 8000 bytes instead of 1000 for screen information, and to interpret what it sees as high-resolution graphics. To revert to normal character mode, use the command

```
POKE 53265, PEEK(53265) AND 223
```

and the HRG bit turns off.

A curious thing happens when you switch to HRG mode. Remember our old friends, screen memory at 1024 and color memory at 55296? Well, in HRG, what used to be screen memory becomes *color* memory. *High-resolution graphics looks to the 1000 bytes starting at 1024 for colors.* Moreover, color memory still uses the concept of 8×8 dot cells, but it lets you specify two different colors for each cell.

In character mode, color memory indicated the cell's color, and in the corresponding screen memory cell a 1 bit took this color and a 0 bit had the background color. In HRG, on the other hand, the upper four bits of the color memory cell indicate the foreground (1-bit) color and the lower four bits give the color for 0 bits.

To set up a cell with a white foreground on a brown background, multiply the foreground color number by 16 to shift it to the upper nibble, then add the background color number. According to Fig. 3.9, white is 1 and brown is 9, so the instruction reads

```
POKE CM, (1 * 16) + 9
```

where CM is the address of the color memory cell where you want this combination. It is, of course, necessary to set up a color combination in each cell of color memory, or else you'll end up with unpredictable results.

When you go into HRG mode and access a different screen memory, your program is likely to find random garbage there. Consequently, you must not only initialize color memory for the combination(s) you want, but you must also pass through all 8000 bytes of screen memory and set them to 0s to make sure the 1 bits are turned off. Our first exercises will do this with BASIC to familiarize you with the idea, but be warned: it's an agonizingly slow process. Later we'll do some magic to speed it up to the twinkle of an eye.

We now know enough about HRG mode to develop a standard process for starting and ending programs using it. The steps are:

1. Move the window to a different screen memory.
2. Switch to HRG mode.
3. Initialize color memory for the combination(s) you want.
4. Fill screen memory with 8000 zero bytes.
5. (Insert your actual program here.)
6. Freeze the display until you interrupt it.
7. Restore character mode.
8. Return the window to normal screen memory.

The order of 1 and 2 and of 7 and 8 doesn't matter, as long as you group them at the start and end. We'll talk about the "interruptable freeze" (item 6) later.

Relating XY coordinates to dots on the screen

Each of the screen's 64,000 dots, arranged in a grid 320 dots wide by 200 dots deep, lies at the intersection of a column (X) and a row (Y). The upper left corner of the screen has coordinates X = 0, Y = 0; the lower right corner has coordinates X = 319, Y = 199. Thus, the range of X is 0–319 and the range of Y is 0–199. The problem is how to relate a coordinate pair to one bit somewhere among 64,000 and turn it on. For that we need formulas that map XY coordinates to the specific bits of each cell. It gets a little complicated, but it's not difficult once the concepts are firmly in mind.

HRG cells have the same structure as character cells, which we saw in the character-building experience in Chapter 4. The bytes are stacked with byte 0 on top and byte 7 on the bottom. Byte 8 lies immediately to the right of byte 0 and is, in fact, byte 0 of the next cell (although in HRG we don't start renumbering with each cell). Figure 5.2 is a conceptual diagram.

0	8	16	312
1	9	17	313
2	10	18	314
3	11	19	315
4	12	20	316
5	13	21	317
6	14	22	318
7	15	23	319

Figure 5.2 Arrangement of bytes in high-resolution graphics

Carrying this a little further, byte 320 is at the upper left corner of the second row of character cells (directly under byte 7) and the pattern continues through 8000 bytes.

The mapping process narrows an XY coordinate pair down to the row in which it occurs, then to the character cell, then to the byte within the cell, and finally to the specific bit. The starting address of the screen memory must also be added in to get its absolute location. Let's take the case of X = 315, Y = 199, a position very close to the lower right corner of the screen. To find the cell's row, calculate

```
ROW = INT(Y / 8)
```

In this case Y = 199, so Y / 8 = 24.875. The INT function discards the fractional remainder, so INT(199 / 8) = 24, and indeed, the bottom row of the screen is row 24. In character graphics we adjusted the Y coordinate for the fact that there are 40 cells in a row. In HRG, there are instead 320 bytes in a row, so to find the number of bytes from the start of screen memory to this Y coordinate, multiply the row by 320.

Next we compute the cell within that row. Since the cell is horizontally placed, use the X coordinate to find it with

```
CELL = INT(X / 8)
```

Here X = 315, so INT(315 / 8) = 39. There are 8 bytes per cell, so to compute the number of bytes from the start of the row to the cell, multiply this result by 8.

Trickery with logical operations finds the byte number within the cell. There's a long boring mathematical explanation that I'll bypass by simply pointing out that the low three bits of the Y coordinate always contain the byte number within the character cell. All we have to do is isolate them with

```
BYTE = Y AND 7
```

Instead of doing these steps separately, in programs we can combine

them into the single instruction

```
BA = (INT(Y/8) * 320) + (INT(X/8) * 8) + (Y AND 7) + 8192
```

This equation will infallibly produce the screen memory address for the byte where the X and Y coordinates cross.

So how do we find the bit? Again, let's not get bogged down in the arcane mathematics of computer science. I'll just give you the instruction and you can take my word for it (the computer will, too, because it works). The bit number is found by

```
BP = 7 - (X AND 7)
```

Computers don't think of bits in terms of their numbers, however, but instead regard them as a progression of powers of 2 working from the right to the left. Thus, the bit at BP is set by ORing a number raised to the power of BP, written as 2^BP in BASIC.

All of this discussion leads to an almost disappointingly short routine that we can use as a standard means for translating XY coordinates into dots on the screen in HRG programs:

```
BA = (INT(Y / 8) * 320) + (INT(X / 8) * 8) + (Y AND 7) + 8192
BP = 7 - (X AND 7)
POKE BA, (PEEK(BA) OR 2^BP)
```

The skeletal high-resolution graphics program

It's often useful, when dealing with a set of complex tasks, to develop a standard skeletal program that contains the features we'll always need. That way, the support functions are already in place and we can concentrate on the problem itself rather than frittering away our energies reinventing the wheel.

Figure 5.3 is the basis for two such skeletal programs that this book will develop for standard and multicolor high-resolution graphics. We will use it "as is" for the next few exercises in HRG, and build upon it as we progress.

```
NEW
10 POKE 53265, PEEK(53265) OR 32
20 POKE 53272, PEEK(53272) OR 8
30 FOR X = 8192 TO 16191: POKE X,0: NEXT X
40 FOR X = 1024 TO 2023: POKE X,(1*16)+9: NEXT X
```

(Continued)

```
50 GOTO 100
60 REM  ** SET BIT FOR XY COORDS
65 IF X < 0 OR X > 319 THEN 90
70 IF Y < 0 OR Y > 199 THEN 90
75 BA = (INT(Y / 8) * 320) + (INT(X / 8) * 8)
   + (Y AND 7) + 8192
80 BP = 7 - (X AND 7)
85 POKE BA, (PEEK(BA) OR 2^BP)
90 RETURN
100 REM  ** PROGRAM STARTS HERE
9960 GET X$: IF X$ = "" THEN 9960
9970 POKE 53265, PEEK(53265) AND 223
9980 POKE 53272, PEEK(53272) AND 247
9990 PRINT CHR$(147): END
```

Figure 5.3 Skeletal HRG program

This program is structured so that you load it from a disk or tape, then begin typing your program at line 110. BASIC will automatically insert your lines into the program at that point. The main rule is that you must call your X and Y coordinates by those variable names. To turn on the dot at an XY location, specify the values of X and Y and issue the instruction GOSUB 60. The subroutine automatically ignores any coordinates that are off the screen (lines 65 and 70), thus protecting other memory areas from wild and irresponsible calculations.

Consider line 9960 for a moment. This is the "interruptable freeze" mentioned earlier. It stops action and locks up the computer awaiting an interrupt from the keyboard. Striking any character-generating key causes an interrupt, whereupon the program continues execution by restoring normal operation.

Most of the rest of the programs in this chapter are "inserts" at line 110 that require this skeleton to be in memory, and that show how to use it to make graphics.

Making lines

The simplest graphics figures are straight lines, so it's appropriate to start our adventures in graphics with

```
110 Y = 100
120  FOR X=0 TO 319: GOSUB 60: NEXT X
RUN
```

As soon as you finish typing RUN, the display changes dramatically, switching to high-resolution mode and the new screen memory. But then,

strangely, nothing much seems to happen for about 25 seconds, although you might see some garbage mysteriously disappear. This is the loop at line 30 setting all memory locations to zero, and it takes about one second per row of cells. Later we'll do away with this long pause, but right now think about what the program is doing. At the end of that clearing process, the screen turns brown and a white line begins cutting from left to right across the center of the screen. It is at Y coordinate 100 (see line 110), and the X coordinate increases from 0 (left edge) to 319 (right edge). When the line is complete, the display holds steady. It remains so until you press some key, preferably the SPACE bar, and then it clears and READY appears in the upper left corner. This is reversion to normal graphics operation in response to lines 9970–9990. If you type RUN again, the process repeats from the start.

Now let's vary this slighly by drawing a horizontal line that covers only part of the screen. Line 110 remains unchanged when you enter the following replacement:

```
120  FOR X = 100 TO 220: GOSUB 60: NEXT X
RUN
```

After a delay for initializing the screen, the background again turns brown and a line appears at the vertical center (Y = 100) that stretches partway across the screen, also centered horizontally.

Now let's draw a vertical line instead. In the other two experiments, we held Y steady at 100 and varied X to make a horizontal line. This time we'll hold X steady at 160 (the center X) and vary the Y coordinate with

```
110  X = 160
120  FOR Y = 0 to 199: GOSUB 60: NEXT Y
RUN
```

and, behold! a vertical line drops down the center of the screen.

To make a centered vertical line segment 120 dots high, we need to proceed from the midpoint less 60 to the midpoint plus 60, with the replacement line

```
120  FOR Y = 40 TO 160: GOSUB 60: NEXT Y
RUN
```

Sure enough, we get a centered line segment.

Now let's combine these two centered segments, one horizontal, the other vertical, to make a big plus sign in the middle of the display. For this we need more instructions, since the program has to do twice as

much work. Type

```
110    Y = 100
120    FOR X = 100 TO 220: GOSUB 60: NEXT X
130    X = 160
140    FOR Y = 40 to 160: GOSUB 60: NEXT Y
RUN
```

After the initialization delay the horizontal appears, and then it's crossed by the vertical. Your first HRG multipart display is complete, and it really wasn't very difficult to use this highly complex Commodore 64 capability.

We interrupt this program to bring you two practical messages.

Repairing errors in HRG programs

Chances are excellent that sooner or later you're going to hit the wrong key or otherwise make a program incomprehensible to BASIC. Maybe you already have, in fact. If you haven't, let's force an error on purpose. Type the line

```
110 Y/100
```

This line cannot be executed by BASIC because it gives no name to the result of the division, so when BASIC finds it, it issues the message SYNTAX ERROR. Type RUN and watch what happens.

After the screen turns brown, you suddenly get a bunch of colored blobs that look like the medals on a soldier's chest. The Commodore 64 is trying to tell you there's a bug in your program, but because it's in HRG mode instead of character mode, it no longer speaks the language.

You can restore the screen by holding down RUN/STOP and hitting the RESTORE key (above RETURN). This places the machine back into its condition at power up, but it doesn't erase your program. Thus, you can type LIST to see the program. If you're sure the error is in the problem-solving part of the program (not in the standard part), you can list only that part with the command LIST 100–1000 and check it.

Now let's say you didn't notice the typo in line 110. You can get the actual error message in legible form by typing the command RUN 100, which starts the program at line 100 instead of at the start. This bypasses the steps that change the computer's mode of operation, so you'll get no fancy display, but you *will* get the error message SYNTAX ERROR IN 110. You can then find and fix the error and try again. (If the error is, in fact, in the fixed portion, the error message will tell you that, too, by proceeding in the same fashion.)

Now let's suppose that you have a display that isn't at all what you

expected, maybe just a bunch of random dots, but the computer hasn't tried to issue an error message. This kind of bug is called a logic error: nothing syntactically wrong with your program, but it returns incorrect results. The first thing to suspect is that your calculations are producing bad XY coordinates. You have to have some notion of what they ought to be, of course, to work on the problem. Assuming that you do, insert the extraneous statement

```
81 PRINT X,Y,BA,BP: RETURN
```

and then execute the program with RUN 100. The computer will print a tabulated list of each X, Y, byte address, and bit position. You can stop the listing by hitting the RUN/STOP key, and use this information to figure out what's wrong with the program. When you have made the corrections, RUN 100 again and verify the list manually. To give it the acid test, remove the extra line by typing simply 81 and hitting RETURN, then type RUN and see if your plot looks right.

Speeding up screen initialization

Having run several HRG programs and seen how long it takes to clear the screen of garbage, you're probably wondering how arcade-type games do it so fast. The answer is that they use machine language, and we're using BASIC. The fault lies not with the computer, but with the horrendous inefficiency of BASIC.

This is a book about using the special features of the Commodore 64 and not about machine language, so we're not going to cover 6510 assembly-language programming. That's a whole big subject all by itself. Our programs here all use BASIC because it's easier to understand than assembly language. Once you've mastered the principles discussed here, if you want to speed up your Commodore 64 and do slick games and graphics, buy a book on 6502 assembly language, which is virtually the same as for the 6510 CPU used in your computer. Assembly language translates directly into machine language, and runs a hundred times faster than BASIC in many applications.

In the interest of time, however, we'll employ a little trickery to make BASIC write its own machine language subroutine for clearing the HRG screen memory at addresses 8192–16191. The machine language of the computer consists of numeric codes placed in memory, which we can do by reading and POKEing numbers in DATA statements. To call this subroutine from BASIC, we use the SYS instruction, which differs from GOSUB only in that it goes to a machine-language subroutine at a named address rather than to BASIC instructions at a named line.

For those who are interested, Appendix C lists in assembly-language-

source form the machine-language subroutines used in this book. Figure 5.4 gives the BASIC statements that effect lightning initialization of the screen and abolish forever the interminable delay at the start of the HRG programs.

```
5 FOR A = 49152 TO 49170
6 READ B: POKE A,B: NEXT A
7 DATA 169, 0, 160, 0, 162, 32, 153, 0, 32
8 DATA 200, 208, 250, 238, 8, 192
9 DATA 202, 208, 244, 96
30 SYS 49152
```

Figure 5.4 Machine language to clear the HRG screen

After you've keyed this in, run the program. The effect, you'll see, is astounding. Whereas before there was a pause long enough to permit a trip to the store while the screen cleared, now the screen clears before your finger leaves the RETURN key. This should give you some appreciation of how cumbersome BASIC is, and how essential assembly language is for fast graphics.

Interrupt the display and delete the program lines between 110 and 9960, then save the skeleton on tape or disk as the foundation for all your own HRG programs.

Useful geometric figures

Now that we know a little about making lines in HRG, let's develop a library of useful routines for various common shapes. Along the way we'll create some interesting displays, but our purpose is to demonstrate principles of graphics programming; the creativity of harnessing these principles for your own ends is up to you, and the possibilities are limitless.

A rectangle is the logical outgrowth of what we already know. It combines straight vertical and horizontal lines. As in the case of some of our character graphics figures, we need to define the boundaries of the rectangle. This is best done by assigning them to symbolic names rather than hard-coding them into the program, so that a change in the dimensions of the rectangle can be made by altering only one line rather than having to hunt through gobbledygook looking for them. Let's try it with Fig. 5.5.

```
110 L=120: R=200: T=60: B=140
120 Y=T: FOR X = L TO R: GOSUB 60: NEXT X
130 X=R: FOR Y = T TO B: GOSUB 60: NEXT Y
```

```
140 Y=B: FOR X = R TO L STEP-1: GOSUB 60: NEXT X
150 X=L: FOR Y = B TO T STEP-1: GOSUB 60: NEXT Y
RUN
```

Figure 5.5 Drawing a rectangle with the HRG skeleton

Line 110 declares the left, right, top, and bottom coordinates, using names that suggest what they mean. Lines 120–150 then draw the figure as you might by hand, in a continuous motion. Note that lines 140 and 150 go backwards (from a high coordinate to a low one), which requires the STEP-1 clause of the FOR instruction. You have to be careful to keep in mind which direction the plot is moving in an instruction, so that you can employ the STEP clause to go backwards if necessary.

A diagonal line (45° or 135°) is almost the same as a vertical or horizontal line, except that as you change one coordinate, you must also change the other by the same amount. The increments of the changes are +1 or −1, depending on the direction of the line. Figure 5.6 is an example.

```
120 Y = T: FOR X = L TO R: GOSUB 60
130 Y = Y + 1: NEXT X
140 Y = B: FOR X = L TO R: GOSUB 60
150 Y = Y - 1: NEXT X
RUN
```

Figure 5.6 Making diagonal lines

Line 110 is still in the program to establish the boundaries L, R, T, and B, which we also use in this experiment. As you see, a large X appears at the center of the screen. Lines 120 and 130 form a loop that draws a line sloping downward because each time X increases, Y does too. Lines 140 and 150 work just the same, except that Y decreases for each increase in X, so the line slants upward from left to right.

Now let's combine the last two examples by adding the box figure back in after the X is drawn. Do this with Fig. 5.7, in which the X appears and the box is drawn around its extremes.

```
160 Y = T: FOR X = L TO R: GOSUB 60: NEXT X
170 X = R: FOR Y = T TO B: GOSUB 60: NEXT Y
180 Y = B: FOR X = R TO L STEP-1: GOSUB 60: NEXT X
190 X = L: FOR Y = B TO T STEP-1: GOSUB 60: NEXT Y
RUN
```

Figure 5.7 Making a rectangle around an X

A sloping line is one that goes between any two points on the screen regardless of their locations. (That is, not in the same row or same column, and not at a 45° angle.) Unlike vertical, horizontal, and diagonal lines, a sloping line can't be produced by character graphics unless the two points happen to be in line with each other. It is, therefore, our first real application of high-resolution graphics.

As his legacy, some long-dead mathematician left a method for finding the distance between two points on an XY grid. It goes like this:

$$D = \sqrt{(X2 - X1)^2 + (Y2 - Y1)^2}$$

where $X1$ and $Y1$ are the coordinates of Point 1 and $X2$ and $Y2$ are the coordinates of Point 2. As an example, suppose Point 1 is at 83, 19 and Point 2 is at 217, 59. To figure how far apart they are:

$$D = \sqrt{(217 - 83)^2 + (59 - 19)^2}$$
$$D = \sqrt{17{,}956 + 1600}$$
$$D = \sqrt{19{,}556}$$
$$D = 138.843$$

In other words, Point 1 and Point 2 are about 139 units apart. In BASIC, the distance equation is written as

```
D = SQR((X2 - X1)^2 + (Y2 - Y1)^2)
```

However, it takes BASIC about twenty times longer to raise a number to its square than it does to multiply the number by itself and achieve the same result. Consequently, for reasons of speed we'll rework the instruction to read

```
DX = X2 - X1: DY = Y2 - Y1
D = SQR((DX * DX) + (DY * DY))
```

The value D tells how many steps it takes to draw the line, and the coordinates of Points 1 and 2 tell where its end points are.

The term *vector* describes the direction an object moves as a result of forces acting on it. In this case, the "object" is the plot itself and the "forces" are the rates of change in the X and Y directions required to get from Point 1 to Point 2. The X vector is the difference between X1 and X2 divided by the distance D, or, in BASIC,

```
XV = DX / D
```

The Y vector is computed the same way:

```
YV = DY / D
```

We then move the plot through "D" points starting at X1, Y1, each time adding XV to the most recent X and YV to the most recent Y. At the

end of the line, we'll be at X2, Y2, which is Point 2. To prove these ideas, delete lines 180 and 190 from the current program in memory, then type and run Fig. 5.8. This program works with the HRG skeleton to produce a line in the upper center of the screen that gently slopes to the right.

```
110 X1 = 83: Y1 = 19: X2 = 217: Y2 = 59
120 DX = X2 - X1: DY = Y2 - Y1
130 D = SQR((DX * DX) + (DY * DY))
140 XV = DX / D: YV = DY / D
150 X = X1: Y = Y1
160 FOR P = 1 TO D: GOSUB 60
170 X = X + XV: Y = Y + YV: NEXT P
```

Figure 5.8 Sloping line between two points

This method of plotting slopes works for any direction, since the vectors depend on the numeric differences between the coordinates of the end points. Negative differences thus produce vectors that move the plot in the correct direction to link the starting and ending points, regardless of where they are in relation to each other.

From a slope to a circle is a surprisingly short step in concepts. To draw a circle anywhere on the screen, we need to know only the coordinates of its center and one point anywhere around its circumference. The distance formula finds the radius, and then we draw the circle by plotting the end points of the radius through 360° around the center. That sounds fairly simple, and, in fact, it is. The BASIC statements themselves explain how to calculate a circle, as shown in Fig. 5.9.

```
110 CX = 160: CY = 100: PX = 205: PY = 122
120 DX = PX - CX: DY = PY - CY
130 RC = SQR((DX * DX) + (DY * DY))
140 FOR  A = 1 TO 360: R = A * π / 180
150 X = RC * COS(R) + CX
160 Y = RC * SIN(R) + CY
170 GOSUB 60
180 NEXT A
RUN
```

Figure 5.9 Plotting a circle

Lines 110–130 use the distance method to find the radius between the center (CX, CY) and the point on the outside (PX, PY). The radius,

which swings around the center of the circle, is called RC. Line 140 calls out the 360 angles and converts degrees to radians (which BASIC uses instead of degrees). Lines 150 and 160 compute the X and Y coordinates of the swinging end of the radius at the current angle and offset them for the center point. You can play with this routine, changing the values in line 110, which reposition the circle and make it different sizes.

These basic shapes should give you a good idea of how to utilize the plotting capabilities of HRG. Presently, we'll put the principles to use in a few practical or at least intriguing applications.

Adjusting reference points

X coordinates are numbered left to right from 0 through 319, and Y coordinates top to bottom from 0 through 199. That works fine for dot positioning, but it's not always in concert with our purposes, especially when plotting mathematical functions. The Y axis in particular presents a problem, since in mathematics the Y increases upward, not downward. Also, math graphs shift the axes around to make space for plotting negative X's and Y's.

It's not difficult to make adjustments for a different coordinate system from that of the screen. Before writing to the screen with GOSUB 60, all you have to do is apply factors that convert from your set of reference points to the machine's.

The easy one is the X position of the Y axis; that is, where the Y axis is in reference to the left edge of the screen. In the screen's coordinate system, the Y axis *is* the left edge of the screen, because that's the 0 point for X coordinates. Let's say that instead you want the Y axis to run down the middle of the screen (column 160). Any points to the left of column 160 will have negative X coordinates, and to the right positive. In this scheme, then, if X = −160, it will fall on the left edge of the screen, when X = 0, it will be in the center, and when X = +159, it will be on the right edge. To convert from this X system to the screen's, you simply add the position of the Y axis on the screen to your X coordinates. We can call this the X adjustment factor (XF), and use the formula

SX = XF + PX

to change the program X (PX) into the screen X (SX). In this example, XF = 160.

Adjusting the Y coordinates is a little more complicated, since the coordinates progress in opposite directions, increasing top to bottom in reality, but bottom to top in mathematical applications. The Y position of the X axis (where it is with respect to the top of the screen) we can call the Y adjustment factor (YF). The Y axis is at YF, and all Y points

below it are negative, while all above it are positive. I'll call the program's Y coordinates PY. To convert from PY to the screen's Y (SY), subtract both YF and PY from 199, the highest real Y coordinate, with

SY = 199 − YF − PY

This "flips" the program Y coordinate to the screen's and adjusts for the relative position of the X axis on the display. Thus, if the X axis is at row 100 (vertical center) and we have a program Y coordinate of 20 (i.e., 20 rows above the X axis), this formula adjusts it as

SY = 199 − 100 − 20
SY = 79

Similarly, if we have a PY of −20,

SY = 199 − 100 − (−20)
SY = 119

These results are intuitively correct, and we can use the formulas with minor modifications in all mathematical plotting programs. As a simple example, insert Fig. 5.10 into the HRG skeleton.

```
110 YF = 100: XF = 160
120 FOR H= -80 TO 80: V = 0
130 X = XF + H: Y = 199 - YF - V: GOSUB 60
140 NEXT H
150 FOR V= -80 TO 80: H = 0
160 X = XF + H: Y = 199 - YF - V: GOSUB 60
170 NEXT V
180
```

Figure 5.10 Positioning relative axes

The "180" clears away the leftover line from Fig. 5.9. As this program runs, you'll see it draw two intersecting lines in the center of the screen. Notice that the Y axis is drawn going upward because it steps from a negative Y to a positive value. In this program, we've used the variables H to represent the horizontal position (the "PX" or program X) and V for the "PY" or program Y. Lines 130 and 160 convert to the screen's coordinate system. As far as the program knows, these lines are at X = 0 and Y = 0.

Scaling the display

The sales chart program (Fig. 3.8) demonstrated, among other things, the scaling of the display for the data. Similarly, we need to consider

matters of scale in HRG, since we have a screen of a fixed, finite number of points upon which to represent a universe of infinitely varying dimensions.

To scale the display horizontally, we have to know the maximum range of X's that will be displayed. We can then divide the range by the 320 available dots. For instance, if X varies from −2 to +5, then the range of X is 7. Dividing 7 by 320, we find that each horizontal dot represents 0.021875. If X varies between −1256 and +319, the value of each horizontal dot is 1575 / 320 = 4.921875. Similarly, we can scale the vertical dimension by dividing the range of Y by 200. The scales need *not* be the same in both directions, but of course the resulting display will be distorted.

Because this is a tedious task, we can let the computer do it for us through instructions in the program. As a case study, let's plot the equation

$$y = x^3 - 5x^2 - x + 7$$

Substituting H for x and V for y as we translate it into BASIC, we get the statement

```
V = H^3 - (5 * H^2) - H + 7
```

Now let's say we want to plot this through the range of H between −2 and +5. We aren't sure what the range of V will be, but calculating the equation at its extreme H's, we find that V = −19 when X = −2 and V = 2 when H = +5. For good measure, if we calculate V = 0, we get H = 7. For a little breathing space, then, we can set the range of V as −20 to +10.

This gives us enough to begin the program by scaling the horizontal dimension as follows:

```
110  LH = -2: HH = 5
120  HD = ABS(HH - LH) / 320
```

We can read line 110 as "the low H is −2 and the high H is 5" and line 120 as "a horizontal dot represents the maximum range of H divided by 320 dots." The ABS function forces the result HD to be a positive number when LH and HH are both negative. The scale of the V dimension is computed with

```
130  LV = -20: HV = 10
140  VD = (ABS(LV) + HV) / 200
```

We're using a different coordinate system here than the screen does, so it will be necessary to convert systems as we discussed in the last section. Because we'll be plotting a lot of points from different parts of the program, a subroutine will make the programming easier. Figure 5.11 puts it at line 1000.

```
999 GOTO 9960
1000 REM  **  GRID CONVERSION
1010 X = XF + (H/HD)
1020 Y = 199 - YF - (V/VD)
1030 GOSUB 60
1040 RETURN
```

Figure 5.11 Subroutine to adjust axes

Any time you want to plot a point on the screen at the intersection of V and H, you can simply GOSUB 1000 and it will take care of the grid conversion for you.

There's one problem with this subroutine, however: we don't know (nor does the program) what the adjustment factors XF and YF are. In which display column does the V axis appear, and at which row is the H axis?

Lines 110 and 120 set up the horizontal dimension, and they contain what we need to position the V axis. In this case, LH = −2, and line 120 has found that HD—the value of a horizontal dot—is 0.021875. One full unit along the H axis is 1/HD, or about 45.71 dots. The low H is two units to the left of the V axis (LH = −2), so the V axis has to be at the X coordinate given by 45.71 * 2, or at 91.42. Generalizing this, we get

```
150  XF = (1/HD) * ABS(LH)
```

which you should add to the program accumulating in your computer. The process is similar for the placement of the H axis, giving us

```
160  YF = (1/VD) * ABS(LV)
```

Working this by hand, we get VD = 0.15 and YF = 66.67, which is one-third of the way down the 200-point screen. Zero is one-third of the way between 10 and −20, so the result is right.

This exercise has revealed that we're going to have fractional values for V and H. There is no such thing as a fractional dot on the screen, however. Consequently, to increase the accuracy of the display, we'll need to round fractions to the nearest whole number. The best place to do this is in the grid-conversion subroutine, which calls for minor revisions in lines 1010 and 1020. Change them to read:

```
1010  X = INT(XF + (H/HD) + .5)
1020  Y = INT(199 - YF - (Y/YD) + .5)
```

This more correctly positions the dot without affecting the accuracy of the numbers.

Any self-respecting graph gives reference points, typically at least the X and Y axes. We can draw them on the screen (and verify visually our calculations) with Fig. 5.12.

```
170 FOR H = LH TO HH STEP HD
180 V = 0: GOSUB 1000
190 NEXT H
200 FOR V = LV TO HV STEP VD
210 H = 0: GOSUB 1000
220 NEXT V
RUN
```

Figure 5.12 Drawing X and Y axes

Now all that remains is to plot the equation itself. Everything is in place to do so, including traces of the zero axes. Type the instructions in Fig. 5.13, which display the X and Y axes, and then plot the curve of an equation that rises, then falls, then rises again.

```
230 FOR H = LH TO HH STEP HD
240 V = H^3 - (5 * H^2) - H + 7
250 GOSUB 1000
260 NEXT H
RUN
```

Figure 5.13 Plotting the curve of a complex equation

We now have a complete equation-plotting program, and we can modify it to plot any XY function through any range of values by changing only three lines (110 for the X range, 130 for the Y range, and 240 for the equation itself). This pro forma program requires only that your equation use H as the input variable (or *argument* in mathematical terms) and V as the result.

Just for the fun of it, let's change the program for a different equation. This time we'll plot

$$y = x^4 - 7x^2 + 2x - 10$$

through the x range of -4 to $+4$. Arbitrarily, we'll set the y range as -30 to $+30$. Change the program by typing Fig. 5.14.

```
110 LH = -4: HH = 4
130 LV = -30: HV = 30
240 V = H^4 - (7 * H^2) + (2 * H) - 10
RUN
```

Figure 5.14 Another complex equation plot

See how easy it is for Johnny to get the computer to help with his algebra homework? This program graphs a curve that looks like a lopsided W. It's a useful program for anyone who needs to plot curves mathematically, but we're not quite finished with it yet.

Speeding things up

When you program in BASIC, you make a deal with the gods of computing: you get an easy language, but you lose machine efficiency. No other programming language gobbles up so much of the machine doing simple tasks. Consequently, there isn't anything we can do about the length of time it takes to plot equations and geometric figures and the like. If this bothers you, switch to a more efficient compiled language such as Pascal or FORTRAN—which should be available for the Commodore 64 by the time you read this book—or to compact, fast, excruciatingly tedious assembly language.

On the other hand, it does seem a waste of time to go dot by dot in plotting (plodding?) straight lines from border to border for background visual references such as the X and Y axes. "Gut feel" says there has to be a better way, and "gut feel" is right.

You'll recall that the subroutine at lines 60–90 translates an XY coordinate into a screen-memory address represented by the variable BA and a bit setting represented by BP. These values are available to the main program after the subroutine has been called, and we can use them to accelerate straight lines.

To see how, first consider a horizontal line. Usually we start such a line at the left border and draw it across to the right. Our speed package starts the same way, by presenting a Y coordinate for the vertical position of the line and an X coordinate between 0 and 7, to make sure we start in the leftmost character cell. The subroutine does its thing by turning on the dot to start the line, but more importantly it returns the byte address in variable BA.

In Fig. 5.2, we laid out a map of the byte arrangement in a typical row of character cells, where bytes are stacked eight deep across 40 cells. The bytes in a given row of dots, then, are eight addresses apart; the upper right corner is byte 0, the next byte to the right is 8, the next is 16, etc. Thus, if we know the address of the byte at the start of the row, we can step across by eights to find all the other bytes in that row. That's what BA tells us after we've plotted the first dot in the row.

One byte can represent any value from 0 through 255. The reason we can't go any higher than 255 is because all the bits are on. When a byte in screen memory has a value of 255, it makes a solid line one cell in width. By stepping across 40 cells by eights and POKEing the value 255

into each, we'll draw a straight line from one side to the other. You'll see after you type the modification in Fig. 5.15.

```
170 V=0: H=LH: GOSUB 1000
180 FOR H = BA TO (BA+319) STEP 8
190 POKE H, 255: NEXT H
RUN
```

Figure 5.15 Speeding up horizontal line drawing

Wow! The horizontal line streaks across the screen.

We start a vertical line the same way, but proceed differently from there. In this case, we have to plot one point at the *top* of the screen (not at the bottom, as this program currently does). That will give us the starting address BA and the bit setting BP. We're going to step down through the character cells, which means going through eight bytes in a row, then skipping to the next cell below. It takes 320 bytes to make a complete row of character cells, and eight bytes to make one cell, so the top byte of the next cell down is 320 − 8 = 312 from the bottom of the current cell. Thus, we step through eight, jump 312, step through another eight, jump 312, and so on through the last row on the screen. There are 25 rows to process.

In this case, we can't use the value 255 unless we want to draw a line an entire character cell in width. For one dot only, we need to raise 2 to the BP power and OR it with the contents of each byte in the column. Why OR? Because that will leave intact any bits already set in the byte, keeping our vertical line from wiping out part of the display, such as a horizontal line it crosses. Let's try it with Fig. 5.16.

```
200 H = 0: V = HV
201 GOSUB 1000
202 IF Y < 0 THEN V=V-1: GOTO 201
203 J = 2^BP: BA = BA - 2
204 FOR H = 1 TO 25: FOR V = 0 TO 7
205 POKE BA+V, (PEEK(BA+V) OR J)
206 NEXT V
207 BA = BA + 320: NEXT H
210
220
RUN
```

Figure 5.16 Speeding up vertical line drawing

Line 202 deserves an explanation. Sometimes, due to a round-off error in the coordinate conversion, the Y coordinate is translated to −1. The subroutine disregards negative numbers and thus returns a garbage value for BA. Line 202 compensates by retrying until a valid result is returned. The performance improvement here isn't quite as dramatic as it was for the horizontal line. That's because it's passing through 200 bytes instead of only 40. Nevertheless, it's much faster than it used to be.

Save this program under a name such as "MATH GRAPH." It's a useful addition to your program library.

Paint by numbers

In the last section we came upon an idea, the fast line in high-resolution graphics (HRG), that cries out to be developed further. From tape or disk, reload a fresh copy of the HRG skeleton and we'll dip our brush into a computerized paintpot.

The term "painting" as used in computer graphics means to fill in an area on the screen with a foreground color. Dot-by-dot plotting is so slow that we didn't dare try painting before we discovered the fast line; it would have taken as long to paint the screen as to paint the living room. But now we have a powerful new tool for computer graphics. As a simple example, use Fig. 5.17 to make a solid rectangle.

```
110 FOR Y = 60 TO 139
120 X = 120: GOSUB 60
130 FOR X = 0 TO 80 STEP 8
140 POKE BA + X, 255
150 NEXT X: NEXT Y
RUN
```

Figure 5.17 Painting a rectangle

It takes only a few seconds for a white square to appear. This little routine uses the same technique as the fast horizontal line in the last section, plotting one point and then using the value BA from the subroutine as its starting point. The difference is that it doesn't go all the way across the screen. Line 130 limits it to the 11 character cells in the middle of the screen. The Y loop repeats the process 80 times, each time drawing the line directly below the previous one. The dots run together vertically as well as horizontally, producing a solid figure.

All the dots in this pattern fit neatly into the vertical boundaries of the character cells, so all the program did was to fill the appropriate byte

within each cell with the value 255. One of the benefits of HRG, however, is that it overcomes the cellular limitations of character graphics. What can we do when the shape boundaries occur somewhere within cells, say, between the third bit of the left one and the fifth bit of the right one?

For those situations, we have to make the program smart enough to know how to deal with the odd bits that don't fill a complete byte. As a means of illustrating a solution, we'll take the case of a solid line between X coordinates 67 and 92. Figure 5.18 illustrates the byte configurations to draw this line.

64	65	66	67	68	69	70	71		88	89	90	91	92	93	94	95
0	0	0	1	1	1	1	1	(All 1s col 72–87)	1	1	1	1	1	0	0	0

Figure 5.18 Bits for a solid line between X coordinates 67 and 92

To create this line, the program needs to set the bits from column 67 through column 71, filling in the partial byte starting at the bit position corresponding to column 67. It can then use the fast-line method to set eight bits at a time from column 72 through 87, which are the 16 bits filling two complete bytes. At column 88 it fills in the five high-order bits that finish the line out through column 92.

Painting, an operation performed many times at various places in a program, is the kind of thing for which subroutines were invented, so we'll develop one here for that purpose. The subroutine needs to know three things: the Y coordinate of the line, the X of the starting point (left end), and the X where the line stops. The calling program must therefore set up the variable Y, the starting X (SX), and the ending X (EX) before issuing the instruction GOSUB 9000. The most important variable within the subroutine is CX, which maintains the current X coordinate. Thus the subroutine starts with the instructions

```
8999  GOTO 9960
9000  REM ** PAINT ONE LINE
9010  CX = SX
```

Line 8999 protects the subroutine from accidental entry, and 9010 sets the current X at the starting point.

The subroutine should verify that it's not being asked to paint from right to left, which would make a mess of things. It determines this by comparing the starting and ending X's; SX should always be less than or equal to EX; if it's not, the subroutine flips them with the instructions

```
9020  IF SX <= EX THEN 9040
9030  SX = EX: EX = CX: CX = SX
```

This exchanges the two values and ensures that the line will proceed from left to right as it should. It also enables the subroutine to plot a single point (when SX = EX).

The skeleton's plotting subroutine at lines 60–100 returns the value BP, indicating the bit position of the X coordinate. The paint subroutine plots the first dot in the line via a GOSUB 60 in order to obtain the BP and BA (byte address) for that dot. The BP tells the subroutine how to build the first byte of the line. It also deals with the situation when there aren't enough dots in the line even to fill the remaining space in the first byte, a problem I'll discuss later. For now, let's consider the construction of the first byte of the line.

The first bit can occur any place from the extreme left (BP = 7) to the extreme right (BP = 0). Wherever it is, the remainder of the byte is filled with 1's. Figure 5.19 lists all the BP's, the bytes they need to generate, and their values.

BP	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*Value*
0	0	0	0	0	0	0	0	1	1
1	0	0	0	0	0	0	1	1	3
2	0	0	0	0	0	1	1	1	7
3	0	0	0	0	1	1	1	1	15
4	0	0	0	1	1	1	1	1	31
5	0	0	1	1	1	1	1	1	63
6	0	1	1	1	1	1	1	1	127
7	1	1	1	1	1	1	1	1	255

Figure 5.19 First bytes as related to starting bit positions

The value of each bit position is 2 raised to the power of that position number. For example, for bit 4, 2^4 is 16, which gives a byte of 00010000. From Fig. 5.19, you see that the byte to begin a line at bit position 3 has the value 15. To construct that byte, raise 2 to the power of one greater than BP and subtract 1. In BASIC this is written as

```
PV = 2^(BP + 1) - 1
```

By substitution,

```
PV = 2^(3 + 1) - 1
PV = 2^4 - 1
PV = 16 - 1 = 15
```

That being the case, it would seem that you could construct a byte for any value of BP by the same means. And, if you try it for other values from 0 through 7, you'll find that you can. This is a quick way to make the proper first byte for a line no matter where the line begins.

Having built the byte, you can put it into the screen memory at address BA and the corresponding display dots will turn on. It's necessary to OR it with the current contents of that location, rather than simply POKEing it, in order not to disturb bits that have already been set. Next advance the current X by the BP plus one to keep track of the current position, and finally bump the screen-memory address BA ahead by eight to point it at the same Y in the next cell to the right. Figure 5.20 gives this sequence in BASIC.

```
9040 REM  **  START OF LINE
9050 X = CX: GOSUB 60
9060 IF BP > (EX - SX + 1) THEN 9180
9070 PV = 2^(BP + 1) - 1: POKE BA, PEEK(BA) OR PV
9080 CX = CX + BP + 1: BA = BA + 8
```

Figure 5.20 Making the start-of-line byte

Now let's talk about line 9060, which deals with a line that fills less than one byte. Say the program has reached line 9060 and BP = 5 for the start of this line. However, the length of the line (EX − SX + 1) is only 3 dots. If we simply let the program act on BP, it will generate a byte 00111111 when, in fact, with a line length of 3 it should generate 00111000. The bytemaker in line 9070 has no provision for such a configuration, and the CX counter will also get messed up with a false value if we fail to deal properly with this situation. Consequently, line 9060 checks BP against the length of the line, and if the check discloses that the line won't fill the remainder of the byte, it jumps to a short loop that plots the dots one at a time. The routine is tacked onto the end of the subroutine, but since BASIC rearranges lines into proper order no matter in what sequence we enter them, we can type Fig. 5.21 now.

```
9180 REM  **  BIT PLOT
9190 FOR X = SX TO EX: GOSUB 60
9200 NEXT X: RETURN
```

Figure 5.21 Painting a line shorter than one byte

After the first byte is built and displayed, the program can zip along filling whole bytes with 1's in a manner familiar from the fast-line technique. We do have to add some controls, however, so that it doesn't overrun the last byte in the line. Before filling a byte with 1's, we can have the program calculate the bits remaining in the line by subtracting the current X from the end X and adding 1. If the result is 8 or greater,

it's safe to fill the byte and display it, then update the byte address and the current X and repeat. When the number of bits remaining is less than 8, the program has to move on to a special end-of-line byte builder This portion of the subroutine is in Fig 5.22.

```
9090 REM  **  WHOLE BYTE
9100 BR = EX - CX + 1
9110 IF BR < 8 THEN 9140
9120 POKE BA, 255
9130 BA = BA + 8: CX = CX + 8: GOTO 9090
```

Figure 5.22 Painting whole bytes within the subroutine

At the end of the line we have the situation opposite that of the start of the line. Here the program has to fill in the bits at the *left* end of the byte far enough to reach the end X. The line end-bytes, their bit configurations, and their decimal values are shown in Fig. 5.23.

BR	*7*	*6*	*5*	*4*	*3*	*2*	*1*	*0*	*Value*
1	1	0	0	0	0	0	0	0	128
2	1	1	0	0	0	0	0	0	192
3	1	1	1	0	0	0	0	0	224
4	1	1	1	1	0	0	0	0	240
5	1	1	1	1	1	0	0	0	248
6	1	1	1	1	1	1	0	0	252
7	1	1	1	1	1	1	1	0	254
8	1	1	1	1	1	1	1	1	255

Figure 5.23 End-byte configurations related to bytes remaining

Calculating these bytes is more complicated, but not difficult. If you subtract the bits remaining from 8, you get the bits used. You can then raise 2 to this power and subtract from 256. In BASIC this procedure is written

```
PV = 256 - 2^(8 - BR)
```

Let's try it with BR = 5 (i.e., there are 5 bits remaining in the line). The computation goes

```
PV = 256 - 2^(8 - 5)
PV = 256 - 2^(3)
PV = 256 - 8 = 248
```

If you check the table in Fig. 5.23, you'll see that 248 is indeed the value of the byte needed to fill the remaining five dots in the line. Thus, all we

need to do to complete the line is to construct the byte based on the bits remaining, OR it into screen memory, and return to the calling program. Figure 5.24 does this.

```
9140 REM  **  END OF LINE
9145 IF BR < 0 THEN BR = 0
9150 PV = 256 - 2^(8 - BR)
9160 POKE BA, PEEK(BA) OR PV
9170 RETURN
```

Figure 5.24 Building the end-of-line byte

Now let's test the subroutine and see if we're as smart as we think. One way to be sure we have every possible starting and ending byte is to paint a pyramid, which Fig. 5.25 does.

```
110 TX = 160: SX = TX: EX = 160: T = 60: B = 120
120 FOR Y = T TO B: GOSUB 9000
130 SX = SX - 1: EX = EX + 1
140 NEXT Y
RUN
```

Figure 5.25 Painting a pyramid

Embellishments

Let's face it, a stark white pyramid on the screen is nice, but hardly in the tradition of ancient Egypt. Wouldn't it be more interesting to look down on Cheops' white pyramid against the brown backdrop of the sands from somewhere in the sky? Let's extend his pyramid into a projection drawing and discover how to unpaint selected bits to add detail. Start with Fig. 5.26.

```
150 REM  **  PROJECTION
160 SX = SX + 6: EX = EX - 1
170 IF SX >= EX THEN 190
180 GOSUB 9000: Y = Y + 1: GOTO 160
190 REM
RUN
```

Figure 5.26 Projection drawing of a pyramid

The basic pyramid now extends downward, slanting sharply in from the left and gradually from the right. This gives a projection effect as though we're looking down on it from an angle. The only trouble is, there's nothing marking the pyramid's solid white form to show the meeting of the two sides we can see. For that we need to unpaint some bits.

When a bit is on, it shows the foreground color; when it's off, the background. In a painted area, all the bits are on. The idea of unpainting, then, is nothing more than going through a painted area and turning off selected bits, causing the background color to reappear in those places.

After the projection is finished, we have all the leftover values we need. We know the coordinates of the pyramid's top (X = TX, Y = T) and, from the routine that drew the projection, X and Y now contain the location of the lower tip.

Remember the method for drawing a sloping line? We can modify it slightly to unpaint the dots that define the meeting of the pryamid's walls. It's simply a matter of drawing a line from the tip to the lower corner, but instead of setting bits for that line, we'll reset them.

The way we'll do that is mostly familiar; we'll call the subroutine at line 60 to tell us the bit position and byte address of the desired dot. To reset a bit, raise 2 to the power of the bit position and subtract it from the byte currently at the address. For example, if BP = 5 the mask byte MB has the value 2^5, which is 32. The painted byte has all bits on (11111111), for a value of 255. Subtracting 32, we get 223, which has the bit configuration 11011111. As you see, bit 5 is turned off. A POKE puts it back into screen memory. Figure 5.27 translates this into action.

```
190 REM  ** EDGE OF WALLS
200 X2 = X: Y2 = Y: X1 = TX: Y1 = T
210 DX = X2 - X1: DY = Y2 - Y1: X = X1: Y = Y1
220 V = SQR(DX * DX + DY * DY)
230 XV = DX / L: YV = DY / L
240 FOR Y = Y1 TO Y2 STEP YV: GOSUB 60
250 MB = 2^BP
260 POKE BA, PEEK(BA) - MB
270 X = X + XV
280 NEXT Y
RUN
```

Figure 5.27 Unpainting selected bits to add features

There it sits on the desert sands, a white pyramid with two sides clearly in view and the sharp edge standing out. We've just done a projection drawing with features.

Or how about an endless tunnel? Stare into the distance for a while and you'll become hypnotized. You have to wait a while for that to happen, though, because it takes about eight minutes for this picture to complete. That's because it paints a circle going half a degree at a time and then draws concentric circles, each half the diameter of the previous one, until the radius is only two dots. There's nothing new in the program in Fig. 5.28, and by now you should be familiar enough with graphics programming to understand from the statements what's happening.

```
110 RC = 60: CH = 160: CV=100
120 FOR A = 90 TO 270 STEP .5
130 RD = A * π/180
140 SX = INT(RC * COS(RD) + CH + .5)
150 EX = CH - SX + CH
160 Y =  INT(RC * SIN(RD) + CV + .5)
170 IF EX > (CH + 3) THEN GOSUB 9000
180 NEXT A
190 REM  **  CONCENTRIC CIRCLES
200 RC = INT(RC / 2 + .5)
210 IF RC <= 2 THEN 8999
220 FOR A = 1 TO 360
230 RD = A * π/180
240 X = INT(RC * COS(RD) + CH + .5)
250 Y = INT(RC * SIN(RD) + CV + .5)
260 GOSUB 60: MB = 2^BP
270 POKE BA, PEEK(BA) - MB
280 NEXT A: GOTO 190
RUN
```

Figure 5.28 Staring down an endless tunnel

As a final exercise in unpainting (and creating mesmerizing screens), we'll make the same painted circle and then draw a spiral inside it. This time the circle will be black against a white background, with a white spiral. Lines 40 and 41 specify the colors, and the other instructions in Fig. 5.29 add the spiral. The spiral is built on the principle that for each degree, the radius of the "circle" increases by 0.05 units. Note that it does not plot through 360°, but rather until the radius reaches the full diameter of the black circle. Because it works in such tiny increments, the spiral grows very slowly, making four visible rotations over about five minutes. Even though it is slow, however, the design is compelling.

```
40 FOR X=1024 TO 2023: POKE X,1: NEXT X
41 POKE 53280,1
190 REM  **  SPIRAL
200 R = 1: A = 0
210 R = R + .05: A = A + 1
220 RD = A * π/180
230 X = R * COS(RD) + CH
240 Y = R * SIN(RD) + CV
250 GOSUB 60: MB = 2^BP
260 POKE BA, PEEK(BA) - MB
270 IF R < RC THEN 210
280
RUN
```

Figure 5.29 Staring down a rifle bore

In this chapter we have barely scratched the surface of the potential that high-resolution graphics offers. It is a marvelous plaything that can occupy you for many enjoyable hours. Granted, it is a slow—agonizing might be a better word—process in BASIC, and the improvements we experienced in using a machine language routine to clear the screen memory ought to spur you on to learning the Commodore's assembly language. Still, all the principles are here, and some of the tools and approaches, for creating stunning graphics on the Commodore 64, no matter which programming language you use.

But before you thunder off to learn a better programming language, let us delve further into the capabilities that still lie hidden within this incomplete-looking tan box at your fingertips. We have only begun.

To, Meager

CHAPTER 6

BOLD STROKES AND DASHES OF COLOR: COMPUTERIZING THE ARTIST'S PALETTE

Up to this point, the only truly colorful display we've built was the house at the end of Chapter 3. The high-resolution graphics (HRG) plots in Chapter 5 were mostly white figures against a brown background, which provides nice contrast and resolution but has a certain monotony about it. The Commodore 64 has marvelous color capabilities, and in this chapter we'll computerize its palette.

Colors in standard HRG mode

Reload the HRG skeleton from tape or disk because we're going to add a few enhancements to it. One of its setup steps loads color memory (1024 – 2023) with a color byte by means of the loop in line 40. The program POKEs number 25 into color memory for a brown background with a white foreground. Since the color-memory byte is the basis for all that follows in this chapter, and of color graphics in general, it's important to have a firm understanding of its anatomy and its relationship to screen memory. Consequently, I'll review it briefly before proceeding.

The HRG color-memory byte specifies two colors. The upper four bits hold the number of the foreground color, the lower four the number of the background color. These color numbers are the POKE codes for colors shown in Fig. 3.9. To set the foreground color most easily, multiply the color number by 16, which shifts the color-number bit pattern to the high-order bits. Then set the background color by adding that color number

to the result of the foreground shift and POKE the byte into color memory. As an example, the color number for white is 1, which has a bit pattern of 0000 0001. To shift it, multiply by 16: that is, 1 × 16 = 16 with the bit pattern 0001 0000, so the foreground color is now set. Brown is color number 9, or bit pattern 0000 1001. Adding 9 to 16 produces 25, bit pattern 0001 1001. We think of the result as a single number, but the Commodore 64 regards it as two independent quantities (1 and 9) found at the same address.

That address is one of the 1000 locations in color memory. Each color-memory byte services an 8 × 8-dot area on the screen, which corresponds exactly to a character cell in normal graphics. The 8 × 8 dots are represented in screen memory by eight sequential bytes of 8 bits each. Any bit that is a 0 in screen memory shows up in the background color specified in that cell's color-memory byte; any 1 bit takes the foreground color. (The last part of this chapter will deal with a major exception to this rule, but it does hold true in standard HRG mode.)

It follows, then, that we can control the color combination of every 8 × 8-dot cell on the screen independently of the others by altering that cell's color-memory byte. When the HRG screen is empty (all bits set to 0 in screen memory), we should see different background colors by changing some of the color-memory locations. To try it, type Fig. 6.1 into the HRG skeleton.

```
110 CM = 1024: BC = 0
120 FOR P = 0 TO 479
130 POKE CM + P, BC + 16
140 BC = BC + 1
150 IF BC > 15 THEN BC = 0
160 NEXT P
RUN
```

Figure 6.1 Varying background colors

The upper half of the screen fills with various colors. Remember, we haven't written any bits into *screen* memory; these are variations in the background for each cell, and all screen-memory bits are still 0s. That being the case, any figure drawn on the screen should be in white, since we specified white (the "+ 16" in line 130) as the foreground. The figure will be, in effect, superimposed in white on whatever background is present in the cell. To prove this, let's paint a square in the center of the screen. Tap the SPACE bar to restore normal operation, then type Fig. 6.2.

```
170 SX = 110: EX = 210
180 FOR Y = 50 TO 150: GOSUB 9000
190 NEXT Y
RUN
```

Figure 6.2 Paint a square in the center of the screen

The white box that appears on the screen looks like a piece of paper lying on top of the multicolored checkerboard in the upper half of the display; it overlaps onto the plain brown field in the lower half. That's because the foreground color is the same throughout color memory, regardless of the background.

Keep this pattern in mind, so that you can contrast it with the effect of changing the *foreground* color in each cell while leaving the background the same for the entire screen.

Hit SPACE and type LIST 100–200 to see the program. We need to change only one line to reverse the order of things and set up different foreground colors for each cell, and that's the value POKEd in line 130. Currently, this line reads

```
130  POKE CM + P, BC + 16
```

The 16 is foreground color white, BC the variable color of the cells' backgrounds. To retain a brown background and vary the foreground, change the line to read

```
130  POKE CM + P, (BC * 16) + 9
```

and type RUN. There is a delay while the color memory is altered, and then a multicolored checkerboard emerges. Halfway down the checkerboard, the background changes to white. That's because we altered the foreground colors only in the upper half of the screen. The 1 bits in the upper-half cells are producing whatever foreground color prevails for that cell.

Thinking ahead to the things we might do with this new technique, it becomes clear that we may want to alter either the foreground or the background color of a particular cell without changing the other, and without even knowing or caring (programmatically, at least) what the other color is.

Say, for example, that we want to change the foreground color in the upper nibble without affecting the background color held in the lower four bits. First we have to find out what the color-memory byte holds. We get that with a PEEK. Next we mask out the high-order nibble (change it to 0s) while retaining the lower bits unchanged. That's done by ANDing with 15. AND is similar to a filter; it only lets 1 bits from the operand pass through 1 bits in the mask. The value 15 has all the

low-order bits set to 1, so when we AND it with the color byte we PEEKed, it turns off the upper-nibble bits and leaves the lower nibble alone. We can then multiply the new foreground color by 16 to shift it into the high nibble and add that to the result of the AND. Finally, we POKE it back where it came from. BASIC is more concise about it:

```
POKE CB, (PEEK(CB) AND 15) + (CN * 16)
```

where CB is the address of the color byte and CN is the number of the color we want to set up for the foreground.

Changing the background without affecting the foreground is similar, except that we keep the high nibble with an AND and add the number of the new background color. When all the bits of the high-order nibble are on, the value is 240. Thus,

```
POKE CB, (PEEK(CB) AND 240) + CN
```

These are pretty complicated instructions, and since we'll probably execute them many times in any respectable graphics program, they belong in subroutines where we only have to type them once. Also, we can include instructions to find the color-byte address based on the XY coordinates. These XY's will correspond to character cells (40 × 25) and not to individual HRG dots, and they won't conflict because we'll keep the color-alteration and bit-mapping portions of the program separate. Delete lines 110–190 and add Fig. 6.3, then save the program as the new HRG skeleton.

```
8500 REM  **  CHANGE FOREGROUND
8510 CB = 1024 + (Y * 40) + X
8520 POKE CB, (PEEK(CB) AND 15) + (CN * 16)
8530 RETURN
8540 REM ----------------------
8600 REM  **  CHANGE BACKGROUND
8610 CB = 1024 + (Y * 40) + X
8620 POKE CB, (PEEK(CB) AND 240) + CN
8630 RETURN
```

Figure 6.3 Subroutines to alter one color in the color byte

These subroutines require the color number CN and the X and Y coordinates of the affected cell. To change the foreground to color CN, GOSUB 8500; for a new background color, GOSUB 8600.

Now let's be creative in testing what we've done. We'll set up vertical bars of background colors ranging from white to black through the grays, and horizontal stripes of bright foreground colors, and then we'll paint a circle and see what happens. Figure 6.4 shows the program to do this.

```
110 POKE 53280,0: N = 1
120 FOR X = 0 TO 39
130 FOR Y = 0 TO 24
140 ON N GOTO 150, 160, 170, 180, 190
150 CN = 1: GOTO 200
160 CN = 15: GOTO 200
170 CN = 12: GOTO 200
180 CN = 11: GOTO 200
190 CN = 0
200 GOSUB 8600
210 NEXT Y
220 N = N + 1: IF N = 6 THEN N = 1
230 NEXT X
240 REM  ** SET UP F/G COLOR STRIPES
250 N = 1
260 FOR Y = 0 TO 24
270 FOR X = 0 TO 39
280 ON N GOTO 290, 300, 310, 320, 330
290 CN = 10: GOTO 340
300 CN = 14: GOTO 340
310 CN = 8: GOTO 340
320 CN = 5: GOTO 340
330 CN = 7
340 GOSUB 8500
350 NEXT X
360 N = N + 1: IF N = 6 THEN N = 1
370 NEXT Y
380 REM  ** PAINT A CIRCLE
390 RC = 60: CH = 160: CV = 100
400 FOR A = 90 TO 270 STEP .5
410 RD = A * π/180
420 SX = INT(RC * COS(RD) + CH + .5)
430 EX = CH - SX + CH
440 Y = INT(RC * SIN(RD) + CV + .5)
450 IF EX > (CH + 3) THEN GOSUB 9000
460 NEXT A
RUN
```

Figure 6.4 Multicolor demonstration program

This is a long program because of the ON . . . GOTO statements that select the color numbers. It's also long-running, so there's a lengthy period of apparent inactivity while the foreground memory is loaded. But then a stunning display emerges, a brilliant multihued Easter egg against the subdued striped background of grays. It seems almost three-dimensional, as though floating between us and the background.

The background and foreground stripes run perpendicular to each other because the background was laid out vertically, while the foreground was superimposed horizontally on those background colors, with neither one disturbing the other.

Souping up color memory

A program seldom reworks color memory as extensively as the last one did. Usually we set up color memory for a basic combination of foreground and background colors and then alter individual cells as necessary. Still, I expect your fingertips are getting sore from drumming the table as BASIC grinds through color memory. In Chapter 5 we speeded up the clearing of screen memory with a machine-language subroutine, and now that color memory is becoming the "Great Procrastinator," it's high time we did the same favor for it.

Line 40 in the HRG skeleton sets the screen to the combination of foreground and background colors specified by the value it POKEs into color memory. The machine-language subroutine will do the same thing, but so much faster that it implements the combination of the entire screen instantly. To install it, line 5 has to READ and POKE an additional 34 bytes, and of course those machine-language instructions have to be added to the DATA statements. The present line 10 becomes line 19 to avoid being overlaid by new DATA. Line 40, which is no longer needed, will be replaced altogether. Figure 6.5 shows the instructions to complete the installation.

```
5 FOR A = 49152 TO 49204
19 POKE 53265, PEEK(53265) OR 32
10 DATA 169, 16, 160, 232, 162, 4, 141, 0, 4
11 DATA 238, 26, 192, 208, 3, 238, 27, 192
12 DATA 136, 208, 242, 202, 208, 239, 169, 0
13 DATA 141, 26, 192, 169, 4, 141, 27, 192, 96
35 POKE 49172, 26
40 SYS 49171
RUN
```

Figure 6.5 Installation of color-memory machine subroutine

Line 35 prepares the subroutine by POKEing the value for the desired color combination into address 49172, and line 40 calls the subroutine with SYS 49171. After you type RUN, watch the screen and try to remember how long it used to take for brown to cover the entire display. Now it happens in an instant. We have eliminated one of the annoyances of programming graphics in BASIC. Enter all the line numbers from 110 through 460 to clear out the Easter egg program, then save this as a new version of the HRG skeleton.

Now let's exploit this new tool. Did you ever play a video game and wonder how they make the flash of the explosion when the Starship loses to the Adenoids? Well, let's blow it up and find out.

We'll materialize a white rectangular ship in deep space. You'll have to pretend that the ship is under attack by invisible rays in the gamma spectrum. It will just float there in blackest space for a while, then suddenly blow up with a lot of flashing. Figure 6.6 shows how.

```
35 POKE 49172, 16: POKE 53280, 0
110 SX = 60: EX = 150
120 FOR Y = 70 TO 120: GOSUB 9000
130 NEXT Y
140 FOR T = 1 TO 800: NEXT T
150 REM  **  EXPLOSION
160 FOR X = 1 TO 20
170 POKE 49172, 2 * 16: SYS 49171
180 FOR T = 1 TO 10: NEXT T
190 POKE 49172, 7 * 16: SYS 49172
200 FOR T = 1 TO 10: NEXT T
210 NEXT X
220 POKE 49172, 0: SYS 49171
RUN
```

Figure 6.6 Blowing up the Starship

Line 35 sets up a white foreground and a black background and border. Lines 110–130 build the Starship, and 140 lets it float serene and unsuspecting for a while. Then along come lines 160–210 and, BOOM! the Starship flashes in alternating red and yellow. It would flicker even faster if we eliminated lines 180 and 200, which are delay loops. It would take longer blowing up, too, if we raised the loop count in line 160, but the gamma spectrum is potent. The *coup de grace* comes in line 220 where the Starship vanishes from the universe without a trace.

This particular approach has a drawback (for us as well as for the occupants of the Starship). The machine-language subroutine changes *all*

of color memory at once. It does this exactly the same way the old loop at line 40 did it, by going through 1000 bytes in sequence and POKEing a value into each, but it seems instantaneous because machine language is so much faster than BASIC. The drawback is that all objects on the screen immediately change from their present color(s) to the same new color. No exemptions are granted. Real game programs have subroutines such as this, but with the ability to alter selected areas of color memory and not the whole thing.

All is not lost in this regard, however, because later in this chapter we'll discover that there are ways, even easier than this, to change all instances of a particular color on the screen with one instruction. Meanwhile, let's come back down to earth and talk a little common sense about graphics programming.

HRG display planning

HRG displays are essentially immobile, since to relocate an object entails removing it from its present screen addresses and rebuilding it elsewhere, and that takes twice as long as making it from scratch. Chapter 8 talks about *sprites*, action figures that exist independently of the screen memory and move around easily. Sprites are for animation; HRG and to a great extent character graphics are for the stationary backdrop against which action occurs, or for static representation of data and other fixed displays.

Most of the demonstrations up to this point have been pretty random in nature; it really didn't matter where we put a figure on the screen, because the outcome would be the same. The introduction of colors opens doors for all sorts of new possibilities, but to use them to their fullest you have to plan where figures will appear and what colors they will need.

The planning process differs little from the ideas presented in Chapter 2. Use graph paper laid out to show the row and column numbers. In this case, the coordinates refer to color memory, but you can also translate them into the X and Y coordinates of bits. Sketch the screen layout on the graph paper, bearing in mind that there are 64 points within each cell to take care of the fine-shaping. If you're no successor to Picasso, trace a photo or drawing instead. Remember that each cell holds two colors. It might take some fudging and repositioning when you have a lot of colors close together. If this becomes too much of a problem, your display is a good candidate for the multicolor graphics later in this chapter. The present result should look something like a needlepoint pattern, and in fact you can program graphics directly from charts sold in needlework shops, which we will do presently.

To illustrate some of the subtleties of color planning, we'll take what appears to be the rather trivial example of a flower pattern that abounds in Pennsylvania Dutch art: a red tulip with a yellow heart inside it, against a white background. It's shown in Fig. 6.7.

Figure 6.7 Pennsylvania Dutch tulip design

Some have tried to divorce computer graphics from mathematics, but none have succeeded. The connections are simply too numerous and intimate. Much of art, in fact, is highly mathematical; listen carefully to Bach sometime. At any rate, you cannot do computer art without graphics and you cannot do computer graphics without mathematics. The trick is to describe a shape or a feature mathematically so the computer can act on it.

That sounds intimidating. Few of us are Einstein clones, and you'll notice I said few of *us*. But computer graphics is really a matter of common sense, breaking a problem down into small solutions applied on a step-by-step basis. That, by the way, is a definition of programming in general, and it applies equally to the writing of the federal government's financial accounting system and to the drawing of a Pennsylvania Dutch flower.

This design seems simple enough at first glance, but let's look more

closely. It's a red area superimposed on a white background, and if that's all it amounted to we could make it with character graphics. The yellow heart is what places it into the category of HRG. Why? Because in character graphics a cell can display either its foreground color or the screen color. There is no way for character graphics to accommodate the diagonal parts of the yellow heart without introducing the background color inside the red tulip.

The upper rectangular part of the tulip is fairly straightforward; Fig. 6.8 changes color memory for that area so that it has a red background for the flower and a yellow foreground for the heart. A rectangle of red appears on a white screen with a green border. This is red background (line 130) with a yet-to-be-seen yellow foreground. The notch appears in the center top because of line 160, which resets that cell's background to white.

```
35 POKE 49172,1: POKE 53280,5
110 FOR Y = 7 TO 11
120 FOR X = 15 TO 23
130 CN = 2: GOSUB 8600
140 CN = 7: GOSUB 8500
150 NEXT X: NEXT Y
160 X = 19: Y = 7: CN = 1: GOSUB 8600
RUN
```

Figure 6.8 Top of tulip

Now we get to the lower part that vees down. It's similar to an upside-down pyramid, so we could set those cells' foreground to red and paint the pyramid. However, the yellow heart spoils that easy solution. It needs to be painted into the same lines, and if we do that when the foreground is red, the lower part of the heart won't show up as yellow.

One solution that comes to mind is this: Paint the lower part of the flower with a red foreground, then reset the background of the cells to yellow where the heart occurs and unpaint the dots inside the heart. The problem with this is that it lacks consistency. We're using two methods to make the flower (upper part in background red, lower part in foreground red painting) and two methods to make the heart (upper part painted in foreground yellow, lower part unpainted in background yellow). It is said that a successful program is one that works, and this solution will work. But just because a program works, that doesn't mean it's a good program. Good programming utilizes consistent methods. Consequently, we'll discard this idea and pursue one closer to the approach used earlier. We'll still eventually have to compromise our principles, but

a minor inconsistency undertaken deliberately is better than a gross one stumbled into. In this method (Fig. 6.9), we'll place background red and foreground yellow into all the cells that fall entirely within the flower, just as we did in the upper part.

```
170 SX = 16: EX = 22
180 FOR Y = 12 TO 15
190 FOR X = SX TO EX
200 CN = 2: GOSUB 8600
210 CN = 7: GOSUB 8500
220 NEXT X
230 SX = SX + 1: EX = EX - 1
240 NEXT Y
RUN
```

Figure 6.9 Lower part of tulip

This makes the rough shape of the flower with a pair of upside-down staircases converging at the bottom. We'll fill in the steps in a moment, but first let's paint the heart. If you and I were sitting together at the Commodore 64, you might—with some justification—object on the grounds that this is inconsistent. In fact, however, it's not, because everything we've done so far has been in preparation for painting the heart. I also have another reason for deferring the stairstep-filling, and that is that it's the most difficult part of this exercise.

```
250 REM  **  PAINT THE HEART
260 FOR TX = 140 TO 164 STEP 24
270 SX = TX: EX = TX + 7
280 FOR Y = 72 TO 79: GOSUB 9000
290 SX = SX - 1: EX = EX + 1: NEXT Y
300 NEXT TX
```

Figure 6.10 Top of heart

The manufacture of a heart is a three-step process. First we paint in the two trapezoidal areas at the top with Fig. 6.10. Then Fig. 6.11 makes the rectangular component in the center. Finally, it tapers down to the bottom in Fig. 6.12, thus completing a vividly colored heart.

```
310 SX = 133
320 FOR Y = 80 TO 87: GOSUB 9000: NEXT Y
```

Figure 6.11 Center of heart

```
330 GOSUB 9000
340 Y = Y + 1: SX = SX + 1: EX = EX -1
350 IF EX >= SX THEN GOTO 330
RUN
```

Figure 6.12 Bottom of heart

Speaking of consistency, perhaps I should explain why we don't have a FOR/NEXT loop structure in the last part of the subprogram as we did in all the others. The reason is that we are converging two lines. It's possible, of course, to calculate the point where they meet and tell BASIC how many Y loops to execute to reach that point. But the computer can figure that more easily than we can, by comparing the SX and EX values. When EX—normally to the right of SX—is instead to the *left* of (less than) SX, the plot points have converged and it's time to call it quits.

Okay, we've procrastinated long enough. Let's figure out how to fill in the stairsteps. The object of the game is to join the outside angles and fill the triangular area with red. The problem is that, at the moment, the cells nestled within the steps have a white background color and an unknown foreground color. We can approach the problem from either of two directions:

1. Make the inside-angle cells into a white background with red foreground and fill in each angle with red.
2. Make the inside-angle cells into a red background with white foreground and trim off the edges with white.

Either way, we're dealing with the meeting of two background colors, so there's no way to maintain consistency. On the other hand, there is something to be said for the fact that the white background is essentially passive; we didn't go to any trouble to make it white, whereas we did some work to make the flower's background red. Therefore, it seems more in concert with the overall approach to select the second alternative.

```
360 REM  **  COMPLETE THE FLOWER
370 SX = 15: EX = 23
380 FOR Y = 12 TO 15
390 FOR X = SX TO EX STEP (EX - SX)
400 CN = 2: GOSUB 8600
410 CN = 1: GOSUB 8500
420 NEXT X
430 SX = SX + 1: EX = EX - 1
440 NEXT Y
RUN
```

Figure 6.13 Modify cells

The first step, then, is to have Fig. 6.13 reset the eight involved cells for red background and white foreground. This places red squares into the stairsteps, extending them outward and resolving, so far, nothing. That comes next.

Here we have two distinct problems. First, to trim the red corners off on the left side, and then to trim them off on the right side. The angles oppose, and therefore so do the solutions, although they share common elements.

First the right side. We need to go down cell by cell, painting in white so that the corners are cut off on a precise diagonal. The diagonal crosses four cells, or 32 vertical dot coordinates. We can't set a consistent SX for all 32 because at the second cell down we'll begin painting black triangles outside the cells filling the angles, since they don't have a white foreground. Consequently, Fig. 6.14 steps cell by cell, moving the SX to the right eight bits each time.

```
450 SX = 120: EX = SX
460 FOR Y1 = 12 TO 15: V1 = Y1 * 8
470 FOR V2 = 0 TO 7: Y = V1 + V2
480 GOSUB 9000: EX = EX + 1
490 NEXT V2
500 SX = SX + 8
510 NEXT Y1
RUN
```

Figure 6.14 Left diagonal

On the left side the continually varying point was EX (the right end of the paint line) and SX changed every eighth vertical step. That reverses on the right side so that SX moves steadily to the left and EX follows the stairsteps. There's one other difference, too: Fig. 6.15 forces EX to follow the *last* bit of each cell, unlike on the other side where SX was in the first bit. That's what the first calculation in line 520 does.

```
520 EX = 24 * 8 - 1: SX = EX
530 FOR Y1 = 12 TO 15: V1 = Y1 * 8
540 FOR V2 = 0 TO 7: Y = V1 + V2
550 GOSUB 9000: SX = SX - 1
560 NEXT V2
570 EX = EX - 8
580 NEXT Y1
RUN
```

Figure 6.15 Right diagonal

The tulip now lacks only its stem, a minor but completing detail taken care of by Fig. 6.16.

```
590 REM  **  STEM
600 FOR Y = 16 TO 17: X = 19
610 CN = 13: GOSUB 8600
620 NEXT Y
RUN
```

Figure 6.16 Stem of tulip

Certainly there are more inspired subjects than a Pennsylvania Dutch flower, and just as certainly there are more elegant solutions to graphics than this one. The object has been not to create an enduring masterpiece of computer art, but to give you a close and detailed look at the problems and mode of thinking that surround the making of high-resolution multicolored graphics. The very length of the program—52 lines, supported by perhaps a hundred lines of setup and subroutines—to draw something as simple as this native-art flower should give you some insight into the subtleties and complexities of making fancy graphics.

Of course, we are dealing with the machine at a fairly primitive level in these exercises, as well as furnishing you with some software tools in the form of subroutines and the HRG skeleton that you can use and expand upon as you become more adventuresome. Other computers—Apple, as an example—come furnished with prewritten graphics support software so that in BASIC you simply specify CIRCLE along with the coordinates of the center and one point on the circumference, and the machine automatically and rapidly plots it for you. Those computers cost many times as much as the Commodore 64; in terms of capabilities they deliver about the same or less; in terms of ease of use they are "friendlier." The Commodore 64 gives you a lot, but it is not an easy machine to program. Commodore has promised to deliver graphics enhancements that get away from POKEs and PEEKs and ANDs and ORs, which still boggle this writer after a decade and a half of programming.

Yet even if Commodore has kept its promise, this material and these experiments are worthwhile for understanding what makes things happen. Also, graphics support software gives you a convenient means for accomplishing the ordinary, but nobody has yet devised a software product that does everything for everybody. Hopefully these pages will give you—if not the exact solution for a vexing problem—at least the foundation for exploiting the full and tremendous capabilities of this machine.

In that spirit, then, let us continue our discussion of static graphics.

Multicolor high-resolution graphics

Sometimes you need more than two colors in a cell, or sometimes the effort of planning colors to keep three or more from trying to occupy the same cell at the same time becomes overwhelming. When that happens, it's time for multicolor high-resolution graphics, which we will call MHRG.

There are more comparisons than contrasts between the two types of high-resolution graphics. Like standard HRG, MHRG uses an 8000-byte screen memory in which bits indicate dots on the display. Both use the 1000 bytes at 1024–2023 for color information, and nearly all the methods covered so far under normal HRG apply to MHRG. In fact, the HRG skeleton becomes the new MHRG skeleton by adding only two lines.

But of course, MHRG is not the same as standard HRG. If it were, we wouldn't be discussing it separately. In MHRG, you can have up to four colors per 8 × 8 dot cell, and only one of those colors is of necessity consistent throughout all cells in the display. Thus it's possible to have some 13,500 different color combinations, so such painstaking planning as that for the Pennsylvania Dutch tulip is largely eliminated in MHRG.

So why not use it all the time? you might ask. The answer is that it costs something. MHRG has only half the horizontal resolution: it has 160 dots instead of 320. MHRG is more difficult to program, since not only do you have to plot the *pixel* (picture element, analogous to a dot in standard HRG), but you have to indicate within that pixel what color you want it to have. Also, display planning is more difficult when the X and Y axes differ by a factor of 2, since a numerically square figure (say 15 × 15 units) on the display is twice as wide as its height.

The MHRG pixel consists of two bits in memory, mapped to a two-dot-wide area on the screen. This accounts for the halving of horizontal resolution. The color of the pixel is indicated by the setting of the bits. Two bits can contain any of the four numeric values 0–3, represented in binary as 00, 01, 10, and 11. These values are not color numbers *per se*, but are pointers that tell the VIC-II chip where to find the desired color number. Figure 6.17 dissects a byte in MHRG screen memory.

Each two-bit element represents a two-dot area on the display and indicates its color. The screen memory byte shown here has the pattern 01111000.

Bit settings	⟶	01	11	10	00
Colors indicated	⟶	1	3	2	0

Figure 6.17 Indicating colors by pixel in multicolor HRG

In standard HRG the byte in Fig. 6.17 produces eight dots, of which the first in reading order is background, followed by five foreground dots, and then three more of the background color. This same byte in MHRG

produces four pixels (the same physical width as eight dots), with the first pointing to the location of pixel color 1, the second to 3, the third to 2, and the fourth to 0.

Since the screen memories look the same in both standard and multicolor HRG, we have to tell the Commodore 64 how to interpret the bits. This is done with two instructions that flip bits in the VIC-II control registers at addresses 53265 and 53270. The first we already know about because it initiates HRG mode in the skeleton with

```
19  POKE 53265, PEEK(53265) OR 32
```

The second establishes multicolor mode—the interpretation of bytes in screen memory—with

```
18  POKE 53270, PEEK(53270) OR 16
```

The corresponding reversal of these instructions to place the computer back into standard character mode are

```
9970  POKE 53265, PEEK(53265) AND 223
9975  POKE 53270, PEEK(53270) AND 239
```

the first of which already appears in the HRG skeleton at 9970. Type the new instructions at lines 18 and 9975 to modify the HRG skeleton for MHRG.

VIC has to look several different places to find the colors pointed to by these numbers 0–3. The sources of color information are shown in Fig. 6.18.

Bits	Color Number	Source
00	0	Screen color (address 53281)
01	1	Upper four bits of cell (1024–2023)
10	2	Lower four bits of cell (1024–2023)
11	3	Color memory cell (55296–56295)

Figure 6.18 Sources of color information in MHRG

Note that color memory (55296–56295) has reappeared on the scene in order to hold color 3. Color 0 comes from the screen background register at 53281 and thus prevails within any pixel—no matter where on the screen—where bits 00 appear. The other colors are set on a cell-by-cell basis. Thus, you can change all instances of color 0 with one POKE, and individually control the other colors at specific places on the screen. In a moment I'll illustrate color controls with some machine exercises.

First, though, let's talk about a technique used in the next several

experiments for storing and displaying "canned" cell images, that is, cells that are already set up and don't have to be plotted. This technique works for both standard and multicolor HRG cells, though here I'll focus on MHRG mode.

As you know, a cell image consists of eight consecutive bytes that can be viewed as an 8 × 8 matrix of bits. Therefore, if you want to construct the cell image, you can take graph paper, make a box of 8 × 8 squares, and fill the boxes with 0s and 1s. In standard HRG, these 0s and 1s represent background and foreground colors, respectively. In MHRG, we can group them by pairs to represent the colors of pixels. Let's say that we want to contruct a cell image in a sort of barber's pole fashion, with the four colors slanting across from northeast to southwest. Figure 6.19 shows the cell matrix to do this.

Pixel color patterns				Bit patterns								Decimal
0	1	2	3	0	0	0	1	1	0	1	1	27
0	1	2	3	0	0	0	1	1	0	1	1	27
1	2	3	0	0	1	1	0	1	1	0	0	108
1	2	3	0	0	1	1	0	1	1	0	0	108
2	3	0	1	1	0	1	1	0	0	0	1	177
2	3	0	1	1	0	1	1	0	0	0	1	177
3	0	1	2	1	1	0	0	0	1	1	0	198
3	0	1	2	1	1	0	0	0	1	1	0	198

Figure 6.19 Cell-image matrix for a barber's pole

Notice that in every second byte the pattern "wraps around" so that the leftmost pair of bits goes to the right end and all the rest shift left. Remember, in MHRG one horizontal unit is twice the distance of one vertical unit, so if the barber's pole stripes are to slant at 45°, we need to have two vertical pixels for each horizontal pixel; hence, two bytes of one pattern, two of the next, etc. The decimal numbers that form these patterns are listed at the right in Fig. 6.19. You can calculate them by adding the place values of all the 1 bits in each byte, but that's tedious. Fortunately, there is an easier way with the color pointers. The "place values" of each position in the byte are (left to right) 64, 16, 4, and 1. Multiply each pixel color by its place value and add the results to find the decimal number for the byte. For example, in the last two bytes the colors are 3, 0, 1, and 2, so we can calculate the decimal value as follows:

$$\begin{aligned} 3 \times 64 &= 192 \\ 0 \times 16 &= 0 \\ 1 \times 4 &= 4 \\ 2 \times 1 &= 2 \\ &\overline{198} \end{aligned}$$

which is the same as adding position values for all the 1 bits.

If we know that certain cells in a display aren't going to change, it makes sense to store them in numeric form (the "mathematical description" discussed earlier) and then simply load them into their places on the screen, rather than plotting them with elaborate equations each time. That's an easier and much faster means of building displays.

The DATA statement is useful for building fixed information into programs. We've already used DATA statements to hold the two machine-language subroutines in the HRG skeleton (lines 7 through 13). We can also use them to contain images in the form of sequences of decimal numbers representing bit patterns.

One weakness of DATA statements is that their contents can be conveniently read only once during a program run. Each READ fetches the next data item, and the only way to back up and reread one or several items if you need the same information repetitively is with the RESTORE instructions. Since RESTORE places the data pointer at the very start of the first DATA statement in the program, if what you want is some distance down, you have to read through all the intervening items to get to it. Much bother. It's better to read images and store them in arrays. You can directly access an array as often as you want.

Because eight bytes make a cell, it takes eight values in a DATA statement to describe the cell, and the arrays holding the images will always have a DIMensioned size that is a multiple of eight, less one (the first element in an array is the Øth). If an image occupies three cells, we need 24 DATA items to describe it, and the array is declared with DIM I(23), where I is the name of the array. The first byte is thus called I(Ø), the second I(1), and the last I(23). To move the image from the array to the display, determine the address in screen memory of the Øth byte of the first cell where it goes, then execute a loop that moves the number of bytes in the image from sequential entries in the array to sequential screen-memory addresses. That way you'll be able to place this same image wherever you want on the screen, and you can also repeat it by copying it to several different locations.

Let's illustrate this with some experiments in MHRG. The initial exercises will use the barber's pole cell we developed in Fig. 6.19. To code the image, type

```
3ØØØ  DATA 27, 27, 1Ø8, 1Ø8, 177, 177, 198, 198
```

The image will appear with the following colors:

Ø	Black
1	Red
2	Blue
3	White

The color memory at 1024–2023 needs to be set up for the red-and-blue combination with the statement

```
35  POKE 49172, (2 * 16) + 6
```

which gives the machine-language subroutine a value to place throughout the two-color memory. Now we can write the actual program and place it below line 100.

Figure 6.20 loads the color designated by pixel color 3 into the main color memory at 55296, allocates the image array, and sets the screen (pixel value 00) to black. It also (line 130) moves the cell image from the DATA statement into the array.

```
110 FOR X = 0 TO 999: POKE 55296+X,1: NEXT X
120 DIM I(7): POKE 53281,0
130 FOR B = 0 TO 7: READ I(B): NEXT B
```

Figure 6.20 Set up colors and array for barber's pole

To test the setup, we'll place one cell in the center of the screen. Working in cell XY (rather than bit XY) coordinates, the center of the display is at row 12 (= Y) and column 20 (= X). With eight bytes per cell and 320 per row, the address of the center cell is ((Y * 320) + (X * 8)) + 8192, which works out to 12192. To thrust one lonely little cell into center stage, then, move the eight image bytes into the eight addresses beginning at 12192 with the statement

```
140  FOR B = 0 TO 7: POKE 12192 + B, I(B): NEXT B
RUN
```

Don't be concerned if nothing seems to be happening for a while. We're using BASIC in line 110 to clear color memory, and it's a reminder of how spoiled we've gotten by those machine-language subroutines. (Relief is moments away.) At last, looking tiny and frightened in the vast expanse of the black screen, our MHRG cell appears.

This is a vivid illustration of the synergistic effect: graphics of any sort is more than the sum of its parts. You can't even tell what it is right now, alone and isolated. Let's make something greater using this single multicolor image.

But first we'll do something about the irksome delay of clearing color memory. For this, we can fool the same machine-language subroutine that clears two-color memory at 1024. The subroutine is built to place the byte at 49172 into 1000 consecutive memory locations, so all we have to do is change its starting address from 1024 to 55296, give it a value

to work on, and call it. This we can do by typing

```
110  POKE 49179,216: POKE 49172,1: SYS 49171
```

The address, 216, gives the number of 256-byte blocks from the start of memory where the subroutine is to begin writing the value at 49172. 216 × 256 = 55296, which is the beginning of color memory, so that's where it goes. Type RUN and see how much faster the cell appears. Magic!

Now, to make a rectangle of these multicolor images, we can write them into the same range of columns in consecutive rows, providing a formula for calculating the address of each cell. Figure 6.21 makes that happen.

```
140 FOR R = 5 TO 20: Y = (R * 320) + 8192
150 FOR C = 5 TO 35: X = C * 8
160 FOR B = 0 TO 7
170 POKE X + Y + B, I(B)
180 NEXT B: NEXT C: NEXT R
RUN
```

Figure 6.21 Multicolor rectangle of barber's pole pattern

To see the effects of the MHRG color specifications, let's do a series of loops to alter them. Each loop will step through all 16 colors. The first experiment in Fig. 6.22 changes the basic screen color saved at 53281. Line 200 contains a delay loop to let you have time to see each color before going to the next. Notice that not only does the screen background change, but so does the color combination within the rectangle. This is because we're using pixel value 00, the background color, within the cells.

```
190 FOR X = 0 TO 15: POKE 53281, X
200 FOR T = 0 TO 200: NEXT T: NEXT X
210 POKE 53281,0
RUN
```

Figure 6.22 Color change experiment #1

Next we'll step through the spectrum for pixel color 1, located in the high-order nibble of each location in two-color memory. Figure 6.23 uses the machine-language subroutine at 49171 to rewrite the affected value. Line 190 repoints the subroutine to the color memory at 1024. The rest

should be familiar. Note that all instances of a given color change at the same time.

```
190 POKE 49179, 4
200 FOR X = 0 TO 15: POKE 49172,(16 * X) + 6
210 SYS 49171
220 FOR T=0 TO 200: NEXT T: NEXT X
RUN
```

Figure 6.23 Color change experiment #2

To step through the low nibbles (color 2), type the line

```
200  FOR X = 0 TO 15: POKE 49172, (2 * 16) + X
```

and rerun the program. Finally, to step through pixel color 4 in the main color memory, replace the following lines:

```
190  POKE 49179, 216
200  FOR X = 0 TO 15: POKE 49172, X
RUN
```

This should give you an appreciation for the color-control potential of MHRG. Now let's move on to more interesting images.

Persian carpet-weaving

Needlepoint patterns are an excellent source of material for computer graphics. An example that also brings out a few neat tricks is a small pattern adapted from *Persian Rug Motifs for Needlepoint*, by Lyatif Kerimov (1975: Dover Books, New York). This floret, found on page 9, is one of the smallest in the book, but large enough to demonstrate what can be done with needlepoint patterns and MHRG. The adaptation appears in Fig. 6.24.

The color specifications in Fig. 6.24 are mine and not from the needlepoint pattern. The motif occupies a space 23 stitches—pixels in computer graphics terms—square, which nearly fills a 3 × 3 cell area. This is nine cells, or 72 bytes. Because each horizontal pixel is two dots wide, the essential squareness of the design requires that every odd-numbered byte be a duplicate of the even-numbered byte above it. This means that we can describe the pattern in 36 bytes and let the program replicate it in an array of 72 elements with the following loading loop:

```
FOR B = 0 TO 71 STEP 2
READ F(B): F(B + 1) = F(B)
NEXT B
```

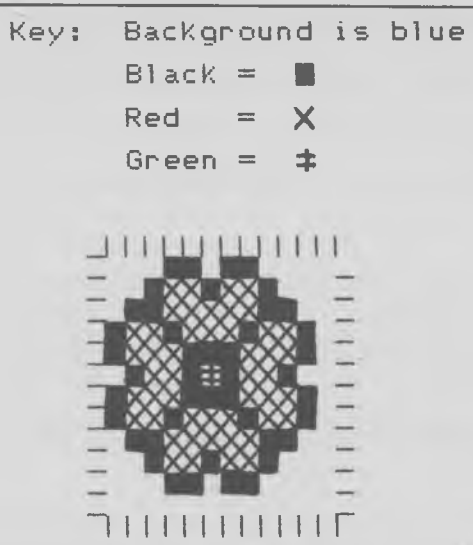


Figure 6.24 Persian floret needlepoint pattern

In other words, the array index B advances by twos, and each time a value is read from the DATA statement it is placed in the current element and in the current element plus one. We therefore don't need to key each byte twice. Figure 6.25 shows the chart for reducing this pattern to color numbers, and from these to numeric values using the multiplication method described earlier.

	Cell 0	Cell 1	Cell 2	Bytes		
	0 0 0 1	1 0 1 1	0 0 0 0	1	53	0
Row 0	0 0 1 2	2 1 2 2	1 0 0 0	6	154	64
	0 1 1 2	2 2 2 2	1 1 0 0	22	170	80
	1 2 2 1	2 2 2 1	2 2 1 0	105	169	164
	1 2 2 2	1 1 1 2	2 2 1 0	106	86	164
Row 1	0 1 2 2	1 3 1 2	2 1 0 0	26	118	144
	1 2 2 2	1 1 1 2	2 2 1 0	106	86	164
	1 2 2 1	2 2 2 1	2 2 1 0	105	169	164
	0 1 1 2	2 2 2 2	1 1 0 0	22	170	80
Row 2	0 0 1 2	2 1 2 2	1 0 0 0	6	154	64
	0 0 0 1	1 0 1 1	0 0 0 0	1	53	0
	0 0 0 0	0 0 0 0	0 0 0 0	0	0	0

Figure 6.25 Defining the motif as bit patterns

In constructing DATA statements from this diagram, we have to be sensitive to the natural progression of memory addresses as related to cells. Assuming duplication of the bit patterns, the order of bytes is the first four bytes of the first column, then the top four of the next column, then the top four of the last column, and from there to the fifth through eighth of the left column, etc. Figure 6.26 contains the resulting DATA statements.

```
3010 DATA 1,6,22,105,      53,154,170,160, 0,64,80,164
3020 DATA 106,26,106,105, 86,118,86,169,  164,144,164,164
3030 DATA 22,6,1,0,        170,154,53,0,   80,64,0,0
```

Figure 6.26 DATA statements for the motif in Fig. 6.25

The spaces are not significant, but serve to illustrate the order of the bytes taken from the pattern. Also, by starting at line 3010, we leave intact the barber's pole bytes already entered on line 3000. The reasons will become clear presently. Type Fig. 6.26 into the program.

Considered on a cell-by-cell basis (rather than byte-by-byte), we have a numeric description of the motif that is in "reading" order, as we might read three lines of three characters each. We might want to place this figure anywhere on the screen and we might want to repeat it, since such patterns often recur in an oriental carpet. Consequently, the best means for placing it on the display is a subroutine that gets its positioning information from a pair of cell coordinates indicating the upper left corner of the floret. We'll call them TX and TY. It will start at the upper left corner, display three cells, then advance to the next row and display three more, and then move to the bottom row. Type the subroutine in Fig. 6.27 to see how it's done.

```
2000 REM  ** DRAW FLORET AT TX, TY
2010 P = 0
2020 FOR R = TY TO (TY + 2): Y = (R * 320) + 8192
2030 X = TX * 8: L = 0
2040 FOR B = P TO (P + 23)
2050 POKE Y + X + L, F(B): L = L + 1
2060 NEXT B: P = P + 24
2070 NEXT R
2080 RETURN
```

Figure 6.27 Subroutine to place pattern anywhere on screen

The value P gives the starting point in array F for the current row of the floret, so that the loop counter B in lines 2040–2060 extracts images in the correct range of elements. The value L keeps track of the current screen-memory address during the loading of the three consecutive cells on a row.

We can't test this subroutine until we've prepared a program within the MHRG skeleton to use it, and set up the color combinations indicated in Fig. 6.24. Figure 6.28 does this.

```
35 POKE 49172, (16 * 0) + 2
110 POKE 49179, 216: POKE 49172, 5: SYS 49171
120 POKE 53281, 14: POKE 53280, 6
```

Figure 6.28 Setting up colors for the floret

The barber's pole pattern still remains in the DATA statement preceding the definition of the floret, so it needs an array and a loop to load it. In addition, of course, the floret also has to have an array. To create these arrays type Fig. 6.29.

```
130 DIM I(7), F(71)
140 FOR B = 0 TO 7: READ I(B): NEXT B
150 FOR B = 0 TO 71 STEP 2
160 READ F(B): F(B + 1) = F(B)
170 NEXT B
```

Figure 6.29 Placing the images in arrays

The patterns are now in arrays and ready for display, so let's have a look at the floret in computer needlepoint. To place it in the center of the screen, type

```
180  TY = 11: TX = 19: GOSUB 2000
```

Even though the initial test program is written and in the machine, we can't type RUN yet. Why not? Type LIST 100–300 and you'll see that there are four lines (190–220) left over from the last program we ran. Delete them by typing their line numbers, hitting RETURN after each. Now you can run the program and watch the floret appear in center screen.

We now have the basis for weaving a Persian carpet. Let's first make a border using the barber's pole figure. It will be located one cell away from the screen's edges (in rows 1 and 23 and columns 1 and 38), giving us Fig. 6.30.

```
180 REM  **  BORDER
190 FOR R = 1 TO 23 STEP 22
200 Y = (R * 320) + 8192
210 FOR C = 1 TO 38: X = C * 8
220 FOR B = 0 TO 7: POKE X + Y + B, I(B): NEXT B
230 NEXT C: NEXT R
240 FOR C = 1 TO 38 STEP 37: X = C * 8
250 FOR R = 2 TO 22: Y = (R * 320) + 8192
260 FOR B = 0 TO 7: POKE X + Y + B, I(B): NEXT B
270 NEXT R: NEXT C
RUN
```

Figure 6.30 Border of carpet

The carpet pattern will consist of three rows of five florets each. The subroutine already exists (Fig. 6.27) to produce a floret at any pair of coordinates TX and TY, so it's merely a matter of defining the coordinates and calling the subroutine with the loops in Fig. 6.31.

```
280 REM  **  DRAW FLORETS
290 FOR TY = 5 TO 17 STEP 6
300 FOR TX = 7 TO 31 STEP 6: GOSUB 2000
310 NEXT TX: NEXT TY
RUN
```

Figure 6.31 Weaving the carpet pattern

A pity you can't put it on the floor and walk on it, for we have woven a classic Persian carpet from a needlepoint pattern. They say it takes months to make a real one; we've done it in a few minutes, though, of course, theirs is more enduring than ours. This one will fly away if you hit the SPACE bar. But this exercise should give you a good feel for creating interesting and pleasing graphics displays without a great deal of difficulty.

Plotting Points in MHRG

A college professor I know uses the multicolor capability of his computer to display complex mathematical relationships, assigning a different color to each variable and then seeing how changes in any one affect the others. This is a powerful use of computer graphics that demonstrates the tech-

niques for coordinate plotting in MHRG, which is a little different than in standard HRG. The sample exercise here will show how the sines and cosines of angles relate to each other.

When plotting mathematical equations in Chapter 5, we converted the screen to an XY coordinate system that had the standard math layout and allowed us to place the X and Y axes wherever we wanted them. These positions, in the screen's "real" coordinate system, were given in the variables XF and YF. We also applied a scale factor in both the vertical (VD) and horizontal (HD) dimensions to convert the equation's coordinates—called H and V—into the screen's X and Y coordinates. This subroutine still lives, ignored recently but capable of operation, at line 1000 in the MHRG skeleton. It frees us from having to think about the constraints of the screen and lets us work in convenient dimensions.

There's a lot of leftover trash from the last exercise lying around the MHRG skeleton right now, so load a fresh copy from tape or disk.

The screen for the new program will be black with the X axis drawn in white. We'll plot the sine in red and the cosine in green. This gives us the following color schedule:

Color Number	*Color*
0	Black
1	Red
2	Green
3	White

which we can implement in the program with Fig. 6.32.

```
35 POKE 49172, (2 * 16) + 5
110 POKE 53280, 0: POKE 53281, 0
120 POKE 47179,216: POKE 49172,1: SYS 49171
```

Figure 6.32 Colors for sine/cosine plot

As an angle increases through its full cycle, its sine and cosine vary between −1 and +1. In MHRG, as in standard HRG, there are 200 vertical points on the screen. Thus, if we position the X axis at its vertical center (YF = 100), there are 100 points above it and 100 below in which to represent fractional values between 0 and absolute 1. The vertical scale factor VD, then, is 1/100 = 0.01. We can add the instruction

```
130  VD = .01: YF = 100
```

to implement the Y scale in the program. This is the same as it would be in standard HRG.

We'll plot only positive angles from 0° through 360°, so the left edge

of the plot area begins at 0; in other words, the position of the Y axis (HD) is at X coordinate 0. Unlike standard HRG, MHRG has only 160 pixels (X coordinates), though it still has 320 bits with which to represent them. Still, we have to think in terms of how many horizontal points there are, and in MHRG there are 160. We have to use those 160 points to plot 360°, so the horizontal scale factor HD is 360/160 = 2.25. In other words, the angle has to increase by 2.25° before the plot advances another pixel to the right. We can put these factors into the program with

```
140  HD = 2.25: XF = 0
```

With the screen colors defined and the data scaled, we can draw the X axis using the painting subroutine at line 9000. Later we'll modify this subroutine to give it color-making capabilities, but now it will draw a line in pixel color 3 (all the bits are 1s), which happens to be white. It "thinks" it's in standard HRG, so it needs its length defined in screen coordinates; therefore, we give the instructions

```
150  Y = YF: SX = 0: EX = 316: GOSUB 9000
```

The reason we use YF instead of Y = 100 directly is that if we decide for some reason to reposition the X axis, we can change the value assigned to YF and the axis will shift accordingly.

The basis of XY plotting in any form of high-resolution graphics is contained in the subroutine located between lines 60 and 90. Type the command LIST 60–100 to display this subroutine, because it is germane to the discussion that follows.

The first thing we'll do is change its description to one that indicates what its new purpose will be:

```
60  REM ** SET COLOR PIXEL FOR XY COORDINATES
```

As currently constituted, this subroutine uses the Y coordinate and an X coordinate in the range 0–319 to calculate the byte address (BA) of the corresponding screen-memory location, and the specific bit (BP) where the dot occurs in normal HRG. It then sets that bit and RETURNs.

There are two major differences when we're in MHRG. The first is that we're dealing with 160 possible X coordinates, not 320, so we'll have to change the range check to read

```
65  IF X < 0 OR X > 159 THEN 90
```

to protect against invalid X coordinates. That part is simple. The second difference, which has to do with calculating the byte and bit positions and turning on the pixel, is less so.

MHRG uses exactly as many bits to make a screen, but they represent half as many points since each pixel takes two bits. The position that was called X = 160 (horizontal center of the screen) in standard HRG is now

called X = 80; what used to be X = 90 is now X = 45. All X coordinates are, as you see, numerically half what they were before, but they take as many bytes. We can adjust for this discrepancy in the byte address calculation by doubling the X coordinate, giving the modified statement

```
75  BA = (INT(Y/8)*320)+(INT((X*2)/8)*8)+(Y AND 7)+8192
```

The second factor (INT((X*2)/8) * 8) contains the change, and this instruction now calculates addresses using the X system for MHRG.

Lines 80 and 85 currently contain the instructions

```
80  BP = 7 - (X AND 7)
85  POKE BA, (PEEK(BA) OR 2^BP)
```

Line 80 derives a value from the low 3 bits of the X coordinate and subtracts them from 7 to get the power of 2 (used in line 85) that corresponds to the bit position. For example, if the X coordinate is 161, its low three bits are 001 in binary, or 1 in decimal. Subtracting from 7, we get BP = 6, and 2^6 is 64, which corresponds to bit 6 (second from the left) in the screen-memory byte. Line 85 then turns it on.

Things are quite different in MHRG, where each byte is zoned into four 2-bit pixels. Starting at X = 0, the first two bits are pixel 0, the second two are pixel 1, etc. Thus, if we turn on the pixel at X = 1, the byte looks like this:

X coordinate	0	1	2	3
Bits affected	0 0	X X	0 0	0 0

where XX are the two bits corresponding to pixel 1. The pixel is turned on by setting them to one of the color pointer values.

Just as the low three bits of the X coordinate helped us find the bit position, so the low *two* bits hold the key to the pixel. I didn't cleverly plan it this way, it just happens to work out. The operation X AND 3 isolates the two low-order bits: when X = 1, the byte holding the value of X is 0000 0001; when X = 81, the byte is 0101 0001. In both cases, X AND 3 masks out all but the lowest two bits, returning the value 0000 0001. This value is of little use by itself, since it indicates the pixel of the X coordinate in a left-to-right progression (pixel 0 is at the left, pixel 3 at the right). Place values, conversely, progress from right to left, so we have to reverse the result to get the place value for the pixel. This is done with

```
BP = 3 - (X AND 3)
```

which gives a place value of 3 to the leftmost pixel and 0 to the rightmost. In the case of X = 1, BP works out as 2.

In HRG we worked with individual bits, which are binary because one bit can represent either of two values, so we raised 2 to the power of the

bit position's place value and used that to turn on the bit. A pixel, on the other hand, can represent any of four values, so it is, in effect, a numbering scheme to the base 4. For that reason, we can develop a "description" of the pixel's numeric position by raising 4 to the power of the place value with

PM = 4^BP

This is called PM because it's a position multiplier. We can multiply the color number (which obviously has to be furnished to the subroutine) by PM to get the bits for the color number properly positioned in the pixel. Taking the illustration of X = 1, its BP is 2 and its PM is 16 ($4^2 = 16$). We know that the pixel is in the third and fourth bits from the left. Figure 6.33 shows how this works with each of the four colors. As you see, the multiplication shifts the color bits to the appropriate pixel.

(In these cases, X = 1 and PM = 16)

Color byte	Color #	x16	Result
0000 0000	0	0	0000 0000
0000 0001	1	16	0001 0000
0000 0010	2	32	0010 0000
0000 0011	3	48	0011 0000

Figure 6.33 Shifting color bits to the proper pixel

Our first impulse, having developed the pixel for the color number, is to POKE it into address BA and be done with it. That will work fine, except for one thing: all those extra 0s will turn off any other pixels that have already been set in the memory byte. We have to preserve them. At the same time, we have to turn off any bits that might be on in the affected pixel, since sometimes we want to change it from one color to another. That takes—you guessed it—more ANDs and ORs.

Continuing the saga of X = 1, we have the position multiplier for that pixel (PM = 16). To turn on both bits in the pixel position, multiply PM by 3 (the maximum value of two bits). This gives us the bit pattern 0011 0000. The complement (opposite bit pattern) of any byte can be found by subtracting it from 255; in this case, 255 − 48 = 239, which has the bit pattern 1100 1111. When we AND this complement with the byte now in memory address BA we get its bit pattern, except that the affected pixel contains 00. In other words, we've turned off pixel 1 and saved the others from wanton destruction. As an example, let's say

that the byte at BA holds the bit pattern 1011 0110 and we AND with 239.

```
1011  0110    (present byte)
1100  1111    (AND 239)
1000  0110    (result)
```

We can now OR this result with a color pixel previously developed to get the new byte for color memory. If the color is 2,

```
1000  0110    (result from above)
0010  0000    (OR pixel)
1010  0110    (new byte)
```

All of this elaborate background is summed up in the following, almost disappointingly brief instructions

```
80  BP = 3 - (X AND 3): PM = 4^BP: PP = 255 - (PM * 3)
85  POKE BA, ((PEEK(BA) AND PP) OR (CN * PM))
```

which you should type into the MHRG skeleton.

This revised MHRG subroutine still performs the same XY plotting task taking the same input values X and Y, but it also requires the value CN, indicating the color selection number (0–3) of the pixel at those coordinates. Figure 6.34 gives a listing of the revised pixel-plotting subroutine.

```
60 REM  **  COLOR PIXEL FOR XY COORDINATES
65 IF X < 0 OR X > 159 THEN 90
70 IF Y < 0 OR Y > 199 THEN 90
75 BA = (INT(Y/8)*320) + (INT(X*2)/8)*8)
   + (Y AND 7) + 8192
80 BP = 3 - (X AND 3): PM = 4^BP: PP = 255 - (PM * 3)
85 POKE BA, ((PEEK(BA) AND PP) OR (CN * PM))
90 RETURN
```

Figure 6.34 Pixel-plotting subroutine in MHRG

The framework is now ready to be fitted with our multicolor plotting problem, which is to draw the sine in red and the cosine in green through 360°. You'll recall that since we're working in a coordinate system other than the one the screen uses, the X coordinates are called H and the Y are called V, and the subroutine at line 1000 translates them into screen positions. The loop that calculates both of these functions and constitutes the problem-solving part of the program is in Fig. 6.35. The X axis slices across the screen, and then the two curves begin to appear, one red, the other green on the black background, just the way we planned it.

```
160 FOR H = 0 TO 359 STEP HD: A = H * π / 180
170 V = SIN(A): CN = 1: GOSUB 1000
180 V = COS(A): CN = 2: GOSUB 1000
190 NEXT H
RUN
```

Figure 6.35 Calculations for the sine/cosine plot

This is a good example of the business/scientific capabilities of the Commodore 64. Though certainly it is a machine for fun, you can also apply its broad powers to a wide variety of serious computing requirements. High-resolution multicolor graphics aren't just for getting frogs across the road and showing the Adenoids blowing up Starships.

Multicolor painting

In order to complete the adaptation of the old HRG skeleton to an MHRG programming aid, we need to overhaul the painting subroutine at lines 9000–9200. When we finish, this subroutine will be able to paint an area line by line in any color we specify.

There is no change to lines 9000–9050, and line 9060 differs only in having a different place to jump to for the short-line plot. It should be changed to read

```
9060  IF BP > (EX - SX + 1) THEN 9160
```

As a result of changing the plot subroutine at lines 60–90, the value BP it returns is now a number in the range 0–3, representing the place value of the pixel at coordinate X. Line 9050 has called this subroutine to ascertain the BP and byte address (BA) of the first pixel in the line to be painted. If the BP is 3, the first pixel of the line is at the left end of the first byte in the line, so we can save time by going directly to the whole-byte routine at line 9100. Conversely, if the BP is 0, the first pixel is at the very end of the first byte; since the subroutine at line 60 has already turned it on, it's a waste of time to have this subroutine plot it again. We can therefore advance the current X by one and the byte address by 8 and then go to the whole-byte routine. This is done with

```
9070  IF BP = 3 THEN 9100
9075  IF BP = 0 THEN CX = CX + 1: BA = BA + 8:
            GOTO 9100
```

These two lines, then, filter out cases where a partial doesn't have to be constructed at the start of the line.

For those cases where a partial does, new lines 9080 and 9090 take a different approach than the old painter did. They use a loop to plot the pixels in the specified color from the starting point to the right end of the screen memory byte, as follows:

```
9080  BR = BP: FOR PB = BR TO 0 STEP -1: X = CX
9090  GOSUB 60: CX = CX + 1: NEXT PB: BA = BA + 8
```

Note that after the loop concludes, the byte address is advanced by 8 to position the BA pointer into the next cell.

The fast whole-byte routine also takes a different tack than the old one did. The latter wrote consecutive bytes with all bits on for as many whole cells as it could. Since we want to select the color of the line, the new routine constructs an entire byte of repeating color pixels in line 9110. Lines 9120 and 9130 form a loop that POKEs this pattern into successive cells until fewer than four pixels remain to be painted.

```
9110  PB=0: FOR PV=3 TO 0 STEP-1: PB=PB+(CN*4^PV):
          NEXT
9120  BR = EX - CX + 1: IF BR < 4 THEN 9140
9130  POKE BA, PB: BA = BA + 8: CX = CX + 4: GOTO 9120
```

The end-of-line routine does another pixel-by-pixel plot from the current position to the last in the line. This same pixel plot at line 9170 is also used when the line is less than a full cell in length. The last part of the revised subroutine is

```
9140  REM ** END OF LINE
9150  IF BR <= 0 THEN 9180
9160  REM ** PIXEL PLOT
9170  FOR X = CX TO EX: GOSUB 60: NEXT X
9180  RETURN
```

Now let's test this subroutine by painting a rectangle of three different colors. Clear out lines 200 and 210 from the last program we worked, and then enter Fig. 6.36.

```
110 POKE 53280, 0: POKE 53281, 0
120 POKE 49179, 216: POKE 49172, 1: SYS 49171
130 SX = 55: EX = 106: TY = 50
140 FOR CN = 1 TO 3
150 FOR Y = TY TO (TY + 16): GOSUB 9000
160 NEXT Y: TY = TY + 17: NEXT CN
RUN
```

Figure 6.36 Painting a multicolor rectangle

The result of this program looks like a flag, with three bold stripes of red, green, and white in the correct proportions for a flag. If you watch carefully, you can see it plotting the right-end pixels individually.

We can further enhance the flag idea by adding features. Try Fig. 6.37 and see what happens.

```
170 CN = 0: Y = Y - 1
180 FOR X = SX TO EX: GOSUB 60: Y = Y - 1: NEXT X
190 Y = Y + 1
200 FOR X = SX TO EX: GOSUB 60: Y = Y + 1: NEXT X
RUN
```

Figure 6.37 Unpainting diagonal lines

This time, after the flag is painted, two lines join the opposite corners. Note that they're black; we've painted color 0 into these pixels (line 170), which in effect "unpaints" them.

In fact, "unpainting" is significantly easier in MHRG than it was in standard HRG; you just paint in color selection 0 (set individual pixels to that value) and those pixels immediately take on the background color of the screen. Figure 6.38 is an example.

```
170 Y = Y - 16: CN = 0: SX = 70: EX = 90
180 GOSUB 9000: Y = Y - 1: GOSUB 9000
190 SX = SX + 1: EX = EX - 1: Y = Y - 1
200 IF SX <= EX THEN 180
RUN
```

Figure 6.38 Unpainting an area

Now our flag acquires a black pyramid located mostly in the green stripe but with its base in the white and its tip in the red. We painted this pyramid by resetting those pixels to pixel color 0, the screen color. Thus it's not really a figure on the flag, but a hole cut into its center through which we can see the background of the screen.

Delete lines 110 through 200 by typing a list of their line numbers, and then save this new MHRG skeleton as a separate program file on tape or disk, so that you can retrieve it and use it in your own multicolor high-resolution graphics projects.

One could readily make a case for doing an entire book on the subject matter of this chapter alone. Clearly it is a technically complex topic, and we have seen only a sampling of the capabilities it offers. There is as

much potential here as there is in the artist's palette and piece of canvas, and just as the potentials of those tools are brought out by the mind and the hand of the artist, so must the possibilities of this machine be extracted by you. One must learn artistic skills by doing, and computer art is no different in that respect than any other. The only difference is the medium used for artistic expression. Don't be afraid to practice and experiment and push back the curtains we have left hanging in this chapter. You have the advantage over the painter that your failures are easier to dispose of without someone finding the evidence. You can also tear out and redo whole sections of a work more readily than anyone ever could on canvas, correcting your mistakes without a trace. And besides, it's an endlessly gratifying source of entertainment, and that's probably why you bought this computer.

CHAPTER 7

MOVIEOLA: THE MOTION PICTURE COMPUTER

A movie camera has a limited field of view, but we've all seen—literally—what can be done with it. Similarly, the screen of your Commodore 64 has a view of only a fraction of memory at a time, and we're about to glimpse what can be done with that. Specifically, this chapter will deal with two ways of making motion pictures by computer: panning the "camera" to shift its view gradually up, down, right, or left (scrolling), and either abruptly changing scenes by switching from one view to another or using several segments of memory as individual frames to create the illusion of motion (multiple screen operations).

Panning the view with scrolling

The French have the term *trompe l'oeil*, which we sometimes use on this side of the water, too. It means "a trick on the eye," and it's usually a visual joke that makes you think you're seeing one thing when, in fact, you're seeing quite another. The famous drawings of Max Escher are *trompes l'oeil* at their best. Figure 7.1 isn't in a class with Escher, but it's still a neat "trick on the eye." Type it and see what you think you're seeing.

```
NEW
100 POKE 53265, PEEK(53265) AND 247
110 PRINT CHR$(147)
```

```
120 FOR A = 1 TO 24: PRINT TAB(15) "SLIDIN' UP"
130 NEXT A
140 POKE 53265, (PEEK(53265) AND 248) + 7
150 FOR A = 6 TO 0 STEP -1
160 POKE 53265, (PEEK(53265) AND 248) + A
170 NEXT A: GOTO 140
RUN
```

Figure 7.1 Trompe l'oeil #1: What do you think you see?

What is it? An endless band of the words "SLIDIN' UP" slidin' up the screen? It sure looks like it.

But of course you know it isn't. You typed the loop in lines 120–130, and you know there are only 24 lines of print on the screen. So how do they appear to be slidin' up from some limitless supply of the same two words under the screen?

A picture is worth a thousand words, much as I, a writer, hate to admit it. Let's change this program slightly and see what it's doing. You have to hit RUN/STOP to put an end to the "slidin'," and since the cursor is probably out of sight under the bottom border of the screen (more on this shortly: stay tuned) hold down RUN/STOP while you press RESTORE. Enter these new lines:

```
120  FOR A = 1 TO 12: PRINT: NEXT A
130  PRINT TAB(15) "JIGGLIN' "
RUN
```

The word "JIGGLIN' " appears now in the center of the screen, jigglin' up and down until you wonder that it doesn't fall apart. It shows that the band of "SLIDIN' UPs" weren't really slidin' up, but were jigglin'. The only things about it that were slidin' were its top and bottom words, which slid under the screen's upper and lower borders so that you couldn't see them jiggle. *Trompe l'oeil* in action.

Let's change the direction of the trickery with Fig. 7.2

```
NEW
100 POKE 53270, PEEK(53270) AND 247
110 PRINT CHR$(147)
120 FOR A = 1 TO 12: PRINT: NEXT A
130 FOR A = 1 TO 40: PRINT "I";: NEXT A
140 POKE 53270, (PEEK(53270) AND 248) + 0
150 FOR A = 1 TO 7
160 POKE 53270, (PEEK(53270) AND 248) + A
170 NEXT A: GOTO 140
RUN
```

Figure 7.2 Trompe l'oeil #2

It looks like a bicycle chain, or maybe fenceposts viewed from the window of a moving train. Suppose we get tired of it moving left to right and decide to back it up and see whether it ever comes to an end from that direction. Push RUN/STOP and you get a rude blow: the left border of the screen is hiding the B in BREAK and the R in READY. Fix it by holding RUN/STOP while pushing RESTORE, the great panacea for all alarming things on the screen. Now type LIST and change line 150 to read:

```
150  FOR A = 6 TO 0 STEP -1
```

Run the program again and the thing—whatever it is—smoothly glides right to left across the screen. Clearly the direction of the loop in line 150 has something to do with the apparent direction of scrolling.

Again, there's no truth to the rumors that it's an endless belt. As you may have guessed, these I's scrolling horizontally are doing much as their slidin' relatives. They're jiggling from side to side, but because the same pattern repeats from border to border they appear to move like a bike chain. You can see what's happening by adding the line

```
130  PRINT: PRINT TAB(20) "I"
```

and running the program again. The I in the center of the screen below the chain looks like the kid who just can't keep up with all the others.

This is smooth scrolling, and here's how it works in a left-to-right fashion. The three low-order bits in memory address 53270 contain a horizontal positioning value. When these bits hold 0s, the screen is at its normal horizontal position. If you change them to the value 1, the entire screen shifts right one bit position. Change them to 2 and it moves right another bit, and so on. This keeps happening until the three bits are 111 (decimal 7), the maximum value they can indicate. If they now return to 0, the entire screen snaps back to the left into its normal position. These bits, then, determine the horizontal position of the screen image with respect to the left border. Note, however, that when the screen shifts, its memory addresses and bit positions don't. What changes is where they appear on the display. That's how the screen image shakes from side to side to give the impression of smooth motion to the right. To reverse the direction, set the low-order bits in 53270 to 7 and keep subtracting 1 until reaching 0, then return to 7 and resume. The rapid jerk thus goes from left to right and the slower scrolling from right to left, giving the semblance of a right-to-left flow.

A similar thing happens in vertical scrolling, but the control register address is 53265 instead of 53270. Decreasing the value of the low-order bits from 7 to 0 makes the display go slidin' up; increasing them from 0 to 7 causes slidin' down. Again the maximum amount of movement is one character cell, and then the screen snaps back into its normal position.

It's a good idea to shrink the screen when scrolling. You can move the borders onto the screen either vertically or horizontally. In horizontal shrinking, the screen narrows by one cell on each side, going from 40 to 38 columns, with the instruction

```
POKE 53270, PEEK(53270) AND 247
```

To restore the 40-column screen, undo this command with

```
POKE 53270, PEEK(53270) OR 8
```

The 38-column mode isn't required with horizontal scrolling, but if you don't do it, the jittering at the edges become apparent. By extending the border onto the screen, the ends of our chain have cover under which to jiggle without being noticeable. You can see this by deleting line 100 and running the program again.

Vertical shrinking works a little differently. The screen is changed to a 24-row display by moving either the upper or the lower border; both don't move at the same time. When you signal your intent to scroll downward, the top border covers the first row on the screen. The signal is the value existing in the low-order bits of 53265 when you shrink. If it's 0, you're going to scroll down. Conversely, if you have 7 in the position bits in 53265, the bottom border moves up. To activate vertical shrinking, the command is

```
POKE 53265, PEEK(53265) AND 247
```

and to deactivate it,

```
POKE 53265, PEEK(53265) OR 8
```

If your chain is still sliding across the screen, stop the program and type the command to deactivate horizontal shrinking given earlier. Watch the side borders as you press RETURN, and you'll see them move outward.

We can scroll one (or more) lines individually, wrapping around on the same row so that the message slides off one side of the screen and reappears on the other. Try Fig. 7.3.

As you see, the text "ROUND AND ROUND" begins in the center of the screen and works its way left. As each letter slips under the left border, it reappears again on the right border and the text thus wraps round and round. The inchworm motion is a result of BASIC's inherent slowness; soon we'll do something about that, but it's important right now to see how it's done.

Stop the program by pressing any key and LIST it so that you can follow this blow-by-blow. Lines 120–130 clear the screen and place the cursor in row 12, where line 140 prints the text. Line 150 calculates the memory offset for the start of row 12. Line 160 fetches and saves the

```
NEW
100 REM  **  ROUND AND ROUND
110 SM = 1024: CM = 55296
120 PRINT CHR$(147)
130 FOR L = 1 TO 11: PRINT: NEXT L
140 PRINT TAB(10) "ROUND AND ROUND"
150 P = 12 * 40
160 S0 = PEEK(SM + P): C0 = PEEK(CM + P)
170 FOR X = 0 TO 38
180 POKE SM+P+X, PEEK(SM+P+X+1)
190 POKE CM+P+X, PEEK(CM+P+X+1)
200 NEXT X
210 POKE SM+P+39, S0
220 POKE CM+P+39, C0
230 GET X$: IF X$ = "" THEN 160
RUN
```

Figure 7.3 Wrapping a single line

screen and color-memory bytes currently at the start of the line. The loop between lines 170 and 200 shifts the screen- and color-memory area for row 12 one character to the left, and then lines 210–220 place the byte that used to be at the start of the line into the end of the line. Finally, line 230 checks the keyboard for a signal to stop and, if it finds none, repeats the wrapping loop from line 160.

And why, you might ask, do we have to wrap color memory along with the screen line? Well, take out lines 190 and 220 and see for yourself. The text scrolls left and vanishes a character at a time in center screen, but stare at the blank display for a moment and suddenly it begins reappearing a hand's width away from the right border. The reason is that color memory contains the text color for characters written into corresponding screen-memory cells, and if we don't shift color memory with the screen, the letters lose their color and blend with the background.

You can also make nonwrapping messages that scroll across the screen in one row, just as the stock-market ticker-tape displays do. It works like this: write a character at the right edge of the screen, scroll the line left, move the cursor back up to the same row on the right edge, and repeat until the message is complete. At that point, you can simply scroll some number of times to make a space and either repeat the message, send another, or stop. Figure 7.4 gives an example.

Notice that this time part of the screen is fixed (the ** HOT NEWS ** heading) while below it the message grows out of the right

```
NEW
100 REM  **  TICKERTAPE
110 SM = 1024: CM = 55296
120 PRINT CHR$(147)
130 FOR L = 1 TO 4: PRINT: NEXT L
140 PRINT TAB(13) "** HOT NEWS **"
150 FOR L = 6 TO 11: PRINT: NEXT L
160 M$ = "CONGRESS TODAY PASSED A LAW BANNING
    RAINY WEEKENDS"
170 L = LEN(M$)
180 FOR X = 1 TO L
190 CH$ = MID$(M$, X, 1)
200 PRINT TAB(38) CH$
210 GOSUB 400
220 PRINT CHR$(145);
230 NEXT X
240 FOR X = 1 TO 39: GOSUB 400: NEXT X
250 END
400 REM  **  SCROLL ONE CHAR LEFT IN ROW 12
410 FOR S = 0 TO 38
420 P = (12 * 40) + S
430 POKE SM+P, PEEK(SM+P+1)
440 POKE CM+P, PEEK(CM+P+1)
450 NEXT S
460 RETURN
RUN
```

Figure 7.4 Ticker-tape scrolling

border and moves left across the screen. The text is wider than the screen, so when it spans the full width of the display, a character gets pushed off the left end each time a new one is added at the right. When the full text has appeared, it continues scrolling until the last character goes out of sight.

Scrolling is done at two points in this program, so we have made a subroutine at lines 400–460 to do it. This is essentially the same as the scrolling loop in Fig. 7.3 (lines 150–200) except that the memory offset calculation is a little different. Figure 7.4 builds the text display from the right with the loop at lines 180–230, which executes once for each character in string M$. Line 190 extracts the next character from M$, 200 prints it inside the right border, and 210 calls the scrolling subroutine. Line 220 is the up-cursor instruction, since line 200 went to

the next row after printing the character. Line 220 places the cursor back on row 12 so that in the next iteration of the loop, line 200 is on the correct row. The program ends after the message has completely scrolled off the screen.

High-speed full-screen scrolling

As slowly as one line scrolled in Figs. 7.3 and 7.4, the prospect of taking 25 times as long to scroll an entire screen conjures up images of a skeleton sitting at the terminal, with someone asking, "Program been running long?" Clearly, the key to fast screen scrolling lies not in BASIC, but in machine language.

The first case we'll consider is that of a Möbius scroll, in which the lines spiral around and around the display, with each wrap going to the next-higher row, and the upper-left and lower-right corners "joined." This is done by saving the first bytes in screen and color memory, shifting both memories forward one position, and moving the former first bytes to the tail ends. Each cell of the machine-language subroutine shifts the display one character position. We have to tell the subroutine where screen memory begins by POKEing its memory page number (the screen-memory address divided by 256) into address 252. Type Fig. 7.5 and see what an improvement there is; the whole screen scrolls much more rapidly and smoothly than did one line using BASIC.

```
1 REM  ** MOBIUS SCROLL
10 FOR P = 49205 TO 49265
20 READ X: POKE P, X: NEXT P
29 DATA 169, 2, 133, 2
30 DATA 169, 4, 133, 151, 169, 231, 133, 163
31 DATA 165, 252, 133, 254, 105, 3, 133, 164
32 DATA 160, 0, 177, 251, 133, 255, 200, 177, 251
33 DATA 136, 145, 253, 200, 208, 2, 230, 254
34 DATA 200, 208, 243, 230, 252, 198, 151
35 DATA 208, 237, 165, 255, 145, 163, 198, 2
36 DATA 240, 5, 169, 216, 133, 252
37 DATA 208, 200, 96
100 POKE 53270, PEEK(53270) AND 247
110 SM = 1024: BA = SM / 256
120 PRINT CHR$(147)
130 FOR L = 1 TO 6: PRINT: NEXT L
140 FOR L = 1 TO 8
150 PRINT TAB(10) ".....MOBIUS, MOBIUS....."
```

```
160 NEXT L
170 POKE 252, BA: SYS 49205
180 GET X$: IF X$ = "" THEN 170
190 POKE 53270, PEEK(53270) OR 8
RUN
```

Figure 7.5 Mobius scrolling with machine language

Lines 10–20 load into memory the machine-language subroutine defined in the DATA statements through line 37. The memory page (BA) is calculated in line 110, and then lines 120–160 produce eight repeating lines in the upper center of the display. The scrolling loop is very simple, involving only lines 170 and 180, where the screen-memory page is POKEd into 252 and the machine-language subroutine is called. BASIC then checks for a stop signal from the keyboard and, finding none, loops back to line 170 to shift by another character position. When you signal a stop by pressing a key, line 190 restores the screen to normal 40-column width.

The lines spiral upward, appearing in the next-higher row each time they slip off the left side. Finally reaching the top left corner, they wrap back down to the lower right. Watch how *fast* it goes! If we didn't let BASIC have control back after each one-character shift but instead built a loop into the machine-language subroutine itself, the scroll would progress so rapidly we wouldn't be able to read the screen.

It is, of course, possible to spiral in any combination of vertical and horizontal directions. That involves reworking the machine-language subroutine. By now you should be sufficiently convinced of the speed advantages of machine language that, when you've mastered the material in this book, you'll buy an assembler and begin creating your own library of fancy stuff. At that point you can delve further into the mysteries of the living Möbius strip of spiraling displays. (This subroutine, by the way, is listed in Appendix C.)

Meanwhile, let's develop a full-screen scroll, in which each row wraps back on itself much like the view of a camera rotating through 360°. With it, as we'll see in a moment, we can create the interesting illusion of horizontal motion.

A full-screen scroll is basically the same as a Möbius, but with an added step. For that, we'll consider just one line on the display and then extend our reasoning to all lines. When we scrolled left in the Möbius, each byte shifted to the address numerically 1 lower than its current position, and the zeroth byte was carried to the 999th place. If Row 12 on the display has an A in the first position and a C in the last, when we scroll the screen the A should move to the end of Row 12 and appear after the C, but in a Möbius, A will instead appear in the last position of Row 11.

Consequently, we'll have to move it "down" one full row, or in other words, advance it to an address numerically 40 greater than its address after the Möbius scroll, thus shifting it from the end of Row 11 to the end of Row 12. The same is true of every other row on the display; the character that should be at the end of Row 24 is instead at the end of Row 23 and the one that should be at the end of Row 1 is at the end of Row 0. To correct this, use the following process to step through screen memory backwards by 40s:

```
FOR Y = 23 TO 0 STEP -1
P = (Y * 40) + 39
POKE SM + P + 40, PEEK(SM + P)
POKE CM + P + 40, PEEK(CM + P)
NEXT Y
```

And what of the top left byte that goes to the end of Row 0? The Möbius scroll moved it to the lower right corner of the screen in Row 24, so before starting the loop we need to set it aside and, afterwards, place it at the end of the top line.

To illustrate this, we'll go scrolling down the avenue. Back in the Dark Ages—before the microcomputer was invented in the 1970s—there was a popular song entitled "Ticky Tacky" that decried the monotony of mass-produced tract houses "all in a row." When we built that house back in Chapter 3, we failed to mention that it was only one of many identical homes along Ticky Tacky Avenue.

Before we continue, let's pause to discuss the nature of machine language. When you run a program, such as that in Fig. 7.5, that loads a machine-language subroutine into memory, the subroutine remains unchanged in memory as long as the machine is powered on and you don't alter its locations. With BASIC, you can delete a program by typing NEW or LOAD, but that doesn't affect the machine-language instructions it might have written elsewhere the last time it ran. The benefit of this is that you can run programs to set up machine-language subroutines and different BASIC programs that use them.

As a case in point, delete all lines from 100 onward from Fig. 7.5 and save the resulting program with the name MOBIUS. Now turn the computer off and back on again, thus wiping its memory clean. Load and run MOBIUS to write the scrolling subroutine into memory and then load the HOUSE program you saved in Chapter 3.

If you now run HOUSE "as is," the machine-language subroutine has no effect because HOUSE doesn't call it. For that we have to make some modifications, as shown in Fig. 7.6.

As soon as we've watched the house go up and the smoke appear from the chimney, we seem to be walking casually along the avenue, looking

```
105 POKE 53270, PEEK(53270) AND 247
125 SM = 1024: CM = 55296: BA = SM / 256
860 REM  **  SCROLL DOWN THE AVENUE
870 POKE 252, BA: SYS 49205
880 B0 = PEEK(SM + 999): C0 = PEEK(CM + 999)
890 FOR Y = 23 TO 0 STEP -1
900 P = (Y * 40) + 39
910 POKE SM + P + 40, PEEK(SM + P)
920 POKE CM + P + 40, PEEK(CM + P)
930 NEXT Y
940 POKE SM + 39, B0: POKE CM + P, C0
950 GET X$: IF X$ = "" THEN 870
960 POKE 53270, PEEK(53270) OR 8
RUN
```

Figure 7.6 Scrolling down Ticky Tacky Avenue. (Changes to the HOUSE program from Chapter 3)

to the left at an endless progression of identical homes. We can stop the procession by pressing the SPACE bar.

The houses we see are actually, of course, all the same house scrolling round and round. The "locomotion" is provided by the loop at lines 870–950, which calls the MOBIUS subroutine to shift the screen left and then move all the right-end bytes down one row, and the bottom one to the top. You can see this happening by stopping the program and removing line 105, then running it again. The right border no longer covers the end bytes, so you can see them rippling downward via lines 910 and 920.

It would have been possible to modify the machine language to perform this realignment of the right edge as BASIC now does, but that would have deprived you of the opportunity to see how it's done. The slowness of BASIC causes the delays that emulate a walking pace; if machine language were rearranging the ends of the rows, the scroll would be much faster and smoother, giving a view from a car driving down Ticky Tacky Avenue.

It's possible to scroll also in both of the high-resolution-graphics modes. That is the stuff of video games, but because it takes a lot of machine language to carry it off, it's beyond the scope of this book. The main complications have to do with the cellular structure of the display and the one-to-eight discrepancy between color and screen memories, plus the requirement of most video games to scroll in all directions and at angles that combine vertical and horizontal scrolls. Our intent here has

been to give a general view of scrolling so that you can extrapolate the concepts to serve your own programming goals.

A sudden change of camera angles: using multiple screens

In movies and television dramas, abrupt changes of camera angles are so commonplace we aren't even aware of them. Two people are talking and the view jumps from one to the other as they exchange lines; the location of action leaps from one coast to the other over a period of years, and we know that because the camera angle and scene change in a twinkling. We don't think of these typical tricks of visual entertainment as unrealistic, even though to watch two people conversing we have to swing our heads to and fro, or though we have just hurtled across thousands of miles and part of a lifetime against all the laws of the universe. Unless you're in that business, you've probably never given a thought to how it's done.

The answer won't be found here, though we will find out how to do similar things with the eye of the Commodore 64 through the use of multiple-screen memories. By switching from one to another we can instantly change the screen. We can also use multiple screens as the individual frames of movie film to create realistic action and animation on the screen.

But first some technical background. Remember VIC, the busy fellow who translates memory into displays? VIC is not lazy, he's just overworked, and as a result he takes the position that he can only pay attention to 16K of memory at a time. He carves the memory into four such hunks, which go by the name *banks*. A bank is 16,384 bytes, or exactly one-fourth of the Commodore 64's total memory capacity, and that is how much VIC sees at one time. Each bank begins on an even multiple of 16K and has a reference number in the range Ø through 3. Figure 7.7 is a table showing the relationships of bank numbers, addresses, and selection values.

Bank number	Address range	Select bits
Ø	Ø–16383	11 (3)
1	16384–32767	1Ø (2)
2	32768–49151	Ø1 (1)
3	49152–65535	ØØ (Ø)

Figure 7.7 Video banks in the Commodore 64

Note that the "select bits" are the threes-complement of the bank number. That is, to calculate the select bits, subtract the bank number from 3. These bits are then written to the two low-order bit positions of address 56576 with the instruction

```
POKE 56576, (PEEK(56576) AND 252) OR SB
```

where SB is the value of the select bits. For example, to swing VIC's camera angle from its current view to Bank 2, type

```
POKE 56576, (PEEK(56576) AND 252) OR (3-2)
```

Huh? The screen display immediately turns to garbage. That's because VIC is now looking at memory in Bank 2, where there is nothing comprehensible for him to display. It's like the camera that, though turned on by accident, nevertheless broadcasts the set that isn't yet ready to go on the air. You can restore decency and order to the screen in either of two ways. The easiest is to hold RUN/STOP and press RESTORE. The other is working in the blind, but it confirms that a computer derives no sense from what it displays: type the command

```
POKE 56576, (PEEK(56576) AND 252) OR 3
```

and, though you see garbage appearing in response to your keystrokes, when you press RETURN the screen immediately becomes normal as if nothing had ever happened. In fact, the command just entered also appears.

The correlation of the bank number, its base address, and its select bits is mathematically direct and simple, and if you know any one of them, you can derive the others. We've already seen how to get the select bits from the bank number. To get the base address, multiply the bank number by 16384. Conversely, to find the bank number from a known base address, divide by 16384 and use the result to calculate the select bits. This simplifies programs by allowing you to specify one value associated with the bank(s) and symbolically derive all the others. We'll see the concept in action presently.

There are, as you might expect, some terms and conditions associated with using different banks. The default video bank is number 0; it's always in effect unless you specify otherwise, and when you reinitialize the computer (with RUN/STOP and RESTORE), you point VIC back at Bank 0 no matter where he was looking before. Banks 0 and 2 have access to the character generator ROM, giving you normal character graphics and alphanumerics in those banks. Banks 1 and 3 do *not* automatically have a doorway to the ROM, and if you PRINT or POKE information there expecting to see something you understand, you'll be disappointed. As an experiment, try the program in Fig. 7.8.

```
NEW
100 POKE 56576, (PEEK(56576) AND 252) OR 2
110 PRINT CHR$(147)
120 PRINT "WELCOME TO BANK 1"
130 Y = 9: SM = 1024 + 16384: CM = 55296
140 FOR X = 1 TO 26: P = (40 * Y) + X
150 POKE SM + P, X: POKE CM + P, 12
160 NEXT X
170 GET X$: IF X$ = "" THEN 170
180 POKE 56576, (PEEK(56576) AND 252) OR 3
RUN
```

Figure 7.8 Noncharacters in video bank 1

What you see is the result of presenting VIC with codes he can't look up in the character ROM tables, since they are unavailable in Bank 1. After you get tired of looking at the blobs, push the SPACE bar to restore the screen to Bank 0. The point of this exercise is that Bank 1 is no good for character output (unless you copy the ROM into it as discussed in Chapter 4). Bank 1 *is* a useful place to do high-resolution graphics work, because the entire 16K is fully available without restrictions.

You should *never* use Bank 3 for screen memories, for the very good reason that the Commodore 64 stores control values of all kinds in this bank and you'll be in grave difficulty if you unwittingly overlay them with screen images. Only the lower 4K (4096 bytes) of Bank 3 are available for your use, enough room for four character screens, but only half the space for an HRG display. This area is better used for your machine-language subroutines (and all those in this book reside there).

There are also some reserved areas in Bank 0 that Thou Shalt Not Clobber. The addresses from 0–1023 are owned by the machine's operating system for sundry control purposes, and your BASIC program lives between 2048 and about 8192, depending on its length and the data workspaces it requires. This leaves 1024–2047, with which we are now intimately familiar, and 8192–16383, which we have also traversed in HRG and MHRG.

From all this we can draw some general guidelines for the selection of video banks:

- Use Bank 0 for normal alphanumeric displays.
- Use Bank 0, addresses 8192–16383, for single-screen high-resolution graphics work.
- Use Bank 1 for multiple-screen HRG and MHRG (see note).
- Use Bank 2 for multiple-screen displays requiring the machine's character set.
- Don't use Bank 3 at all.

Note: In HRG and MHRG, which use 1024–2023 for color memory, these addresses are offset by the bank's base address. Thus, in Bank 1, color memory for HRG/MHRG is between 17408 and 18407.

Just as VIC divides total memory into 16K banks on fixed boundaries, so does he subdivide the banks in 1K (1024-byte) intervals. Each 1K unit has a reference number in the range 0–15, and can be used either for screen memory or character-image memory. When VIC is preparing to send a display to the screen, he looks at the bank number in 56576 and the unit number in the high-order nibble of 53272 and translates these two pointers into the real memory address of the display image. He also checks the mode bit (character graphics or HRG) in 53270 to find out whether screen memory is 1000 bytes or 8000, and if the machine is in character mode, he gets the character-image pointer from 53272 as described in Chapter 4. Armed with this and other relevant information—such as the screen background and border colors—he builds the display and sends it to the video device.

The 1K unit reference number always relates to the position of the unit with respect to the bank's starting address. Thus, Unit 1 begins at address 1024, but if it's in Bank 1, its real address is 16384 + 1024 = 17408, and if in Bank 3, its real address is 49152 + 1024 = 50176. This seems like a lot of gobbledygook and complication, but you can readily solve the problem with a few programming instructions using the following approach:

```
100  BN = 2: UN = 8
110  SM = (BN * 16384) + (UN * 1024)
```

in which BN is the bank number, UN is the unit number, and SM is the consequent screen-memory address where you will POKE your graphics codes. In this example, SM begins at address 40960.

The upper four bits of address 53272 hold the screen-memory unit pointer. You'll recall from Chapter 4 that the lower nibble of this same byte holds the character-image pointer. Whenever you alter this byte, you have to treat it with tender loving care to avoid repointing the other half, which calls for a precautionary "AND 15" while setting the screen-memory portion and an "AND 240" while changing the character pointer. Because the screen pointer is in the upper nibble, you have to multiply it by 16 to shift it there and OR that value with the still-intact lower nibble. To select Unit 8,

```
POKE 53272, (PEEK(53272) AND 15) OR (8 * 16)
```

If you are also changing banks, that takes a separate instruction that can either immediately precede or follow this one. The screen-memory selection table in Fig. 7.9 resembles the character-image selection table in Fig. 4.1.

Command to change the pointer in BASIC:
POKE 53272, (PEEK(53272) AND 15) OR (UN * 16)

Location in bank	Bit setting in 53272	UN	
0	0 0 0 0 X X X X	0	(Note 1)
1024	0 0 0 1 X X X X	1	(Note 2, 3)
2048	0 0 1 0 X X X X	2	(Note 2)
3072	0 0 1 1 X X X X	3	(Note 2)
4096	0 1 0 0 X X X X	4	(Note 2, 4)
5120	0 1 0 1 X X X X	5	(Note 2, 4)
6144	0 1 1 0 X X X X	6	(Note 2, 4)
7168	0 1 1 1 X X X X	7	(Note 2, 4)
8192	1 0 0 0 X X X X	8	(Note 4)
9216	1 0 0 1 X X X X	9	(Note 4)
10240	1 0 1 0 X X X X	10	(Note 4)
11264	1 0 1 1 X X X X	11	(Note 4)
12288	1 1 0 0 X X X X	12	(Note 4)
13312	1 1 0 1 X X X X	13	(Note 4)
14336	1 1 1 0 X X X X	14	(Note 4)
15360	1 1 1 1 X X X X	15	(Note 4)

Notes:
1. Do not use in Bank 0. Belongs to Kernal.
2. Do not use in Bank 0. Belongs to BASIC.
3. Color memory in HRG/MHRG modes.
4. Unavailable in Bank 3.

Figure 7.9 Selecting the screen-memory unit location

From this chart there are a few general observations that we can make. First of all, be careful about assigning screen memories in Banks 0 and 3, since many units are already committed. Secondly, the Commodore 64 will support up to 43 simultaneous screens in character-graphics mode (9 in Bank 0, 14 in Bank 1, 16 in Bank 2, and 4 in Bank 3), but, thirdly, only three in HRG/MHRG mode. This is because high resolution requires Unit 1 for color memory and 8000 consecutive bytes for screen memory. The 8K screen memory can begin on any unit boundary above 2023, but the available 14K falls short of the room needed for even two HRG screens in the same bank. Also, Bank 3 simply hasn't enough space available for any HRG screen. Thus, if your computer drama calls for a lot of scene changes, either you'll have to use character graphics or else your program will have to modify screens that are out of view while at the same time carrying out action on the one that is visible. That's pretty sophisticated

stuff, beyond the scope of this book and far exceeding the speed limits of BASIC. Nevertheless, we can do some nifty things, as the following couple of exercises demonstrate.

The first cinematic drama we will play out is entitled *The Waving Robot,* an unending tale of mechanistic friendliness. In this gripping story, a robot will be built behind the scenes and then it will burst into view, waving at us from the depths of the Commodore 64. The story will illustrate—though you won't be visually aware of it—the aiming of the camera at one scene while out of sight other scenes are being prepared, and the use of multiple screens to convey a continuum of motion in much the same way as do the separate frames of a movie film.

Before you can make a movie, you must, of course, have a plot outline. This is the purpose of Fig. 7.10, which shows the robot and the three positions of its waving arm. From the outline, and indeed that's exactly what it is, we can write a script in the form of a computer program. At the conclusion of that step, we'll actually film and screen this Academy Award nominee.

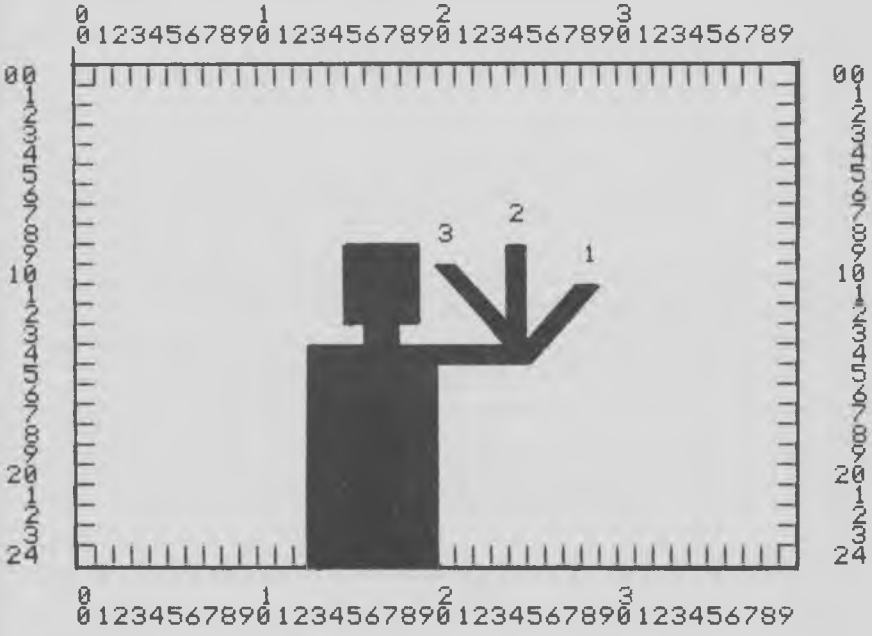


Figure 7.10 Plot outline of *The Waving Robot*

Every film has to have a title screen and a certain amount of setup, and that's what Fig. 7.11 does. Of the most long-range significance is line 110, which defines the bank number and the three screen units that will

be the "sets" for the story. The rest of these instructions create a title screen that hides the busy but orderly behind-the-scenes filming.

```
NEW
100 X$ = "THE WAVING ROBOT": DL = 70
110 BN = 2: S1 = 1: S2 = 2: S3 = 3: CM = 55296
120 POKE 53280,0: POKE 53281,9
130 PRINT CHR$(147): PRINT: PRINT
140 PRINT TAB(12) X$
```

Figure 7.11 *Waving Robot* title and setup

As in all plots involving graphics, it's useful to have a common subroutine for placing elements of pictures and colors. For this, we'll write Fig. 7.12. This subroutine requires that certain values be furnished: X and Y, which are the column and row coordinates we know and love; SM, the screen-memory starting address; and CN, a color number. The value CM is already declared as the color memory at 55296.

```
150 GOTO 200
160 REM  ** PLOTTING THE SCHEME
170 P = (Y * 40) + X
180 POKE SM+P, GC: POKE CM+P, CN
190 RETURN
```

Figure 7.12 Plotting subroutine for *Waving Robot*

Now we're ready to build the robot. The overall strategy is to make its basic form in Screen 1 (less the waving part of the arm) and then copy it to Screens 2 and 3. First we need to establish where this is to happen and to clear any clutter from the viewing area with Fig. 7.13.

```
200 REM  ** BUILD THE BASIC ROBOT
210 SM = (BN * 16384) + (S1 * 1024)
220 CN = 9: GC = 32
230 FOR Y = 3 TO 24
240 FOR X = 0 TO 39: GOSUB 160
250 NEXT X: NEXT Y
```

Figure 7.13 Clear the way for building the robot

With that out of the way, we can use Fig. 7.14 to construct a robot of black squares. Lines 270–300 shape the head and neck and line 310

makes the shoulders and the arm out to the elbow. Lines 320–340 fashion the body.

```
260 GC = 32 + 128: CN = 0
270 FOR Y = 9 TO 12
280 FOR X = 15 TO 18: GOSUB 160
290 NEXT X: NEXT Y
300 Y = 13: FOR X = 16 TO 17: GOSUB 160: NEXT X
310 Y = 14: FOR X = 13 TO 24: GOSUB 160: NEXT X
320 FOR Y = 15 TO 24
330 FOR X = 13 TO 19: GOSUB 160
340 NEXT X: NEXT Y
```

Figure 7.14 Building the robot

This completes the fixed portion of the robot in Screen 1. Figure 7.15 copies it to the other two screens, which will form the three frames of the film.

```
350 D2 = (BN * 16384) + (S2 * 1024)
360 D3 = (BN * 16384) + (S3 * 1024)
370 FOR Y = 0 TO 24
380 FOR X = 0 TO 39
390 P = (Y * 40) + X: T = PEEK(SM + P)
400 POKE D2 + P, T: POKE D3 + P, T
410 NEXT X: NEXT Y
```

Figure 7.15 Copy robot to other two screens

It's important to note that we are working with absolute memory locations, not relative bank and unit numbers. BASIC needs to know the real addresses, whereas VIC, when he displays the frames, only needs the pointers to the bank and unit. Consequently, lines 350 and 360 develop screen-memory addresses (SM already contains the address for Screen 1), and the first part of line 390 figures out the current position. Because the other two screens are exact replicas of Screen 1, P points to the corresponding position in each one. Lines 370–410 copy the figure.

Screen 1 has the left forearm extended away from the head. We fit it in this frame with Fig. 7.16, which forms an arm out of alternating right triangles. In Screen 2 the forearm is in the upright position, and is put in place by Fig. 7.17. The arm in Screen 3 bends back toward the head. The method for making it, shown in Fig. 7.18, resembles that in Screen 1.

```
420 REM  **  ADD ARM IN S1
430 X = 25: FOR Y = 13 TO 11 STEP -1
440 GC = 105 + 128: GOSUB 160
450 X = X + 1: GC = 105: GOSUB 160
460 NEXT Y
465 X = 25: Y = 14: GOSUB 160
```

Figure 7.16 Waving arm #1

```
470 REM  **  ADD ARM TO S2
480 SM = D2: GC = 32 + 128: X = 24
490 FOR Y = 9 TO 13: GOSUB 160: NEXT Y
```

Figure 7.17 Waving arm #2

```
500 REM  **  ADD ARM TO S3
510 SM = D3: X = 20
520 FOR Y = 10 TO 13
530 GC = 95: GOSUB 160
540 X = X + 1: GC = 95 + 128: GOSUB 160
550 NEXT Y
```

Figure 7.18 Waving arm #3

We have now fashioned three screens holding the image of the robot, each with the forearm in a different position. The filming is complete, and now we can roll the projector. The action begins in Fig. 7.19, which makes sure there's no junk in the top of color memory and then switches to Bank 2.

```
560 GC = 32: CN = 9
570 FOR Y = 0 TO 3
580 FOR X = 0 TO 39: GOSUB 160
590 NEXT X: NEXT Y
600 BB = 3 - BN
610 POKE 56576, (PEEK(56576) AND 252) OR BB
```

Figure 7.19 Clear top of color memory

And so at last we come to the action itself. The arm oscillates by playing the three frames back and forth in the sequence 1-2-3, 3-2-1, 1-2-3, etc.

Each time we move to the next frame, we switch the VIC's screen pointer. The usual BASIC POKE-iness is not a problem here, and, in fact, we have to slow it down with a delay loop lest the arm appear to wave frantically instead of in casual greeting. We can halt the action and restore the screen to normal mode by pressing the SPACE bar. Figure 7.20 shows how it's all accomplished.

```
620 REM  **  ROLL 'EM
630 FOR S = S1 TO S3: SS = S * 16
640 POKE 53272, (PEEK(53272) AND 15) OR SS
650 FOR T = 1 TO DL: NEXT T
660 NEXT S
670 FOR S = S3 TO S1 STEP -1: SS = S * 16
680 POKE 53272, (PEEK(53272) AND 15) OR SS
690 FOR T = 1 TO DL: NEXT T
700 NEXT S
710 GET X$: IF X$ = "" THEN 620
720 POKE 53272, (PEEK(53272) AND 15) OR 16
730 POKE 56576, (PEEK(56576) AND 252) OR 3
RUN
```

Figure 7.20 . . . action, camera!

The delay in the title screen is due to the work going on behind the scenes. When action begins, notice that there's no flicker or any other indication that we are seeing three different screens. The arm merely waves in mechanical nonchalance. (You can change the rate of motion by altering the value DL in line 100.) While the star is neither glamorous nor the scenery arresting, this example clearly illustrates how multiple screens can introduce the illusion of motion into graphics in cinematic fashion. And the sudden evaporation of the title screen shows how scenes can change abruptly.

You can, of course, use multiple screens in the HRG and MHRG modes, subject to the limitations discussed in connection with Fig. 7.9. To do this, it is necessary to modify line 75 in the skeleton (whichever one you are using) to replace 8192 with the formula that converts the bank number and screen unit into the actual memory address. That is the only change required.

If you've played video games, you've seen the lengths to which full-screen animation can be carried. Here we've used simplistic examples, but all the principles are present and accounted for, waiting for you to apply them to your own more ambitious projects.

But we haven't yet covered the full scope of animation. The next two chapters delve into screen-independent action figures ("sprites") and external control over events on the screen via joysticks. These three topics—the two just named plus the full-screen movieola effect—encompasses the whole of the advanced computer graphics available on the Commodore 64.

CHAPTER 8

WHITHER YON SPRITELY APPARITION?

Meanwhile, as we have been pointing the camera elsewhere, the Adenoids' flying saucer, flushed with victory from destroying the Good Guys' Starship, has reached the remote moon of Darthia, where it is preparing to land. The captain enters the landing instructions in Fig. 8.1 into his Commodore 64.

```
100 REM  **  FLYING SAUCER
110 DATA 0, 126, 0, 1, 255, 128, 1, 255, 128
120 DATA 15, 255, 240, 127, 255, 254
130 DATA 255, 255, 255, 255, 255, 255
140 DATA 127, 255, 254, 127, 255, 254
150 DATA 15, 255, 240, -1
160 REM  **  LOAD SPRITE IMAGE AT 12288
170 P = 0: SL = 12288: SP = SL / 64
180 READ X: IF X < 0 THEN 200
190 POKE SL + P, X: P = P + 1: GOTO 180
200 IF P >= 63 THEN 220
210 POKE SL + P, 0: P = P + 1: GOTO 200
220 POKE 2047, SP
230 POKE 53294, 2
240 POKE 53281, 0: POKE 53280, 0: PRINT CHR$(147)
250 REM  **  LUNAR TERRAIN
260 FOR Y = 24 TO 22 STEP -1
```

(Continued)

```
270 FOR X = 0 TO 39: P = (40 * Y) + X
280 POKE 1024 + P, 32 + 128: POKE 55296 + P, 12
290 NEXT X: NEXT Y
300 REM  **  LAND THE SAUCER
310 POKE 53263,0: POKE 53277, 128: POKE 53269, 128
320 X = 0: Y = 80: H = 3: V = 0.7
330 X = X + H: POKE 53262, INT(X + .5)
340 Y = Y + V: POKE 53263, INT(Y + .5)
350 H = H - .02: IF H < 0 THEN H = 0
360 IF Y < 223 THEN 330
998 GET X$: IF X$ = "" THEN 998
999 POKE 53269, 0: PRINT CHR$(147)
RUN
```

Figure 8.1 Landing instructions for the Adenoids' flying saucer

The desolate plain of Darthia's surface appears against the profound blackness of the end of the universe, and then the red saucer of the Adenoids comes into view at the top left corner of the screen. It rapidly decelerates along a curving downward path until finally it comes gently to rest on the moon. There it will remain until you tap the SPACE bar.

What we have just seen is a *sprite*, a visual object that, like mobile objects in reality, has an existence independent of the background against which it moves and likewise is independent of other moving objects. The Commodore 64 can support up to eighty sprites on the screen at one time, and these sprites can move freely and separately, can have multiple colors, can be expanded horizontally (as the Adenoid's saucer is) and/or vertically, and provide a number of "services," such as collision detection, partial transparency, passing in front of or behind background objects, animation while moving, and. . . . Well, sprites are remarkably powerful, as we're about to discover.

Let's dissect the landing instructions to get a general idea of how to create and control sprites. Lines 110–150 define the shape of the sprite (more on this later), which always takes 63 bytes. Lines 170–190 read and build the definition, and lines 200–210 complete the sprite's 63-byte image by filling unused spaces with 0s. Line 220 tells the VIC-II chip where to find the image in memory, and line 230 assigns it the color red. The program then sets up the screen in black and creates the lunar plain in a familiar manner (lines 240–290). Line 310 expands the sprite to twice its normal width and turns it on. The loop between 330 and 360 slides the sprite steadily downward at a decreasing horizontal rate until finally the saucer is moving only vertically as it comes in for a landing. Line 998 holds the display until a signal arrives from the keyboard, at which time line 999 turns off all sprites and clears the screen.

To illustrate the sprite's independence from the background, let's do an experiment. Retype line 999 to read

```
999  PRINT CHR$(147)
```

and run the program again. It behaves just the same until you press the SPACE bar, and then, although the background clears, the saucer remains where it was. It is unimpressed by the clear-screen command PRINT CHR$(147). Type some garbage on the screen, space the cursor down to the vicinity of the sprite, and type more garbage out past where the saucer sits. Notice that the letters go behind it and that parts of them peek out around the edges. Now press the RETURN key to make the screen scroll upward, and watch: the letters slip from behind the sprite and move up as if they had been perfectly visible all the time. Also, even though the rest of the screen scrolls, the sprite ignores it and stays in its position. The only way to make this thing go away is to type the clear-sprite command

```
POKE 53269, 0
```

The flying saucer is an expanded sprite, meaning that it is double its normal size along one or both axes: in this case, horizontally. To get it back on the screen, type

```
POKE 53269, 128
```

and then, to "unexpand" it, enter the command

```
POKE 53277, 0
```

It should now look like an old-time car or perhaps a World War I tank painted red. It is three character cells wide and slightly more than one cell high, which is the full width of an unexpanded sprite and about half its full height. We will discuss expansion in more detail presently.

Sprite definition

A normal sprite occupies a space 24 bits wide by 21 bits high. Sprites are always in HRG or MHRG mode, regardless of the mode of the screen itself. Just as in screen HRG, each bit represents a dot on/dot off, depending on whether it is 1 or 0, respectively; a multicolor sprite is in MHRG mode and is constructed of 2-bit pixels in the same manner as screen MHRG. I'll deal first with standard HRG sprites, and discuss multicolor sprites later in this chapter.

Since the size of the sprite is 24 × 21 dots, this works out to 63 bytes

to hold its image. The pattern of bytes is

Byte 0	Byte 1	Byte 2
Byte 3	Byte 4	Byte 5
.	.	.
.	.	.
.	.	.
Byte 60	Byte 61	Byte 62

Note that this pattern differs from HRG character cells, where the bytes are stacked eight deep. A 1 in bit 7—the MSB—of byte 0 makes a dot in the upper-left corner of the sprite area; a 1 in bit 0—the LSB—of byte 62 makes a dot in the lower-right corner of the sprite. Wherever a 0 bit occurs in the sprite image, the sprite is transparent and whatever is behind it shows through. Patterns, then, are made by turning on 1 bits at the appropriate points in the 63-byte image, and wherever a 1 bit appears, a dot of the sprite color turns on.

There are several methods for defining a sprite. The first I'll discuss closely resembles that for the Persian rug floret we needlepointed in Chapter 6; only the numeric order of bytes differs. Use graph paper and mark off an area of 24 columns by 21 rows, numbering them as shown in Fig. 8.2. Draw heavy lines between columns 7 and 8 and between columns 15 and 16 to denote byte boundaries, then lay out the pattern of the sprite by marking squares with X's. Each X represents a 1 bit, and you can

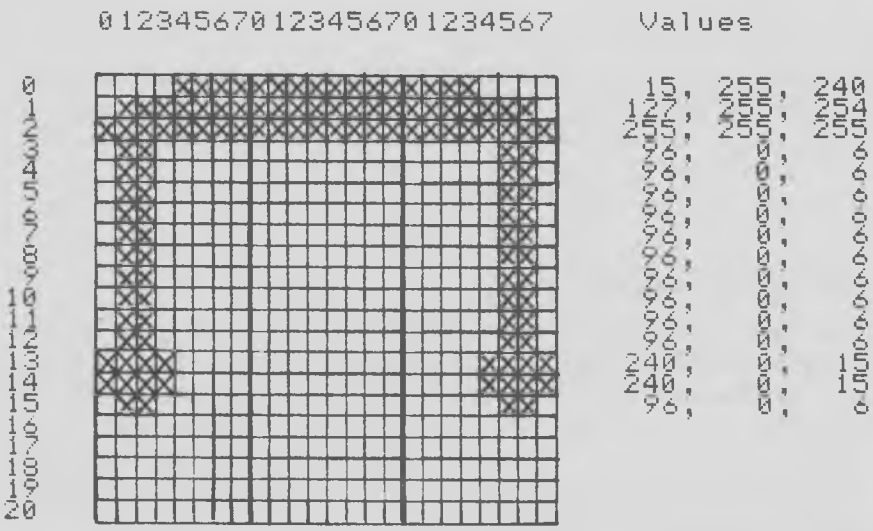


Figure 8.2 Defining the Adenoids' saucer-shelter sprite

use them and the byte boundries to calculate the decimal value of each of the 63 bytes comprising the sprite.

As an example, let us suppose that the Adenoids have a movable flying-saucer shelter on the bleak surface of Darthia. It is merely a roof resting on wheeled pillars to protect the saucer from counterattacks by the Good Guys. When the saucer lands, the shelter rolls over it. Fig. 8.2 defines this shelter. The values for the bytes that constitute the visible part of the sprite are listed on the right, having been calculated from the bit positions marked with X's.

To add the new shelter to the lunar scene, we'll tack a subroutine onto the end of the program, starting at line 800. This subroutine defines the sprite and loads its image into memory, as shown in Fig. 8.3.

```
235 GOSUB 800
799 GOTO 998
800 REM  **  SAUCER SHELTER
810 DATA 15, 255, 240, 127, 255, 254, 255, 255, 255
820 DATA 96, 0, 6, 96, 0, 6, 96, 0, 6, 96, 0, 6
830 DATA 96, 0, 6, 96, 0, 6, 96, 0, 6, 96, 0, 6
840 DATA 96, 0, 6, 96, 0, 6, 240, 0, 15
850 DATA 240, 0, 15, 96, 0, 6, -1
860 REM  **  IMAGE AT 12352
870 P = 0: SL = 12352: SP = SL / 64
880 READ X: IF X < 0 THEN 900
890 POKE SL + P, X: P = P + 1: GOTO 880
900 IF P >= 63 THEN 920
910 POKE SL+P, 0: P = P + 1: GOTO 900
920 POKE 2040, SP: POKE 53287, 6
930 POKE 53248, 100: POKE 53249, 219
940 RETURN
310 POKE 53263, 0: POKE 53277, 128: POKE 53269, 128 + 1
999 POKE 53269, 0: PRINT CHR$(147)
RUN
```

Figure 8.3 Building the shelter sprite on the moon

The shelter we have created now stands toward the lower-left corner of our view, somewhere this side of the horizon and at a distance from the saucer. This sprite is unexpanded to illustrate that sprites have independent attributes, whereas the saucer is double normal width.

In Fig. 8.3, the DATA statements are taken directly from the worksheet in Fig. 8.2. Lines 860–920 are almost exact copies of lines 160–210 in the original program in Fig. 8.1, and had we not tacked this shelter on now but instead included it first, we would have made this

instruction sequence a subroutine. (We are focusing on sprites here, not programming practices.) Lines 920–930 set up the pointers and control values for this sprite, at locations that will be discussed presently.

A sprite definition has 63 bytes, but if you count the bytes in Fig. 8.2, you'll find only 48, and 49 in DATA statements 810–850. This is not an oversight, but instead a programming shortcut that saves the trouble of typing long strings of 0s. Lines 880 and 900–910 in Fig. 8.3 handle this for us, filling in the rest of the sprite image with 0s after −1 is encountered in DATA. I occasionally use this trick throughout the chapter, and also sometimes fill in the top of a sprite definition with a fixed number of 0s before READing DATA.

Although it takes 63 bytes to describe a sprite, lines 170 and 870 divided the address of the sprite description by 64. This is because an extra byte is added to the end of the sprite image as a "placeholder," a value whose only purpose is to reserve space. In this case, the placeholder increases the length of the sprite definition to 64, which is a round number in the computer's internal hexadecimal numbering system. By placing sprite images in 64-byte blocks, we can locate them on 64-byte intervals in memory and refer to them not by their complete address, but by an abbreviated address that is the result of dividing the real address by 64. That is what the value SP represents; in line 170, SP = 12288 / 64 = 192, its abbreviated address. To find a sprite, VIC multiplies its abbreviated address by 64. Why? Because then it takes only one byte, and not 2, to hold the sprite pointer.

Sprite priorities

To avoid having to staff all their shelters throughout the universe, the Adenoids have automated them with remote controls that can be activated from saucers. The captain therefore types the instructions in Fig. 8.4 to position the shelter over his craft.

```
370 REM  **  SHELTER THE SAUCER
380 D = X - 8
390 FOR X = 100 TO D: POKE 53248, X: NEXT X
RUN
```

Figure 8.4 Instructions to move the shelter

Now, as soon as the saucer has come to rest, the shelter quickly rolls over to it. It stops a little short of covering the saucer, however, so that you can see an interesting feature of sprites. Look under the shelter and

you'll notice that, between the legs, you can see the gray of the moon's surface, the black of the sky, and the red end of the saucer. This is because, in Fig. 8.2, the areas between the supports is all 0s, and wherever 0s appear in the sprite definition, the sprite is transparent to what lies behind it. What lies beyond in this case is the soil, the sky, and part of the saucer. You can now make the shelter cover the center of the spacecraft by changing line 380 to read

```
D = X + 12
```

and rerunning the program.

But, one might ask, why is the saucer *behind* the shelter legs? The answer is that sprites have priorities in relation to each other. There can be up to 8 sprites visible at one time, numbered 0 through 7. Sprite 0 has the highest priority, 7 the lowest. When two sprites occupy the same space, the one with the higher priority moves "in front of" the other. In this case, the shelter is Sprite 0 and the saucer is Sprite 7, so the shelter legs are on this side of the saucer.

The priority of each sprite is inherent in its number and cannot be overridden: no matter what, Sprite 4 is *always* lower in priority than Sprite 3, Sprite 7 is *always* on the bottom of the hierarchy, and Sprite 0 is *always* the one that comes out in front when it meets another. The concept of priority runs throughout spritedom in setting control bits, locating pointers, and other programming concerns, not so much from the viewpoint of relative importance, but for purposes of location, as we shall see. It is important when assigning numbers, however, to keep in mind which sprites must pass in front of others on the display.

Sprite pointers

We have defined two sprites and placed them into 64-byte blocks of memory, and we know that they have relative priorities and abbreviated addresses. The way VIC finds them is by a set of prioritized pointers located in the otherwise-unused space between the top of screen memory and the next page boundary.

Screen memory usually goes from 1024 to 2023, for a total of 1000 bytes. The next page boundary, an even multiple of 256, is at 2048, which means there are 24 unused bytes above screen memory from 2024 to 2047. Rather than simply letting this memory lie fallow, VIC uses the last 8 bytes (2040 through 2047) for sprite pointers. The pointers contain the abbreviated addresses of the sprite definitions in numeric order 0–7. Hence, 2040 points to the definition for Sprite 0, 2041 to Sprite 1, etc. In the landing program, line 220 sets up the pointer for

the saucer by POKEing its definition address into 2047, and 920 loads the Sprite 0 pointer for the shelter with a POKE to 2040.

Each sprite is independent of the background and of other sprites, but it is not independent of the screen memory and the video bank. When you switch to another screen memory, you must move the sprite pointers to the corresponding bytes above the new screen memory. Also, the sprite definitions must be within the 16K bank currently in use. If you switch to a different bank without first moving the sprite images to that bank, the sprites instantly turn to random mishmashes, since the pointers will point to garbage.

The value of the sprite pointer is relative to the starting address of the bank and not to the start of memory as a whole. A sprite definition located at 12288 is in Bank 0 and the pointer to it has the value 12288 / 64 = 192. In Bank 1, a sprite definition located at address 28672 has the abbreviated pointer computed as (28672 - 16384) / 64, which is also 192. Because they occur at the same relative positions within each bank, even though at different real memory locations, their pointers are the same.

When you are working with symbolic variable screen-memory addresses ("SM" in many of this book's programs), you can refer to the sprite pointer set by offsetting 1016 from SM, as in

```
SP = SM + 1016
```

To locate the pointer to Sprite 4, refer to it as SP + 4. The saucer program uses absolute addresses instead for clarity.

Turning sprites on and off

The byte at 53269 is known as the "sprite enable register" because its bit settings govern whether a sprite is enabled (visible) or not. There is a bit for each sprite and the bit numbers correspond to the sprite numbers as follows:

```
7 6 5 4 3 2 1 0
```

If bit 4 is set to 1, Sprite 4 is turned on; if to 0, it's off. You can disable all sprites at the end of a program or during significant pauses by POKEing 0 into this location. The enabling or disabling of individual sprites, which is more common, usually takes the form of a complex POKE/PEEK. In general, it is easiest from the programming standpoint to refer to a sprite by placing its number into a variable such as SN and then using the following instructions. To turn Sprite SN *on*:

```
POKE 53269, PEEK(53269) OR (2^SN)
```

To turn Sprite SN *off*:

```
POKE 53269, PEEK(53269) AND (255 - 2^SN)
```

The saucer program doesn't use this convention, so that it is clear what the instructions are doing.

Sprite colors

The color of each sprite is established by POKEing a color code into the consecutive memory locations from 53287 through 53294. These bytes are arranged in the sprites' numeric priority order, so 53287 holds the color for Sprite 0, 53288 for Sprite 1 . . . 53294 for Sprite 7. Lines 230 and 920 illustrate this by setting the saucer to red and the shelter to blue, respectively.

The 1 bits in the sprite image translate on the screen to dots of the specified sprite color. The 0 bits are "transparent," allowing the color behind to "pass through." If you need to have more than one color in a sprite, you can use the multicolor mode discussed later in this chapter. The mode of each sprite (monochrome, as the ones in the exercise are, or multicolor) is set independently, permitting you to have both kinds of sprites active at the same time.

You can change the color of a sprite "on the fly," that is, while it is visible, by simply POKEing a new color code into that sprite's color register. The sprite instantly takes the new color without any other alteration in its form or location. For example, suppose the captain powers down his saucer once it's safely under cover by typing the instruction

```
400  POKE 53294, 11
```

Run the program again and watch the craft turn dark gray as soon as the shelter has moved into place. Once you've seen this, take the instruction out of the program by typing 400.

Sprite expansion

In addition to the horizontal expansion used with the saucer, it's also possible to expand a sprite vertically, doubling its height from 21 to 42 rows of dots. The two dimensions of expansion are independent of each other, and you can expand a sprite in both directions at the same time to create one 48 dots wide by 42 rows high.

The point of reference for a sprite is its upper-left corner at Row 0, Column 0 of the image. When you expand a sprite horizontally, expansion occurs to the right, with the left edge remaining where it is. Vertical

expansion occurs downward, so that Row 0 stays put and the bottom of the sprite moves down. Resolution does not increase with expansion. Instead, a 1 bit generates two dots either side by side or up and down, depending on the direction of expansion, or a 2 × 2 dot square if the sprite is expanded both ways.

The Commodore 64 has two memory locations in which the bit settings govern horizontal and vertical expansion of individual sprites. Just as in the enable register, the bit numbers of these bytes correspond to the sprite numbers, so that bit 0 pertains to Sprite 0, bit 1 to Sprite 1, etc. The horizontal expansion register is at 53277, the vertical at 53271. Line 310 in the landing program expands the saucer horizontally by setting bit 7 of 53277 to a 1. In practice, you'll usually want to use generalized statements that expand or unexpand the sprite identified by SN without affecting the others. Expand SN horizontally with

```
POKE 53277, PEEK(53277) OR (2^SN)
```

and unexpand it to normal width with

```
POKE 53277, PEEK(53277) and (255 - 2^SN)
```

To expand and unexpand Sprite SN vertically, use the same instructions, but substitute the address 53271.

You can see the effect of vertically expanding the shelter by adding the instruction

```
400  POKE 53271, 1
```

This sets bit 0 to a 1 after the shelter is already in place. Run it and watch what happens. Notice in particular that the roof of the shelter smashes down on top of the saucer. This is because the number of dots has doubled along the vertical axis in a downward direction. Obviously this is unacceptable to the Adenoids inside, who are relying on the shelter for protection and not expecting it to crush their craft in the process, to say nothing of the fact that it looks peculiar with the legs so long. To right the situation, remove line 400 and enter the command

```
POKE 53271, 0
```

to reset the vertical expansion register to 0 (otherwise, since the program doesn't reset it, the shelter will remain expanded for the duration of this exercise).

Sprite cloning

It is perfectly legal for two or more sprites to use the same definition even if they are of different colors and different expansions in unrelated

locations on the screen. The only thing they share is a memory image, "cloning" from the same source. As an example, suppose the Adenoids are carrying a smaller craft on board their saucer, and they need to send it home for mail and laundry. Figure 8.5 separates it from the mother ship, turns it green and sends it on its way.

```
400 REM  **  SEND OUT MESSENGER
410 SN = 5: POKE 2040 + SN, 12288/64
420 POKE 53287 + SN, 2
430 CY = PEEK(53263)
440 POKE 53249 + (SN * 2), CY
450 POKE 53248 + (SN * 2), D
460 POKE 53269, PEEK(53269) OR (2^SN)
470 FOR X = D TO (D-80) STEP -1
480 POKE 53248 + (SN * 2), X: NEXT X
490 POKE 53287 + SN, 5
500 FOR T = 1 TO 300: NEXT T
510 FOR Y = CY TO 0 STEP -3
520 POKE 53249 + (SN * 2), Y: NEXT Y
RUN
```

Figure 8.5 Dispatching the messenger

Line 410 points Sprite 5 at the saucer image and 420 makes it red, the same as its mother ship. Line 430 fetches the current Y position of the saucer (discussion of positioning is coming up soon, so don't worry if you don't understand some of these instructions). Line 440 sets Sprite 5 to the same Y, and 450 centers it horizontally on the saucer. Line 460 enables the messenger-craft sprite without affecting the others. Lines 470–480 move it away from the mother ship. The little saucer turns green at line 490 and pauses in the delay loop at line 500 before lines 510–520 blast it off into the vastness of outer space (really, it's still there, but you can't see it under the top border of the screen).

This little messenger craft is, of course, the same shape as the mother ship and has merely cloned from it, demonstrating that sprites may readily share the same image even though their activities and other characteristics differ.

Another view of spritemaking

Remember the Starship that got blown up by the Adenoids? It was an undistinguished-looking craft, simply a white rectangle. Well, unbe-

knownst to the Adenoids, its sister ship is approaching Darthia. The quickie spritemaker in Fig. 8.6 creates it.

```
530 REM  **  BUILD THE STARSHIP
540 SL = 12416: SP = SL / 64: SN = 4
550 FOR P = 0 TO 63
560 POKE SL + P, 255: NEXT P
570 POKE 2040 + SN, SP
580 POKE 53277, PEEK(53277) OR (2^SN)
590 POKE 53287 + SN, 1
```

Figure 8.6 A quick spritemaking trick

There's no point in running the program with this addition, because its effect is not visible with the Starship out of sight. Let's look instead at what it does. The heart of the matter is in the loop at lines 550–560, in which the entire sprite image is loaded with bytes consisting entirely of 1 bits. The effect is to turn on every dot in the sprite without using DATA statements as we did for the others.

The Starship will be Sprite 4 with its image at 12416; we now have definitions for Sprites 7, 0, and 4, in that order (which incidentally shows that images themselves do not need to be in any kind of sequence). Line 570 sets the pointer for Sprite 4 to the new image, 580 expands it horizontally to replicate the Starship destroyed in Chapter 6, and 590 makes it white.

The Good Guys' Starship prepares to approach.

Positioning sprites

The location of a sprite on the screen follows the now-familiar XY coordinate system, with some modifications. In HRG, the screen has 320 horizontal ("X") points numbered 0–319 and 200 vertical ("Y") points numbered 0–199 (Fig. 8.7). For sprites, there are still 320 × 200 points on the screen, except that the numbering system works a little differently. The visible area of the screen is the 320 points between X coordinates 24 and 343 and the 200 vertical points between Y coordinates 50 and 249. The HRG coordinate X = 0 is the left edge of the screen, whereas a sprite X coordinate describes the same point as X = 24. Therefore, to adjust an HRG X coordinate to the corresponding X for a sprite, add 24, and to adjust an HRG Y coordinate to the sprite counterpart, add 50. The reason for this discrepancy is to provide an area off the screen (under the border) for placing sprites so that they can slide smoothly into and out of the viewing area.

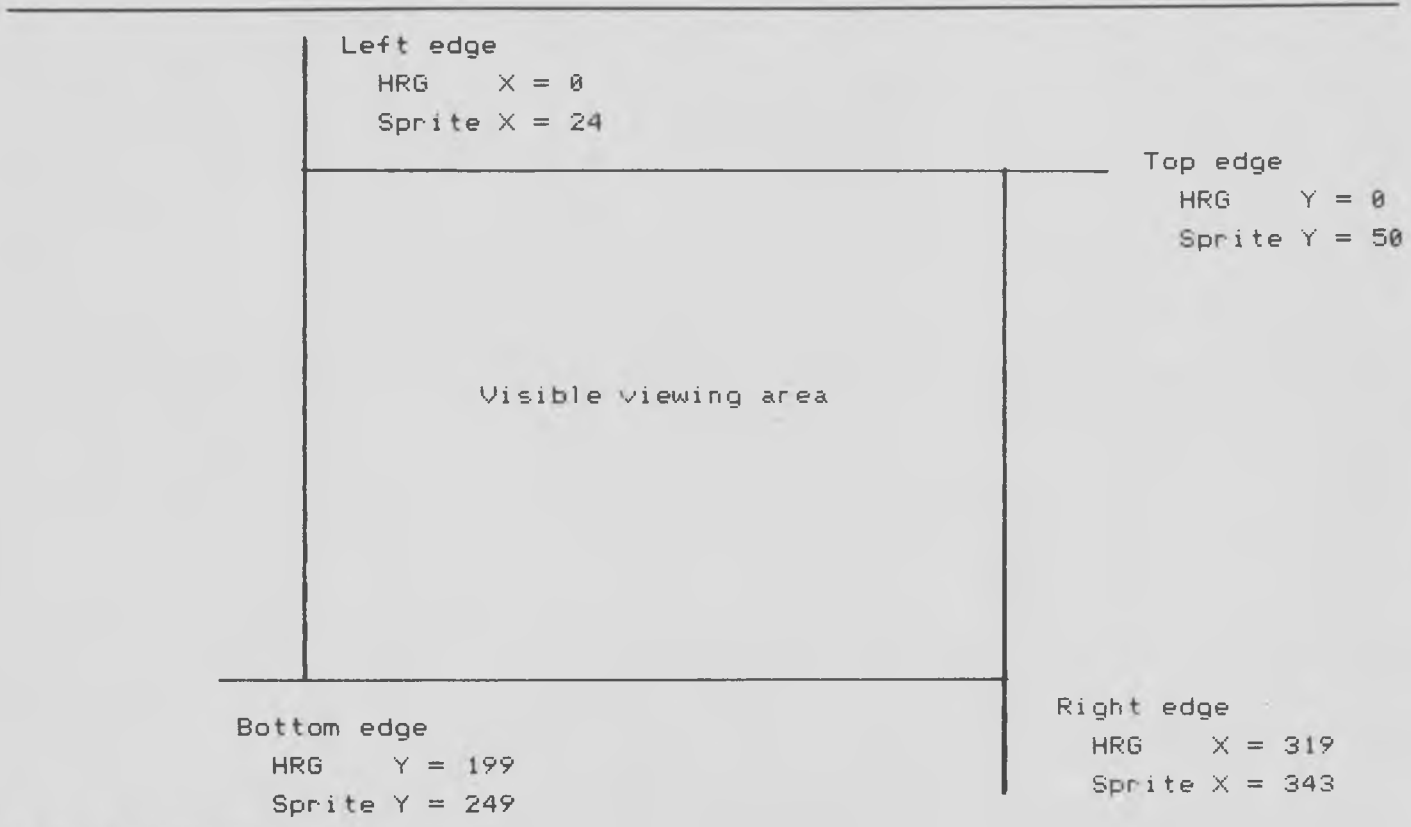


Figure 8.7 Screen XY coordinate systems

Each sprite has a pair of memory bytes ("registers") where its XY coordinates are stored for reference by the VIC-II chip. The XY coordinates refer to the position of the sprite's upper-left corner, whether a dot is visible there or not; the upper-left corner of the *image area* is the point of reference. The location registers begin at address 53248 and extend through 53263, arranged in pairs: 53248 and 53249 give the X and Y coordinates, respectively, for Sprite 0; 53250 and 53251 give the X and Y for Sprite 1, . . .; 53262 and 53263 give the X and Y for Sprite 7.

Figure 8.5 calculates the addresses of the position registers for a sprite by offsetting from the first register pair using the sprite number SN. Since it takes 2 bytes for each sprite, the sprite number has to be multiplied by 2 for the offset, thus

```
POKE 53248 + (SN * 2), X
POKE 53249 + (SN * 2), Y
```

These are generalized statements that will update the position registers for any sprite identified by SN. Similarly, line 430 in Fig. 8.5 finds the current Y of the saucer by PEEKing into the Y position register for Sprite 0, and in lines 440 and 510 it uses that position to place the new sprite and move it vertically.

There is a problem with X coordinates: if the sprite's X coordinate is greater than 255, it can't be held in one byte, and yet the X coordinates of the visible viewing area go up to 343. BASIC won't even let you try to POKE a value of 256 or greater, but instead crashes the program with an ILLEGAL QUANTITY ERROR.

As you might imagine, the Commodore folks discovered this before we did and came up with a solution. The memory byte past the last position pointer (53264) is called the "sprite X MSB register," a name that merits explanation. There is a bit position in this register for each sprite, with the bit and sprite numbers corresponding. If the bit for a sprite is turned on in this register, the X coordinate given for the sprite position is added to 256 by the VIC-II chip. Under this scheme, when X = 255, the X register holds 255 and the sprite's bit in the X MSB is 0. If X = 256, the X register holds 0 and the X MSB bit becomes 1. The "X MSB" is a ninth bit in the most significant position, which extends the range of X coordinates to 511.

This doesn't happen automatically, however. Your program must set and reset the X MSB bit and adjust the X position register according to changes in the X coordinate. This adds a level of complication that is, fortunately, easily handled by the subroutine in Fig. 8.8, which sets X's for all sprites and takes into account the X MSB bit. The subroutine needs to know the number of the involved sprite (SN) and the real X. As a fringe benefit, it also sets the Y register. With this subroutine, you can confidently place a sprite at any X on the screen without having to worry about its relationship to the 256 barrier.

```
950 REM  **  PLACE SPRITE AT X
955 X9 = X: N2 = SN * 2: S2 = 2^SN
960 IF X9 < 256 THEN POKE 53264,PEEK(53264)
    AND(255-S2):GOTO 980
970 X9 = X9 - 255: POKE 53264, PEEK(53264) OR S2
980 POKE 53248 + N2, X9
990 POKE 53249 + N2, Y: RETURN
```

Figure 8.8 Sprite horizontal positioning subroutine

There is one other problem that falls into the category of a minor annoyance, and that is the available space to the left of the screen under the border. Above, below, and to the right of the viewing area we have plenty of room to push a sprite out of sight. To the left, however, there are only 24 unused dot positions, just enough to conceal an unexpanded sprite. Perhaps you've noticed that the saucer doesn't slide out from under the left border on its descent, but pops out instead. Because it is expanded and positioned at X = 0, half of it sticks out into the viewing area. To conceal an expanded sprite under the left border, you have to place it at X = 488. Moving right, step the X up to 511 and then wrap it back to 0 to continue. Moving left, decrease X to 0, then jump it to 511 and continue decreasing until it drops below 488. The Commodore engineers weren't so clever about this problem, which seems explicable only as an oversight, since there are 166 invisible X positions to the right of the screen; they could have given us 48 concealed X's under the left border instead of 24 and saved a lot of programming complexity.

The Starship cometh

It was quite by accident that the Starship chanced upon this remote moon so soon after the Adenoids landed on it. The Good Guys might have passed it without a glance had the departing messenger craft not attracted their attention. Seeing that it resembles the saucers of the evil Adenoids, they have decided to reconnoiter in case any of the enemy are lurking down there someplace. If they spot anything suspicious they'll back up and hover over it. Their flight plan is shown in Fig. 8.9.

As you see, this flight begins at X = 350, off the right edge of the screen, passes serenely across the 256 barrier and onward to the left edge of the screen. At that point, someone on board sees the sheltered saucer on the ground and sounds an alarm, causing the Starship to back up until it is immediately over the Adenoid craft (same X coordinate). There it hovers, making a visual verification. And what's it going to do next? Attack, of course! But first. . . .

```
600 REM  **  STARSHIP FLIGHT PLAN
610 X = 350: Y = 65: GOSUB 950
620 POKE 53269, PEEK(53269) OR S2
630 FOR X = 350 TO 25 STEP -5
640 GOSUB 950: NEXT X
650 REM  **  BACK UP AND HOVER
660 CX = PEEK(53262)
670 FOR X = 25 TO CX STEP 3
680 GOSUB 950: NEXT X
RUN
```

Figure 8.9 The Starship's flight plan

Controlling the rate of motion

So far this Drama of Deep Space has used several rates of motion for sprites. In line 320 the vertical rate of descent for the landing saucer was set at 0.7 dots per unit of horizontal motion and was applied in line 340; the rate of horizontal motion began at 3 dots, but gradually decreased until reaching zero. The shelter moved at a rate of 1 dot per motion unit (loop) in line 390, as did the messenger craft in rolling away from the mother ship in line 470. When the little saucer blasted off, it rose at 3 dots per motion unit in line 510, and that was pretty fast. Then came the Starship, cruising across the screen at a rate of 5 in line 630, backing up to check out the saucer at a speed of 3 in line 670, and yet, although those were among the highest rates of motion in the program, the Starship didn't seem to move very fast.

The reason is that the subroutine at line 950, which takes care of the over-256 problem, has a lot of work to do. POKEs and PEEKs are "high-overhead operations"—meaning they take a lot of machine resources—and consequently they slow things down. Consider how performance improved when we went from BASIC to machine language in earlier chapters. POKE, PEEK, AND, and OR all have direct one-instruction counterparts in machine language. Another question to ask the Commodore designer if and when you meet him is why Commodore BASIC has such inefficient POKEs and PEEKs when the computer's features require such lavish use of them. They could have done a better job of language design.

But they didn't, so we have to deal with them as they are. Setting the rate of motion is a matter of trial and error, of fiddling with the STEP clause in a FOR loop or, as in the saucer landing, establishing a rate-of-change value that is added successively to the most recent position.

A sprite usually moves as a result of executing a loop that advances

its position in each repetition. It is axiomatic, then, that the rate of motion is inversely proportional to the amount of work done by the loop to effect movement. In other words, the more the loop has to do, the longer it takes, and the slower the sprite moves.

The commonsense means of acceleration is to increase the distance moved in each iteration of the loop, and that's where the experimentation comes in. This solution, however, has a Catch-22. The larger the increments of movement (the STEPs), the more jerky the motion of the sprite. A sprite moving at a low motion index such as 1 slides smoothly, but you'll notice that the Starship has a perceptible quiver as it proceeds grandly across the sky. It is moving 5 dots at a time. This jitter becomes even more noticeable if you change the STEP value in line 630 to −10. Some of that is attributable to its size; the smaller the sprite, the less noticeable the jumps it makes.

Don't be afraid to experiment with the units that control the rate of motion of your sprites. Monkey with those in this program to see the differences they make.

You may fire when you are ready, Gridley

Right now the Adenoids are sitting ducks even under cover, because the Starship has a new weapon capable of penetrating the shelter. Working to the Adenoids' advantage, however, is the fact that it hasn't yet been perfected. The defense contractor, who has managed to overrun costs by only 450 percent so far, is still working out the bugs. The trouble is with the aim; the operator fires at a target, but the projectiles angle away from the intended line of fire. Studies have determined that when the projectile moves at a forward rate of 10, the random spread can be expressed in BASIC as

```
H = (RND(1) * 10) - 5
```

which means that for every 10 units of forward motion, the projectile will move up to 5 units left or right, and there's no predicting what this deviation will be from one firing to the next.

In fact, the problem has been studied to such an extent—at lavish taxpayer expense, of course—that a general description of its random firing pattern for 10 projectiles (which is the number the Starship happens to have on board) can be embodied in the following BASIC statements:

```
FOR P = 1 TO 10
V = 10: X = D + 12: Y = 88
H = (RND(1) * 10) - 5
Y = Y + V: X = X + H
  (Move the projectile until it hits something)
NEXT P
```

Before pursuing this random trajectory further, let's describe the projectile itself. It's a yellow object rectangular in shape (the Good Guys' trademark) that carries a sensor. Each time it moves one motion unit, it pauses to find out whether it has hit anything, and if it has not, it moves another unit. It determines if it has scored a hit by means of a Top Secret method called "sprite collision detection," which spies will disclose in a moment. A direct hit is an unusual event with this weapon, because instead of causing an explosion, it spreads a banner reading "HOORAY FOR THE GOOD GUYS!" next to the Starship and ends the program. Figure 8.10 creates a visual image of the projectile.

```
690  REM  **  PROJECTILE IMAGE
692 SN = 6: SL = 12480: SP = SL/64
694 FOR P = 0 TO 63: POKE SL + P, 0: NEXT P
696 FOR P = 0 TO 24 STEP 3
698 POKE SL + P, 255: NEXT P
700 POKE 2040 + SN, SP: POKE 53287 + SN, 7
```

Figure 8.10 Defining the projectile

Line 694 clears the image area, and lines 696 and 698 form the projectile as a square in the upper-left corner. This is a modification of the method used to generate the Starship. Gridley now has his arsenal prepared and he is ready to fire. First, though, we need to discuss. . . .

Sprite collision detection

The eye is too slow to take it all in, but each time VIC moves a sprite to a new location, he does an astonishing number of things. Among them is deciding what needs to be done with each dot the sprite covers, since sprites have priority over background and, except for tail-end number 7, over some or all of the other sprites. When a dot in the newly arrived sprite competes with a dot in another sprite or overrides a dot of foreground color such as in a character, a "collision" has occurred. A collision does *not* occur when two or more sprites overlap in their transparent areas (0 bits), or when a transparent part of the sprite overlays a character.

VIC shares the news of a collision by setting bits in a couple of bytes. A sprite-to-sprite collision is registered in address 53278 by turning on the bits corresponding to the sprites involves in the crash. A sprite-to-data collision (hitting a background feature) sets the involved sprite's bit in 53279. You can therefore check these two bytes after a move, sampling the bits for the sprites you're interested in, and take appropriate action.

These two registers are a little unusual, because each one automatically resets to 0s whenever you PEEK at it. It's like a blown bubble floating in the air: you can touch it once, and then it's gone. For this reason, if you need to test more than one bit, you have to move the byte into a variable (see line 722 in Fig. 8.11) where you can run tests without resetting it.

In this program, the saucer is Sprite 7 and the shelter is Sprite 0, which are in contact and as a result in a constant state of collision. This means that bits 0 and 7 in 53278 are always set. You can tell if the projectile, Sprite 6, has struck the shelter or saucer by testing bit 6. If a 1, it's hit the target.

Commence firing

Given the preceding conditions, the Range Officer aboard the Starship now types the instructions in Fig. 8.11 into the computerized weapon-control system.

```
702 REM  **  COMMENCE FIRING
704 V = 10: SN = 6: POKE 53278, 0
706 FOR P = 1 TO 10
708 X = D + 12: Y = 88: GOSUB 950
710 POKE 53269, PEEK(53269) OR S2
712 H = (RND(1) * 15) - 10
714 Y = Y + V: X = X + H
716 IF Y > 255 THEN 728
718 GOSUB 950
720 REM  **  COLLISION CHECK
722 C = PEEK(53278)
724 IF (C AND S2) <> 0 THEN 730
726 GOTO 714
728 NEXT P: GOTO 799
730 REM  **  BULLSEYE!
732 PRINT CHR$(19): PRINT: PRINT
734 PRINT "HOORAY FOR THE GOOD GUYS!"
RUN
```

Figure 8.11 Commence firing

In light of all we have said about collisions, this should make sense to you, the only surprise being in line 704, which POKEs a 0 into the collision register. This clears any garbage left over from a previous run that could mislead the projectile into thinking it had struck the target.

This is not truly a game, since you have no control over its events, but it does possess the other elements of arcade video games: animation, fantasy, a story of sorts, action, an uncertain outcome (Will the Good Guys prevail over the wicked Adenoids?). The purpose here has been to demonstrate the techniques for creating and controlling sprites, and that, consistent with the intent of sprites, has been fun.

Foreground/background priority

The pecking order among sprites is fixed by their assigned numbers, as I've already discussed, so that Sprite 0 is the most dominant and Sprite 7 is at the bottom of the social order. As a general rule, all sprites regardless of their priority pass in front of the background, thus taking priority over it. This rule may, however, be overridden for each sprite, so that the sprite passes *behind* ordinary graphics and yet—depending on its priority—in front of other sprites (that might themselves have priority over screen figures).

The sprite-to-data priority is established by byte 53275, in which each sprite owns the bit corresponding to its number. When bit 0 has a 0 value, Sprite 0 passes in front of screen data; if this bit is set to 1, the sprite goes behind graphics. You can use this capability to create three-dimensional imagery. Type Fig. 8.12 to see a striking example.

```
NEW
100 REM  **  ORBIT AROUND A POLE
110 SM = 1024: CM = 55296
120 PRINT CHR$(147)
130 POKE 53281, 6: POKE 53280, 6
140 FOR Y = 0 TH 24: P = (40 * Y) + 20
150 POKE SM + P, 160: POKE CM + P, 14
160 NEXT Y
170 REM  **  BUILD SPRITE
180 SL = 12288: SP = SL / 64: P = 0
190 READ X: IF X < 0 THEN 210
200 POKE SL + P, X: P = P + 1: GOTO 190
210 FOR X = P TO 63: POKE SL + X, 0: NEXT X
220 POKE 2040, SP
230 POKE 53287, 1
240 POKE 53248, 0: POKE 53249, 0
250 POKE 53269, 1
260 REM  **  ORBIT
270 BX = 184: BY = 130: A = 0: RC = 55
280 A = A + 3: IF X > 359 THEN A = 0
```

```
290 R = A * (π / 180)
300 X = (RC * COS(R)) * 1.3 + BX
310 Y = (RC * SIN(R)) / 3 + BY
320 POKE 53248, X: POKE 53249, Y
330 IF Y < BY THEN POKE 53275, 1: GOTO 350
340 POKE 53275, 0
350 GET X$: IF X$ = "" THEN 280
360 POKE 53269, 0: POKE 53275, 0
370 PRINT CHR$(147): END
1000 DATA 7, 224, 0, 31, 248, 0, 63, 252, 0
1010 DATA 127, 254, 0, 127, 254, 0, 255, 255, 0
1020 DATA 255, 255, 0, 255, 255, 0, 255, 255, 0
1030 DATA 127, 254, 0, 127, 254, 0, 63, 252, 0
1040 DATA 31, 248, 0, 7, 224, 0, -1
RUN
```

Figure 8.12 3-D using sprite-to-data priority

This program orbits a ball around a pole, giving the impression that you're looking down from above the orbital plane. The ball passes in front of the pole, then swings around and goes behind it, then in front again, continuously orbiting until you press the SPACE bar.

Lines 260–350 use the circle equation to plot positions for the sprite through 360° (swinging 3° at a time). The circular path is compressed into an ellipse by flattening the Y axis in line 310 and extending the X axis in line 300. Lines 330 and 340 are the crux of this three-dimensional effect. They make the ball pass behind the pole when its Y coordinate is above the Y reference point (BY), and restore the normal sprite priority over data, making it pass in front of the pole, when it is on the downward leg of its orbit.

You can reverse the entire perspective of this animation by changing the priority as follows:

```
330  IF Y < BY THEN POKE 53275, 0: GOTO 350
340  POKE 53275, 1
```

The only values changed here are the bit settings in the sprite-to-background register, and yet the ball now appears to be going in the opposite direction, as though we are looking up from under its orbital plane.

One final note has to do with line 360, which not only turns off the sprite enable register, but also resets 53275. This is a "courtesy POKE" to make sure that all sprites are restored to normal priority with respect to the screen data. That way we won't have misbehaving sprites in the next program.

Automating the sprite shop

I have a complaint. The making of sprites is a tedious, exacting, boring job that's. . . . Well, precisely the kind of thing computers are supposed to relieve people of doing. I'll bet you agree with me, too. It's important to understand the structure of sprites, but now that we do, our efforts and time can be better spent on creativity than on clerical chores. Let's automate the task with the sprite-making tool in Fig. 8.13.

```
NEW
100 REM  ** SPRITE MAKER #1
110 POKE 53269, 0: PRINT CHR$(147)
120 FOR Y = 1 TO 21
130 FOR X = 0 TO 23: PRINT ".";: NEXT X
140 PRINT: NEXT Y
150 PRINT CHR$(19)
160 PRINT TAB(28) "STEP 1:"
170 PRINT TAB(28) "RUN 200 NEXT"
180 FOR Y = 3 TO 20: PRINT: NEXT Y: END
190 REM ----------
200 REM  ** STEP 2 BUILDS SPRITE
210 SL = 14336: SP = SL/64: PRINT CHR$(19)
220 PRINT TAB(28) "STEP 2:"
230 PRINT TAB(28) "K IF OK      "
240 PRINT TAB(28) "N IF NOT": PRINT TAB(28);
250 FOR P = 0 TO 63: POKE SL+P, 0: NEXT P
260 POKE 2040, SP: POKE 53287, 1
270 POKE 53248, 255: POKE 53249, 200
280 POKE 53269, 1: E = 8: B = 0
290 FOR Y = 1 TO 21
300 FOR X = 0 TO 23: P = 1024 + X + (Y * 40)
310 E = E - 1: V = PEEK(P)
320 IF V = 46 THEN 340
330 POKE SL + B, PEEK(SL + B) OR 2^E
340 IF E = 0 THEN E = 8: B = B + 1
350 NEXT X: NEXT Y
360 INPUT X$
370 IF X$ = "K" THEN 410
380 PRINT CHR$(19)
390 FOR X = 1 TO 4: PRINT TAB(28) " [12 spaces] "
400 NEXT X: GOTO 160
410 REM  ** PRINT VALUES FOR SPRITE
420 POKE 53269, 0: PRINT CHR$(147)
430 FOR Y = 0 TO 20
```

```
440 FOR X = 0 TO 23: P = X + (Y * 40)
450 PRINT PEEK(SL + P),
460 NEXT X
470 PRINT: NEXT Y
RUN
```

Figure 8.13 A tool for making sprites

If you don't have a sprite in mind, just make one up to see how this program works. It's a two-part program. The first part displays a grid of periods, each representing a dot position in the sprite. It's very important not to press RETURN or the down-cursor key while the cursor is at the bottom of the screen, since that will cause the display to scroll up and confuse the program. Use the SHIFT and cursor keys to move the cursor around the grid. You can type any character (except a period) to signify a dot in the sprite 1; I suggest an asterisk because it's a very full character that stands out. Don't press RETURN to move to a new line, because BASIC will print a SYNTAX ERROR message right in the midst of your sprite. In case that does happen, go over the letters replacing them with periods and asterisks as appropriate to undo the damage.

After you've made a sprite pattern, move the cursor down to the line immediately below the last row of the grid. The message in the upper-right corner of the screen is a reminder of what you need to type now: RUN 200.

The program breaks in two with the END in line 180, and line 200 begins a new but related program. This part, as the REMark says, builds the sprite. It does this by scanning the grid left to right and downward (lines 290–350) and setting the bit corresponding to each nonperiod character. The sprite is enabled, so as the scan progresses the sprite takes shape in the lower-right portion of the screen. You can see it exactly as you specified it in the grid.

There is a minimenu in the upper-right corner of the screen telling you your choices. When the program is finished making the sprite, it flashes the cursor under the menu. Type N (or any other character except K) if the sprite needs more work and the program will jump back into the conclusion of part 1 to let you make revisions. In that case, you'll have to type RUN 200 again after the changes are entered. When you accept the sprite, type K and the program will print a tabular list of all the values you need to key into a DATA statement to form the sprite. That's all there is to it! You can now knock out a sprite in a couple of minutes and know exactly how it looks before even setting pencil to paper.

When making expanded sprites, you first have to set up the expansion register(s) with POKE(s) typed at the keyboard. This program builds the images as Sprite 0, so to expand it horizontally type the command

POKE 53277, 1

and to expand it vertically enter

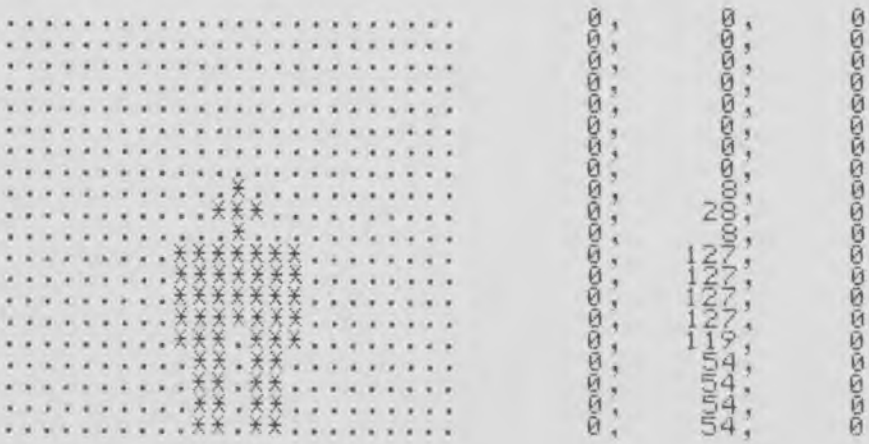
POKE 53271, 1

then run Step 1 of the program and proceed normally. The sprite will appear with the expansion(s) you specified. Don't forget to cancel the expansion afterward by POKEing 0 into the affected register(s), since the program does not clear those addresses.

Animating the animation

Some tireless fitness fiend is doing jumping jacks on the lawn I can see through the window of my display screen. He stands there briefly, then throws his arms over his head as he jumps up and comes back down on spread feet, and then he jumps again and returns to standing position with arms at his sides. I'm out of breath just watching, but he shows no sign of fatigue and he'll keep it up forever unless I touch the SPACE bar to stop him. Come, let's watch him together.

This paragon of PT works on much the same principle as the guy who waved at us from several screens in the last chapter. He consists of three images that rapidly replace one another with a delay loop freezing each frame long enough to simulate motion. It's the old movie analogy again, but on a smaller scale. Here we simply change the sprite pointer back and forth through three images (which I formed with the sprite-making tool in about 10 minutes). These images are shown in Fig. 8.14. Because



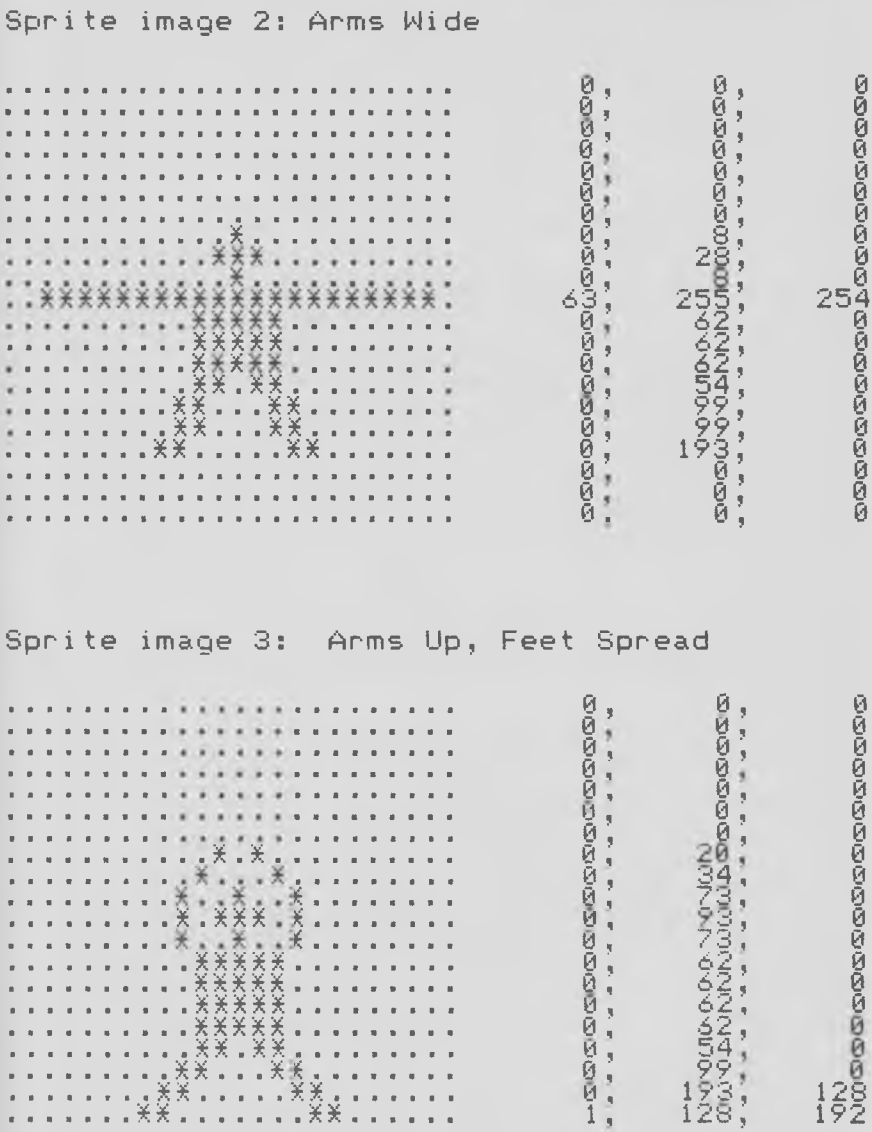


Figure 8.14 Jumping Jack's images from the tool

they are vertically expanded, the layout proportions seem a trifle strange, but they work reasonably well on the final product. Note that the "motion" comes from different vertical placements of the image within the total area of the sprite, making it unnecessary to relocate the sprite itself.

The program that makes Jack jump is listed in Fig. 8.15. Type it into

```
NEW
100 REM  ** JUMPING JACKS
110 I1 = 13824: I2 = I1 + 64: I3 = I2 + 64: DL = 250
120 PRINT CHR$(147): POKE 53281, 14
130 FOR Y = 24 TO 19 STEP -1
140 FOR X = 0 TO 39: P = X + (Y * 40)
150 POKE 1024 + P, 160: POKE 55296 + P, 5
160 NEXT X: NEXT Y
170 SL = I1: S1 = I1 / 64: GOSUB 900
180 SL = I2: S2 = I2 / 64: GOSUB 900
190 SL = I3: S3 = I3 / 64: GOSUB 900
200 POKE 53287, 0: POKE 53271,1
210 POKE 53248, 140: POKE 53249, 200
220 POKE 53269, 1
230 REM  ** 1 - 2 - 3 - 2
240 POKE 2040, S1
250 FOR T = 1 TO DL: NEXT T
260 POKE 2040, S2
270 FOR T = 1 TO (DL/4): NEXT T
280 POKE 2040, S3
290 FOR T = 1 TO DL: NEXT T
300 POKE 2040, S2
310 FOR T = 1 TO (DL/4): NEXT T
320 GET X$: IF X$ = "" THEN 230
330 REM  ** END OF EXERCISE
340 POKE 53269, 0: POKE 53271, 0
350 PRINT CHR$(147): END
900 REM  ** BUILD SPRITE IMAGES
910 FOR P = 0 TO 20: POKE SL + P, 0: NEXT P
920 READ X: IF X < 0 THEN 940
930 POKE SP + P, X: P = P + 1: GOTO 920
940 FOR X = P TO 63: POKE SL + X, 0: NEXT X
950 RETURN
1000 REM  ** STANDING TALL
1010 DATA 0, 0, 0, 0, 0, 0, 0, 8, 0, 0, 28, 0
1020 DATA 0, 8, 0, 0, 127, 0, 0, 127, 0, 0, 127, 0
1030 DATA 0, 127, 0, 0, 119, 0, 0,  54, 0, 0, 54, 0
1040 DATA 0, 54, 0, 0, 054, 0, -1
1050 REM  ** ARMS WIDE
1060 DATA 0, 8, 0, 0, 28, 0, 0, 8, 0, 63, 255, 254
1070 DATA 0, 62, 0, 0, 62, 0, 0, 62, 0, 0, 54, 0
1080 DATA 0, 99, 0, 0, 99, 0, 0, 193, 128, -1
1090 REM  ** ARMS UP, LEGS SPREAD
1100 DATA 0, 0, 0, 0, 20, 0, 0, 34, 0, 0, 73, 0
```

```
1110 DATA 0, 93, 0, 0, 73, 0, 0, 62, 0, 0, 62, 0
1120 DATA 0, 62, 0, 0, 62, 0, 0, 54, 0, 0, 99, 0
1130 DATA 0, 193, 128, 1, 128, 192, -1
RUN
```

Figure 8.15 Doing jumping jacks by computer

your computer to see how effectively it simulates strenous physical exercise.

While this is apparently a fairly long program, on closer look you find that most of it involves setting up the screen and building the three sprite images. The action itself occurs in lines 230–320, whose REM line accurately portrays what happens: the sprites are displayed in the order 1–2–3–2 and then the keyboard is scanned for a halt signal. If there is none, the sequence repeats. Note the use of delay loops to hold the two on-ground positions and, for one-fourth that interval, the jump. You can change the cadence by replacing the value for DL in line 110, making this indefatigable exercise nut become either frantic or apathetic; the higher the value, the slower he goes.

This is animation and motion achieved without moving the sprite itself, but rather simply by changing the pointer. We can, of course, change the pointer *and* move the sprite to enhance the animation, as, very soon, we shall discover.

But first let's discuss the simultaneous control of several sprites by giving our lonely friend some company. Add Fig. 8.16 to the program.

```
195 POKE 53288, 0: POKE 53289, 0: POKE 53290, 0
200 POKE 53287, 0: POKE 53271, 15
211 POKE 53250, 165: POKE 53251, 190
212 POKE 53252, 190: POKE 53253, 180
213 POKE 53254, 220: POKE 53255, 170
220 POKE 53269, 15
240 FOR P = 0 TO 3: POKE 2040 + P, S1: NEXT P
260 FOR P = 0 TO 3: POKE 2040 + P, S2: NEXT P
280 FOR P = 0 TO 3: POKE 2040 + P, S3: NEXT P
300 FOR P = 0 TO 3: POKE 2040 + P, S2: NEXT P
RUN
```

Figure 8.16 A group of fitness freaks

These instructions activate the sprites (line 220) in vertical expansion (last part of 200) and then, instead of repointing Sprite 0 only, repoint all four sprites (lines 240–300) to the same image concurrently. If you look very carefully you can see that the last man in the row acts a split-

second behind the first, a result of the sequential updating of the pointers by the loop. This is not a problem as long as the loop doesn't have a great many instructions in it; you probably didn't even notice until it was pointed out.

More spritely physical fitness

Having jumped his fill, Jack is now going for a run. This is a similar cinematic approach to that for the jumping jacks, with the exception that the sprite physically changes location as its image switches, simulating a cartoon figure. Type Fig. 8.17 and then watch carefully; Jack sprints rapidly across the screen from left to right, and it doesn't take him very long.

```
NEW
100 REM  **  SPRINTER
110 PRINT CHR$(147); POKE 53281, 14
120 L = 24: MI = 12: DL = 40
130 S1 = 12288; S2 = S1 + 64
140 SL = S1: I1 = S1 / 64: GOSUB 900
150 SL = S2: I2 = S2 / 64: GOSUB 900
160 REM  **  SET UP SPRITE
170 POKE 53287, 0
180 POKE 53277, 1: POKE 53271, 1
190 POKE 53248, L: POKE 53249, 125
200 POKE 53269, 1
210 REM  **  RUN
220 POKE 2040, I1
230 FOR T = 1 TO DL: NEXT T
240 GOSUB 500
250 IF L > 344 THEN 300
260 POKE 2040, I2
270 FOR T = 1 TO DL: NEXT T
280 GOSUB 500
290 IF L < 345 THEN 210
300 POKE 53269, 0: POKE 53264, 0
310 POKE 53277, 0: POKE 53271, 0
320 END
500  REM  **  ADVANCE SPRITE
510 L = L + MI: X = L: F = 0
520 IF X > 255 THEN X = X - 256: F = 1
530 POKE 53248, X: POKE 53264, F
```

```
540 RETURN
900 REM  **  LOAD SPRITE IMAGES
910 FOR P = 0 TO 20: POKE SL + P, 0: NEXT P
920 READ X: IF X < 0 THEN 940
930 POKE SL + P, X: P = P + 1: GOTO 920
940 FOR X = P TO 64: POKE SL + X, 0: NEXT X
950 RETURN
1000 REM  **  IN THE AIR
1010 DATA 0, 6, 0, 0, 6, 0, 0, 12, 0, 0, 28, 0
1020 DATA 0, 60, 0, 0, 127, 128, 0, 28, 0
1030 DATA 0, 31, 192, 0, 16, 64, 3, 240, 64
1040 DATA 2, 0, 64, 0, 0, 96, -1
1050 REM  **  A FOOT ON THE GROUND
1060 DATA 0, 6, 0, 0, 6, 0,  0, 12, 0, 0, 28, 0
1070 DATA 0, 28, 0, 0, 63, 0, 0, 28, 0, 0, 28, 0
1080 DATA 0, 12, 0, 0, 10, 0, 0, 14, 0, 0, 24, 0
1090 DATA 0, 56, 0, 0, 12, 0, -1
RUN
```

Figure 8.17 The sprinting sprite

Note that this time there are only two frames that alternate. You can view them by typing the following commands:

```
POKE 53248, 255: POKE 53249, 125
POKE 53277, 3: POKE 53271, 3
POKE 2040, I1: POKE 53269, 1
```

A figure appears at the right center of the screen, motionless but in a running attitude. The commands have, respectively, positioned it, expanded it, and turned it on. To view the other figure, type

```
POKE 53250, 255 - MI: POKE 53251, 125
POKE 2041, I2: POKE 53269, 3
```

The figure now appears immediately behind the first. It is more in a walking position, with one foot on the ground. These two shapes comprise the action frames, representing the two phases of running (one foot on the ground, and both feet in the air). The spacing between them here is the distance that actually separates their brief appearances on the screen, as given by the value MI (for motion index).

When moving animated sprites, it's important to establish an appropriate motion index when the sprite is "propelled by its own efforts," as is the running man. To some extent you can anticipate the motion index (the distance traveled by the legs, plus a little hop, in this case), but like the delay factor, it's largely a matter of trial and error until you find one

that looks real. There are no rules and experience only helps a little. Making animated sprites move realistically is not as easy as it looks, but this example should provide some hints that you can apply on your own.

Multicolor sprites

Up to now we have confined our examples to monochrome (single-color) sprites, in which 1 bits in the image turn on dots of the sprite color and 0 bits let the backdrop show through. This permits only silhouettes to move against the scenery which, though adding interest and action, lacks an element of realism.

We can add features to sprites by the use of the multicolor mode, which allows each individual sprite to have up to three colors in addition to transparency. Multicolor sprites have all the same characteristics as monochrome—expansion in both directions, collision detection, foreground/background priority, etc. Each sprite can be set to this mode independently of the others, so that some can be multicolor and some monochrome. There are, however, some differences between the two types, and some limitations on color selection.

The default mode of all sprites is monochrome. To place a sprite in multicolor mode, set its bit in 53276 to a 1. The bit structure works the same as in the enable register: to place Sprite SN into multicolor mode, use the instruction

```
POKE 53276, PEEK(53276) OR 2^SN
```

(where SN is the sprite number 0–7). To disable multicolor mode for Sprite SN,

```
POKE 53276, PEEK(53276) AND (255 - 2^SN)
```

These instructions turn the bit for Sprite SN on and off, respectively. To set all sprites to monochrome, POKE a 0 into 53276; to set all to multicolor, POKE the value 255.

When a sprite is in multicolor mode, it acts very much like multicolor high-resolution graphics in that, instead of consisting of individual dots, it uses pixels, each of which is two dots wide by one dot high. Each pixel is represented in the sprite image by a 2-bit value (00, 01, 10 or 11, corresponding to 0, 1, 2, and 3), which is shown in Fig. 8.18. A 0 pixel value is transparent, while the other three values refer to locations that hold color numbers. As with MHRG, a multicolor sprite has only half the horizontal resolution of a monochrome (12 pixels versus 24 dots), but the same physical width on the display. Multicolor mode sacrifices resolution for greater color capability.

Assume the colors for this sprite are:

1	Red
2	White
3	Blue

A byte in the sprite image has the pattern 11011000:

Pixel:	11	01	10	00
Value:	3	1	2	0
Color:	Blue	Red	White	(Transparent)

Figure 8.18 Color selection in a multicolor sprite

The pixel color numbers point to three memory locations that hold the color POKE codes from Fig. 3.9. For example, when the first pixel in the byte in Fig. 8.18 is decoded and found to be 11, or color selection 3, the VIC-II chip looks in the color 3 location and finds that it holds the value 6, corresponding to blue. It then displays this pixel in blue and goes to the next, which points it to the color 1 location where there is a 2 corresponding to red, etc. When the pixel contains 00, VIC lets the backdrop shine through the sprite.

These "color locations" are two special color registers for multicolor sprites, plus the same sprite color registers (in addresses 53287 through 53294) used by monochrome sprites. The correlation of pixel values to registers is shown in Fig. 8.19.

Pixel bits	Value	Register address
01	1	53285
10	2	Sprite color register (53287–53294)
11	3	53286

Figure 8.19 Pixel values mapped to color registers

The multicolor registers at 53285 and 53286 hold one color number apiece that applies to all multicolor sprites, whereas the sprite color registers at 53287–53294 holds colors that pertain to individual sprites, depending on the sprite number. This means that if you have several multicolor sprites, all of them will have the same color in pixels containing 01 and 11, but that the color can vary from one sprite to another for pixels containing 10.

One further limitation on multicolor sprites has to do with collision detection. For purposes of evaluating whether a sprite has collided with another sprite or with a background object, the VIC-II chip regards 01 as a transparent pixel. Therefore if a pixel containing 01 occupies

the same screen location as a pixel or dot from another sprite, or the same spot as a data object, a collision is *not* registered by the VIC-II. (Another question for the Commodore design department. . . .) The upshot is that, if collision detection is important, you should outline all multicolor sprites in pixels of values 10 or 11 (2 or 3) and use 01 only inside the sprite.

As a demonstration, we'll construct a rather simple sprite in which the top seven rows are all of color 1 (byte 0101 0101, a decimal value of 85), the middle seven are of color 2 (1010 1010, decimal 170), and the bottom seven of color 3 (11111111, or 255). Initially, this sprite will *not* appear in multicolor mode, but instead as a monochrome figure. After it's on the screen, we'll change it to multicolor by a keyboard command. Enter the program in Fig. 8.20.

This program constructs a red square against a black background at the right center of the screen. The bottom third of the sprite is strong red, and the upper two-thirds are hazy because of the alternating dot-on/dot-off pattern. It takes its color, being a monochrome sprite, from the value 2 POKEd into 53287 by line 220.

Type the command

```
POKE 53276, 1
```

to activate the sprite in multicolor mode. The bright band now moves to the center of the sprite, which calls for color 10 in 53287, and the upper

```
NEW
100 REM  **  MULTICOLOR SPRITE BEFORE COLOR ADDED
110 PRINT CHR$(147)
120 POKE 53280, 0
130 SL = 12288: SP = SN / 64: L = 0
140 FOR Y = 0 TO 2: READ V
150 FOR X = 0 TO 20
160 POKE SL + L, V: L = L + 1
170 NEXT X: NEXT Y
180 REM  **  SET UP SPRITE
190 POKE 53285, 0: POKE 53286, 0
200 POKE 2040, SP
210 POKE 53248, 255: POKE 53249, 115
220 POKE 53287, 2
230 POKE 53269, 1
240 END
250 DATA 85, 170, 255
RUN
```

Figure 8.20 Multicolor sprite demo

and lower thirds extinguish because there are 0s in the multicolor registers at 53285 and 53286. Now type

POKE 53285, 1

and watch the top of the sprite appear in white, which we just placed into color register #1. Enter the command

POKE 53286, 6

and the lower third becomes blue in a blaze of patriotism. If you aren't convinced this is a sprite, type the following:

FOR X=1 TO 100: POKE 53248, PEEK(53248) - 1: NEXT X

The sprite slides smoothly halfway across the screen by successively decreasing its X coordinate, just as a program would move it. The only difference is multiple colors. Play with this sprite to see the effects of changing color values in 53285–53287 with POKEs from the keyboard.

Making multicolor sprites

It's harder to make multicolor sprites because you have to fiddle with pixels having any of four values, and translate the resulting bit patterns into decimal numbers for DATA statements. This is, like the crafting of monochrome sprites, a tedious job better done by computer. Luckily, we can easily modify the old sprite-making program to create a multicolor spritemaker. It involves adding some new lines and changing others. Reload the spritemaker and enter the modifications in Fig. 8.21.

```
10 A$ = "MULTICOLOR SPRITE MAKER"
20 PRINT CHR$(147): PRINT A$
30 PRINT: PRINT
40 INPUT "NUMBER FOR COLOR 1"; C1
50 INPUT "NUMBER FOR COLOR 2"; C2
60 INPUT "NUMBER FOR COLOR 3"; C3
70 POKE 53285, C1: POKE 53286, C3
80 POKE 53287, C2
140 FOR X = 0 TO 23 STEP 2: PRINT " .";: NEXT X
255 POKE 53276, 1
280 POKE 53269, 1: E = 256: B = 0
300 FOR X = 0 TO 23 STEP 2: P = 1024 + X +(Y * 40)
310 E = E / 4: V = PEEK(P)
325 V = V AND 3
330 POKE SL + B, PEEK(SL + B) OR (E * V)
340 IF E = 1 THEN E = 256: B = B + 1
```

Figure 8.21 Modifications for multicolor spritemaker

The multicolor spritemaker is listed in its entirety in Appendix D. The new lines capture and set up the multicolor aspects. The most significant change is in the value of E, which becomes a multiplier that shifts the pixel color number V into the bit positions corresponding to its place in the image byte.

This program works, as you might expect, a little differently. It's still a two-phase job, but the first phase is now preceded by a dialog in which the computer asks for each of the three foreground color numbers. The grid then appears, consisting of dot-space-dot-space lines. Use the cursor keys as before to move around the grid. On each period, place the number (1, 2, or 3) for the color you want that pixel to have. You can also key 0s for transparent pixels, but the program treats periods as 0s.

When you have constructed the sprite image, move the cursor below the grid and type RUN 200. The multicolor sprite will appear at the lower-right corner of the screen. You can then revert to step 1 for corrections by typing N, or get a listing of the decimal numbers that describe the sprite by typing K.

As with the first spritemaker, if you want an expanded sprite you have to POKE a 1 into the expansion registers before running step 1—and remember to reset them afterward.

Figure 8.22 shows the screen layout for a horizontally expanded truck that will soon drive across the screen. Because of the numeric designations and their wider spacing, the layout of a multicolor sprite is less apparent at a glance than using asterisks in monochrome. The interactive sprite design provided by the multicolor sprite-making program is therefore a great time- and grief-saver.

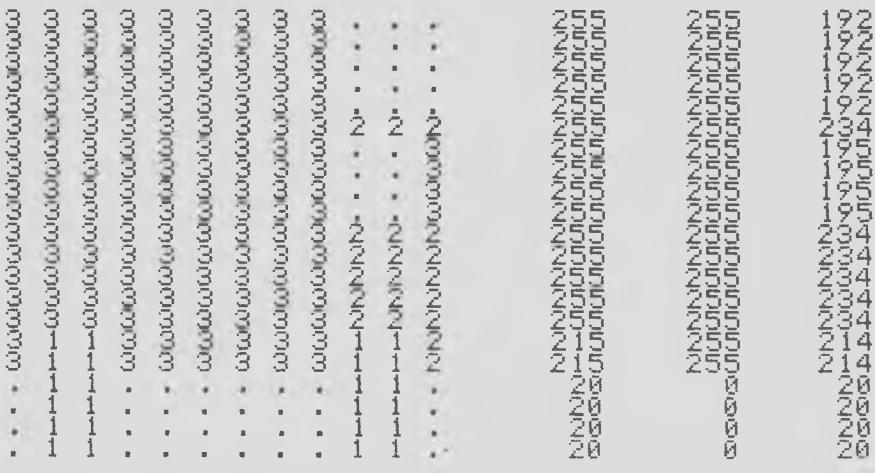


Figure 8.22 Multicolor truck

Breaker one-nine

At this moment there are countless trucks rolling along Interstate 80. I propose that we go out in spritely fashion and set up a camera somewhere on the Great Plains to watch them pass.

As everyone knows, it's simple courtesy—if not the law—for slower moving traffic to stay in the right lane and let the faster vehicles have the left lane. It so happens that, along this stretch of road, the faster trucks move at twice the speed of the slower. Also, all the faster trucks have green cabs and the slower ones red. Just coincidence, of course, though the values in the sprite color registers might have something to do with it. Type Fig. 8.23 and watch the traffic flow.

```
NEW
100 REM  **  KEEP ON TRUCKIN'
110 SL = 14336: SP = SL / 64
120 REM  **  SKY, EARTH, AND HIGHWAY
130 PRINT CHR$(147): POKE 53281, 14
140 CH = 160: CN = 13
150 FOR Y = 24 TO 21 STEP -1
160 FOR X = 0 TO 39: GOSUB 600
170 NEXT X: NEXT Y: CN = 1
180 FOR X = 0 TO 39: GOSUB 600: NEXT X
190 Y = Y - 1: CH = 228
200 FOR X = 0 TO 39: GOSUB 600: NEXT X
210 REM  **  BUILD SPRITE
220 GOSUB 700
230 REM  **  SET UP SPRITES 0 AND 7
240 POKE 53276, 129
250 POKE 53285, 0: POKE 53286, 15
260 POKE 53287, 10: POKE 53294, 5
270 POKE 53277, 255
280 POKE 53248, 232: POKE 53249, 192
290 POKE 53262, 232: POKE 53263, 184
300 POKE 53264, 129: POKE 53269, 129
310 POKE 2040, SP: POKE 2047, SP
320 P0 = 488: P7 = 488
330 REM  **  ROLL 'EM OUT
340 P = P0: SN = 0: GOSUB 800: P0 = P
350 P = P7: SN = 0: GOSUB 800: P0 = P
360 P = P7: SN = 0: GOSUB 800: P0 = P
370 GET X$: IF X$ = "" THEN 330
```

(continued)

```
380 REM  **  THAT'S ALL, FOLKS
390 POKE 53269, 0: POKE 53277, 0
400 POKE 53276, 0: PRINT CHR$(147)
410 END
420 REM  ----------
600 REM  **  POKE SCREEN/COLOR MEMORIES
610 P = X + (Y * 40)
620 POKE 1024 + P, CH: POKE 55296 + P, CN
630 RETURN
640 REM  ----------
700 REM  **  BUILD SPRITE FROM DATA
710 FOR P = 0 TO 62: READ X
720 POKE SL + P, X: NEXT P
730 RETURN
740 REM  ----------
800 REM  **  ADVANCE SPRITE
810 P = P + 2: PV = P
820 IF P < 512 THEN 850
830 P = 0: PV = P
840 POKE 53264, PEEK(53264) AND (255-2^SN): GOTO 880
850 IF P <> 256 THEN 870
860 PV = 0: POKE 53264, PEEK(53264) OR 2^SN
870 IF P > 255 THEN PV = P - 256
880 POKE 53248 + (SN * 2), PV
890 RETURN
900 REM  **  CAB-OVER TRUCK IMAGE
910 DATA 255, 255, 192, 255, 255, 192, 255, 255, 192
920 DATA 255, 255, 192, 255, 255, 192, 255, 255, 234
930 DATA 255, 255, 195, 255, 255, 195, 255, 255, 195
940 DATA 255, 255, 195, 255, 255, 234, 255, 255, 234
950 DATA 255, 255, 234, 255, 255, 234, 255, 255, 234
960 DATA 255, 255, 214, 255, 255, 214, 20, 0, 20
970 DATA 20, 0, 20, 20, 0, 20, 20, 0, 20
RUN
```

Figure 8.23 Truck traffic on Interstate 80

It is not a misprint that line 360 is a repeat of 350. The repetition is necessary for smoothness in making the truck in the fast lane (Sprite 7) go twice the speed of the other.

Notice that both trucks are of the same make and model (use the same image). That one is green and the other red is due to the different values POKEd into the sprite color registers for pixel color 2 in line 260. The

truck in the nearer lane is Sprite 0, the faster one Sprite 7, which accounts for the green truck passing on the far side. Note, too, that parts of the overtaking green truck show through the cab windows of the slower one as it roars past.

This, our final exercise in the sprite-making arts, by no means exploits the full potential of multicolor sprites, but it should provide plenty of ideas for introducing action and for creating and controlling lively, interesting displays.

Quick reference for spritemakers

It's no accident that it took seven chapters to lay enough groundwork to broach the subject of sprites. They are the apex of video features on the Commodore 64 and, as such, also the most complex. There are a lot of control locations to remember. We've discussed them all in this chapter and now, as a concise reference, we'll summarize them in the chart in Fig. 8.24, where you can quickly look them up as you create your own sprites.

DEFINITION	63 bytes (3 wide by 21 high) located at an address that is an even multiple of 64 (64th byte is unused). Must be in the same bank as screen.
POINTERS	Located in last 8 bytes of unused space above screen memory (when screen memory is at 1024–2023, pointers are at 2040–2047) in this order: Sprite 0, Sprite 1 . . . Sprite 7. Pointer value is image address / 64.
ENABLE	Address 53269. Each sprite owns a bit in the order: 7, 6, 5 . . . 0. Bit set to 1 enables sprite and reset to 0 disables it.
SPRITE COLOR	Addresses 53287–53294 in the order: Sprite 0, Sprite 1 . . . Sprite 7. The color of the sprite is POKEd to its color register. In multicolor mode, these registers hold the color number for pixel value 10.
MULTICOLOR ENABLE	Address 53276. Each sprite owns a bit in the order: 7, 6, 5 . . . 0. Bit set to 1 enables multicolor in the corresponding sprite, reset to 0 places sprite in monochrome mode. In multicolor mode, pixel value: 00 = transparent; 01 = color in 53285; 10 = color in sprite color register (53287–53294); 11 = color in 53286.

(continued)

EXPANSION

Expansion doubles the dimension of the sprite along either the X or the Y axis (or both), but does not increase resolution.

Horizontal expansion—address 53277.
Vertical expansion—address 53271.

In both of these registers, each sprite owns a bit in the order: 7, 6, 5 . . . Ø. Bit set to 1 expands the sprite, reset to Ø makes it normal size along the affected axis.

SPRITE POSITION

The position of a sprite is determined by the X and Y coordinates of the upper-left corner of its image frame (leftmost bit of byte Ø). For sprites the extreme visible coordinates are:

Left X = 24 Right X = 343
Top Y = 50 Bottom Y = 249

The sprite coordinates are contained in pairs of registers as follows:

X—53248, 53250 . . . 53262 in order
Sprite Ø, 1 . . . 7.
Y—53249, 53251 . . . 53263 in order
Sprite Ø, 1 . . . 7.

When the X coordinate is greater than 255, you must subtract 256 from it and POKE the result into the appropriate X register, and then set the bit owned by the sprite in the "X MSB" register at 53264 to 1. When the X coordinate exceeds 511 or drops below 256, reset this bit to Ø. Each sprite owns a bit in the order: Sprite Ø, Sprite 1 . . . Sprite 7 at 53264.

PRIORITY

The priority of a sprite with respect to the others is implicit in its sprite number. The lower the sprite number, the higher its priority, so that Sprite Ø passes in front of all other sprites and Sprite 7 passes behind all others.

SPRITE–BACKGROUND PRIORITY

Normally sprites have higher priority than static graphics and character data on the screen. This can be changed on an individual basis to make the sprite pass behind background objects. Each sprite owns a bit in the register at address 53275, in the order: Sprite 7, Sprite 6 . . . Sprite Ø. Bit set to 1 makes the sprite lower priority than data, reset to Ø makes it higher priority (default is Ø).

SPRITE–SPRITE COLLISION

Two sprites collide when nontransparent pixels occupy the same spot on the screen. (*Note:* In multicolor mode, a pixel with color value Ø1 is considered transparent for purposes of collision detection.) A collision sets bits corresponding to the involved sprites in the byte at address 53278. Each sprite owns a bit in this register in the order: Sprite 7, Sprite 6 . . . Sprite Ø. You can tell which sprites collided by sensing which bits

	are on (if no collision, none are on). This byte automatically resets to all 0s when read by a PEEK.
SPRITE–BACKGROUND COLLISION	A sprite collides with a background object (any nonsprite graphic or character) when a nontransparent sprite pixel occupies the same spot as an object pixel. (*Note:* In multicolor mode, a sprite pixel with a color value of 01 is considered transparent for purposes of collision detection.) A collision sets a bit in the sprite-to-background collision register at 53279. Each sprite owns a bit in this register, in the order: Sprite 7, Sprite 6 . . . Sprite 0. When set to 1, the sprite has collided with a background object, and when reset to 0, has not collided with background since last tested. This register automatically resets to all 0s when read by a PEEK.

Figure 8.24 Summary of sprite control locations

Twilleayer

CHAPTER 9
THE JOY OF STIX

Poke your head into any video arcade and you'll immediately see that, unlike our programs up to this point, games give the player control over events on the screen. So do home-computer games such as those you can buy for your Commodore 64. The concept of external control is central to all video games and, in fact, to all interactive applications of computers.

The most common device for controlling the movements of graphics such as sprites is the joystick, a gadget resembling the aircraft control mechanism from which it gets its name. With a joystick, you move the handle up, down, right, left, or at an angle to make a figure travel in a corresponding direction on the screen. A computer joystick also has a "fire" button which, when pressed, triggers some action other than travel.

The Commodore 64 has two joystick ports marked "Control Port 1" and "Control Port 2" located on the right end next to the power switch. Turn the machine off and plug your joystock into Port 2 for the experiments that follow. (You should always power off the computer to attach and disconnect peripheral devices.)

A joystick generates a digital signal in response to a mechanical action. This signal is a number indicating the direction in which the stick is moved. It has four basic directions called North, South, West, and East, which have the values 1, 2, 4, and 8, respectively. When the joystick is moved north, it sends a 1 to the computer, when west, a 4. If you move the stick at an angle—say northwest—the joystick generates the sum of the angle's two base directions. Since northwest is the composite of North and West, with values of 1 and 4, the joystick generates 5. The directions and their corresponding values are shown in Fig. 9.1.

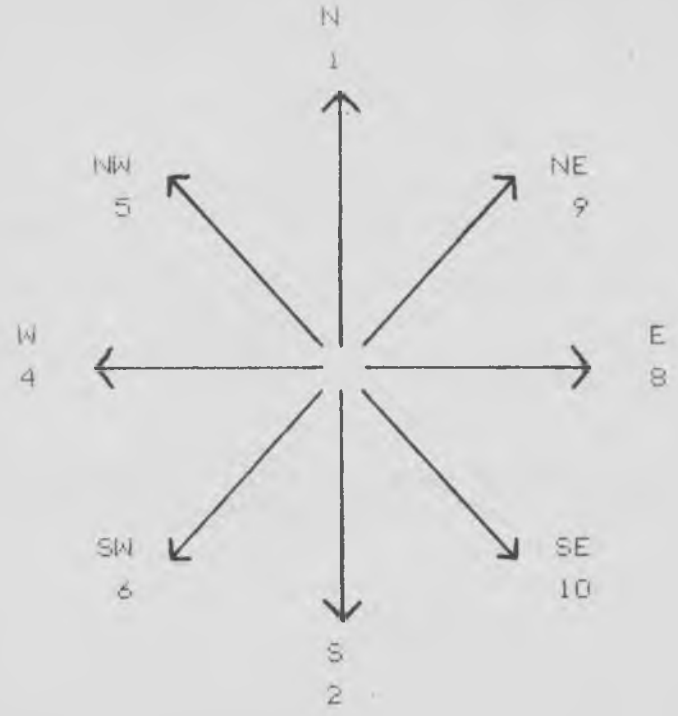


Figure 9.1 Joystick direction values

The fire button has a value of 16; that is, when pressed it generates that number, which your program can detect and use as an action trigger for shooting at the Adenoids or starting the game or whatever. The significance of the fire button is up to the programmer, and it can change at different phases of the program. Fig. 9.2, shown later, demonstrates one use of this programmable button.

Control Ports 1 and 2 are monitored in the Commodore 64 by two memory addresses located at 56321 and 56320, respectively. The lower five bits of these registers contain status information about the joystick attached to the port. (The upper three bits are used for entirely unrelated purposes that are of no concern here.) Of these five bits, the lower four hold the joystick direction number and bit 4 (place value 16) indicates whether the fire button is pressed.

These registers have the peculiar characteristic that they contain the complements of the values represented. That means that if the stick hasn't been moved and the fire button hasn't been pushed, the bits are all 1s, *not* 0s as you might expect. If the stick has been moved northwest, giving a value of 5, the 1 bits corresponding to binary 5 (0101) are 0s

and the 0s are 1s (nibble holds 1010). To convert the direction nibble back to its "real" value, then, subtract it from 15. The complement of binary 5 is 1010, a binary value of 10, and 15 − 10 = 5. The full BASIC statement that finds the direction value is

```
DV = 15 - (PEEK(56320) AND 15)
```

(Naturally, you use 56321 when checking the joystick on Port 1.) If DV = 0 after this statement is executed, the joystick is not currently indicating a direction.

Bit 4 in the port register shows the status of the fire switch. It, like its cousins in the lower nibble, is a 1 when no action has occurred and becomes 0 when somebody pushes the button. The BASIC instruction to sense the fire-button status is

```
FB = 16 - (PEEK(56320) AND 16)
```

Following this statement, FB will be 0 if the button has not been pushed and 16 if it has. You can then act on a nonzero result with a statement such as

```
IF FB <> 0 THEN (trigger the action indicated)
```

paraphrased as "if the fire button has been pushed, then. . . ."

Virtually without exception, the joystick controls a sprite. Ordinarily this happens by repetitions of a loop that samples the joystick and moves the sprite in response. Exits from the loop occur only under carefully defined circumstances, such as winning or losing the game. It's important to sample the status of the joystick frequently, because the register monitors the port on a "real-time" basis; the bits change as the stick is moved, and a delay in sampling can cause a signal to go unnoticed.

Let's put some of this discussion into practice with the simple exercise in Fig. 9.2. This program places a square sprite in the center of the screen and moves it in response to the joystick. The program ends when you press the fire switch.

The sampling loop occurs between lines 170 and 220, which read the port status byte and check it for signals. If the fire button has been pushed, line 190 triggers the end of the program. On discovering a 0 direction value, line 220 repeats the loop; otherwise (if the joystick is indicating a move), line 240 jumps to the instruction that makes the appropriate changes in the sprite's X and Y coordinates. Line 330 then calls the sprite positioning subroutine and the loop repeats.

Although Fig. 9.2 doesn't accomplish anything particularly useful, it does show all the principles of joystick programming. There are other ways of acting on joystick signals. The game that follows evaluates the direction number with multiple IF statements, showing an alternative approach.

```
NEW
100 REM  **  JOYSTICK DEMO
110 PRINT CHR$(147): POKE 53281, 0
120 SL = 12288: SP = SL / 64
130 FOR P = 0 TO 62: POKE SL + P, 255: NEXT P
140 POKE 2040, SP: POKE 53287, 1
150 X = 172: Y = 139: GOSUB 500
160 POKE 53269, 1
170 REM  **  JOYSTICK SAMPLING LOOP
180 JS = PEEK(56320)
190 FB = 16 - (JS AND 16)
200 IF FB <> 0 THEN 400
210 DV = 15 - (JS AND 15)
220 IF DV = 0 THEN 170
230 REM  **  MOVE SPRITE
240 ON DV GOTO 250,260,170,270,280,290,170,300,310,320
250 Y = Y - 1: GOTO 330                    :REM  N
260 Y = Y + 1: GOTO 330                    :REM  S
270 X = X - 1: GOTO 330                    :REM  W
280 Y = Y - 1: X = X - 1: GOTO 330         :REM  NW
290 Y = Y + 1: X = X - 1: GOTO 330         :REM  SW
300 X = X + 1: GOTO 330                    :REM  E
310 X = X + 1: Y = Y - 1: GOTO 330         :REM  NE
320 X = X + 1: Y = Y + 1                   :REM  SE
330 GOSUB 500
340 GOTO 170
400 REM  **  STOP THE PROGRAM
410 POKE 53269, 0
420 END
500 REM  **  MOVE SPRITE TO XY
510 Z = X: POKE 53249, Y
520 POKE 53264, Z/256
530 IF X > 255 THEN Z = Z - 256
540 POKE 53248, Z
550 RETURN
RUN
```

Figure 9.2 Moving a sprite with the joystick

Ace Spinvader to the rescue: a video game

To: Ace Spinvader
From: The Commodore
Subject: Rescue Mission

One of our Starships patrolling in the vicinity of Darthia has sustained damage in an encounter with the Adenoids. The most severe damage is to the guidance system, which has resulted in loss of control and a tendency for the Starship to jump erratically. The only way to rescue this Starship is to link up with its docking bay using a craft with a strong guidance system that will stabilize the damaged ship. This is a hazardous operation that requires the utmost skill and valor, which is why you have been selected for the mission. It is likely, due to the random jerking of the Starship, that it will collide with your spacecraft or jump away just as you are about to dock. Although a collision will cause you to be thrown about, the rescue craft has been fitted with protective equipment to prevent damage, thereby freeing you to concentrate on firmly securing the top of your craft into the bay on the Starship's underside. You are ordered to proceed at once to the vicinity of Darthia, there to locate the damaged Starship, connect with it, and effect repairs. Good luck.

Well, there you have it, Ace. It's not much to go on, but the Good Guys are relying on you, so en route you'll have to devise your strategy. I'll gladly pitch in and help with that, but it's up to you to maneuver the rescue craft's conical upper tip snugly into the corresponding opening in the Starship.

The first step is to define the context in which the operation occurs, which the statements in Fig. 9.3 summarize. Line 110 gives the image location and sprite number for the Starship; line 120, for the rescue craft. The variable JF in line 130 sets a "jump frequency" value for the Starship; every JF (50, in this case) passes through the maneuvering loop, and the Starship jumps to a new position. JC is the counter for this.

```
NEW
100 REM  **  ACE SPINVADER TO THE RESCUE
110 L1 = 12288: I1 = L1/64: S1 = 0
120 L2 = L1+64: I2 = L2/64: S2 = 1
130 JF = 50: JC = 0
```

Figure 9.3 Rescue program heading

In addition to the general context, we need to develop the craft themselves and the view you'll have of them with your maneuvering radar. This phase of the setup is shown in Fig. 9.4, where lines 170 and 180 place the Starship at the center of the screen, and lines 190 and 200 start your rescue craft at the upper-left corner. The variables X1, Y1 are the coordinates of the Starship, and X2, Y2 give the position of the rescue craft.

```
140 REM  **  SET UP SCREEN, SPRITES
150 PRINT CHR$(147): POKE 53269, 0
160 POKE 53280, 0: POKE 53281, 0
170 X1 = 172: Y1 = 115: X = X1:  Y = Y1
180 SN = S1:  SL = L1:  SP = I1: GOSUB 300
190 X2 = 25:  Y2 = 51:  X = X2:  Y = Y2
200 SN = S2:  SL = L2:  SP = I2: GOSUB 300
```

Figure 9.4 Setting up Ace's radar scope

The subroutine at line 300, called from line 180 and 200, builds each sprite from DATA statements and places it at the XY coordinates using another subroutine for positioning. The image-making subroutine (which though numerically out of sequence can be entered now) is shown in Fig. 9.5. The subroutine called at line 350 is the positioner, which we'll discuss presently. The rest of this routine should be familiar.

```
300 REM  **  BUILD AND DISPLAY SPRITE
310 P = 0: POKE 53287, 1: POKE 53288, 14
320 READ A: IF A < 0 THEN 340
330 POKE SL + P, A: P = P + 1: GOTO 320
340 FOR A = P TO 62: POKE SL + A, 0: NEXT A
350 GOSUB 700
360 POKE 2040 + SN, SP
370 POKE 53269, PEEK(53269) OR 2^SN
380 RETURN
390 REM  ----------
```

Figure 9.5 Constructing the spacecraft

Now, Ace, we get down to the basic strategy of maneuvering the rescue craft into the docking bay. Here's what has to happen:

- Check the joystick and move the rescue craft.
- Check for a collision and rebound if one occurred.

- Check whether the docking is successfully completed and end the mission if it is.
- Count passes through this loop and, if it's time, jerk the Starship to a new position.
- Repeat the process.

Figure 9.6 gives the BASIC statements to put this plan into action.

```
210 REM  **  MANEUVERING LOOP
220 JM = 15 - (PEEK(56320) AND 15)
230 IF JM <> 0 THEN GOSUB 400
240 IF PEEK(53278) <> 0 THEN GOSUB 500
250 IF (X1 = X2) AND ((Y1 + 11) = Y2) THEN 280
260 JC = JC + 1: IF JC >= JF THEN GOSUB 600
270 GOTO 210
280 PRINT "HOORAY FOR ACE SPINVADER!"
290 END
```

Figure 9.6 Ace's maneuvering strategy

Just before we started on this daring mission, Ace, we mentioned that there are other ways of evaluating commands from the joystick. Figure 9.7 gives an alternate approach that you can see by comparing the IFs with the vectors shown in Fig. 9.1.

```
400 REM  **  MOVE RESCUER PER JOYSTICK
410 IF JM > 7 THEN X2 = X2 + 1: GOTO 430
420 IF JM > 3 THEN X2 = X2 - 1
430 IF JM = 1 OR JM = 5 OR JM = 9
    THEN Y2 = Y2 - 1: GOTO 450
440 IF JM <> 4 AND JM <> 8 THEN Y2 = Y2 + 1
450 X = X2: Y = Y2: SN = S2: GOSUB 700
460 RETURN
470 REM  ----------
```

Figure 9.7 Another way of responding to the joystick

We keep running across this GOSUB 700, which actually moves a sprite, so let's again go out of line-number order and see what it does. Although the movements of the two craft on the screen differ greatly, the physics of motion are common to them both, and thus we can use the common subroutine shown in Fig. 9.8.

Back in line 240, the procedure checked to see whether the rescue craft, while maneuvering, had accidentally banged against the Starship,

```
700 REM  **  MOVE SPRITE SN TO NEW XY
710 P = SN * 2: Z = INT(X + .5)
720 PV = PEEK(53264) AND (255 - 2^SN)
730 POKE 53249 + P, Y
740 IF Z > 255 THEN Z = Z - 256: PV = PV OR 2^SN
750 POKE 53248 + P, Z: POKE 53264, PV
760 RETURN
770 REM  ----------
```

Figure 9.8 Sprite positioning subroutine

and if it had, caused a rebound with GOSUB 500. We've studied the characteristics of the two spacecraft and their relative masses, Ace, and determined that if they bump together, the rescue craft will be thrown five distance units away along both the X and Y axes. The direction depends, of course, on where the rescue craft is in relation to the Starship. The rough handling you can expect in the event of a collision is listed in Fig. 9.9.

```
500 REM  **  REBOUND FROM COLLIDING WITH STARSHIP
510 IF X2 > X1 THEN X2 = X2 + 5: GOTO 530
520 X2 = X2 - 5
530 IF Y2 > Y1 THEN Y2 = Y2 + 5: GOTO 550
540 Y2 = Y2 - 5
550 IF X2 < 0 THEN X2 = 4
560 IF (Y2 < 0) OR (Y2 > 255) THEN Y2 = 115
570 X = X2: Y = Y2: SN = S2: GOSUB 700
580 IF PEEK(53278) <> 0 THEN 500
590 RETURN
595 REM  ----------
```

Figure 9.9 Reaction of rescue craft to a collision

Notice two lines in particular. In line 560, if you get thrown off the top or bottom of the screen, you'll suddenly go to its center, which is a heck of a jolt. In line 580, if you move and the two craft are still in contact, you'll be thrown again until finally you're clear of the Starship. The Starship, of course, moves periodically in a random, uncontrollable manner, which is precisely the reason for the rescue mission. If it slams into you during one of these unpredictable jumps, the same rebound factor applies and you'll be knocked out of the way.

According to the captain of the Starship, who's had little to do lately except hang on and think about his predicament, the craft jumps randomly

up to 20 distance units along each axis of movement. As an example, he calculates a jump along the X axis as

```
JX = INT((41 * RND(1)) -20)
XI = XI + JX
```

Fortunately, the erratic motion of the Starship is confined within the area covered by your radar screen. It would be an impossible situation if it did jump off the screen entirely, because then you wouldn't be able to see what you're doing. The calculations for the Starship motion are in Fig. 9.10.

```
600 REM  **  MOTION OF STARSHIP
610 JX = INT((41 * RND(1)) - 20): JC = 0
620 JY = INT((41 * RND(1)) - 20)
630 X1 = X1 + JX: Y1 = Y1 + JY
640 IF (X1 < 24) OR (X1 > 319) THEN X1 = X1 - (JX * 2)
650 IF (Y1 < 50) OR (Y1 > 229) THEN Y1 = Y1 - (JY * 2)
660 X = X1: Y = Y1: SN = S1: GOSUB 700
670 IF PEEK(53278) <> 0 THEN GOTO 500
680 RETURN
690 REM  ----------
```

Figure 9.10 Random motion of the Starship

All that now stands between this moment and the mission itself is the administrative matter of defining the shapes of the spacecraft so they'll show up properly on your radar scope. This is done in Fig. 9.11.

There you go, Ace. Have at it. This mission has no time limit, so you can take all week if necessary to dock with the Starship. When you finally do achieve success, the Good-Guy banner "HOORAY FOR ACE SPIN-VADER" will appear at the top of the screen and you can take justifiable pride in your skill.

Helpful Hints: If you want to practice docking with a stable Starship, there happens to be one in the area. You can find it by deleting line 260. This Starship stays put in the center of the screen. To resume the mission of rescuing the one that's in trouble, type line 260 back into the program. Also, the difficulty of the mission depends on the value of JF in line 130. The lower the number, the more erratic the Starship; the higher JF is, the less frequently it jerks to a new position, giving you more time to maneuver into its docking bay. A value of 50 yields a moderate level of difficulty; 20 is nearly impossible.

You'll probably want to save this program on a tape or disk, Ace, to relive your exploit and impress your friends with the level of skill you possess.

```
800 REM  **  STARSHIP DEFINITION
810 DATA 255, 255, 255,  255, 255, 255
812 DATA 255, 255, 255,  255, 255, 255
814 DATA 255, 255, 255,  255, 255, 255
816 DATA 255, 255, 255,  255, 255, 255
818 DATA 255, 255, 255,  255, 255, 255
820 DATA 255, 231, 255,  255, 195, 255
822 DATA 255, 129, 255,  255,   0, 255
824 DATA 254,   0, 127,  252,   0,  63
826 DATA 248,   0,  31,  240,   0,  15
828 DATA 224,   0,   7,  192,   0,   3
830 DATA 128,   0,   1, -1
840 REM  **  RESCUER DEFINITION
842 DATA   0,  24,   0,    0,  60,   0
844 DATA   0, 126,   0,    0, 255,   0
846 DATA   1, 255, 128,    3, 255, 192
848 DATA   7, 255, 224,   15, 255, 240
850 DATA  31, 255, 248,   63, 255, 252
852 DATA 127, 255, 254,  255, 255, 255
854 DATA 127, 255, 254,   31, 255, 248
856 DATA   1, 255, 128, -1
RUN
```

Figure 9.11 Definitions of the sprites

With this triumph, Ace Spinvader and I leave behind the Adenoids, deep space, sprites, and all the other trappings of computer graphics. We're going to retire and go back down to earth in order to devote ourselves to sound effects and music. You're invited to come along, but after a time you'll probably get a hankering to make visual adventures that will outshine those of Ace and myself. Certainly you now have the background and experience to do so.

CHAPTER 10
THE SOUND OF MUSIC

The voice of the Commodore 64 has been silent since the end of Chapter 2, when it spoke ingloriously of a stick banging on a garbage can. Let us now raise its voice in a song everybody knows, though it might not look familiar in Fig. 10.1.

Old MacDonald has had that farm for a long, long time, and now computers have touched even him. Want to hear it faster? In line 110, change TS to 240 and rerun the program. TS is the time signature, a term familiar to those who know something about music. It's also called the tempo. For a slower beat, change TS to 120. At 60 it sounds like something played at Old MacDonald's funeral, and even more so if you change RO in the same line to 1. The RO is the relative octave, which selects the reference point for the notes within the eight-octave range of the Commodore 64. When RO = 1, it's close to the bottom (RO = 0 makes it so low you can hear the speaker cone moving in and out). Set TS at 600 and RO at 6 and try it now. Sounds like a frantic piccolo, doesn't it?

Right now this program probably makes little sense to you, but it's fun to mess around with these two values and hear what a difference they make in the sound coming out of the speaker. This is but one of the three voices of the Commodore 64 synthesizer, and only one of the infinitely variable sounds they can make.

```
100 REM  **  DOWN ON THE FARM
110 TS = 180: BT = 22140/TS: S = 54272: RO = 3
120 FOR P = 0 TO 24: POKE S+P, 0: NEXT P
130 POKE S+5, 9          :REM  GUITAR A/D
140 POKE S+2, 255        :REM  LOW PULSE RATE
150 POKE S+24, 15        :REM  VOLUME
160 REM  **  PLAY TWICE
170 FOR R = 1 TO 2       :REM  REPEATS
180 READ NV, NL          :REM  GET NOTE, LENGTH
190 IF NV < 0 THEN 280
200 NP = NV + (RO * 12) :REM  RAISE TO OCTAVE
210 FQ = 2^(NP/12) * 268.234
220 HF = INT(FQ/256): LF = FQ - (HF * 256)
230 POKE S+0, LF: POKE S+1, HF
240 POKE S+4, 65         :REM  PULSE WAVEFORM
250 FOR T = 1 TO (NL * BT): NEXT T
260 POKE S+4, 64         :REM  SILENCE VOICE
270 GOTO 180             :REM  NEXT NOTE
280 RESTORE              :REM  BACK TO START
290 FOR T = 1 TO (NL * BT): NEXT T
300 NEXT R
310 FOR P = 0 TO 24: POKE S+P, 0: NEXT P
320 END
330 DATA 12, 1,  12, 1,  12, 1,   7, 1
340 DATA  9, 1,   9, 1,   7, 2,  16, 1
350 DATA 16, 1,  14, 1,  14, 1,  12, 2, -1, -1
RUN
```

Figure 10.1 A familiar refrain

Music Theory 101

Before we get into the control of the synthesizer itself, let's discuss music in general for a moment. The synthesizer can make sounds other than pure music, as we know from Chapter 2, but the concept of notes and octaves and duration and the like are central to all sound effects. Music, after all, is only organized noise, although Beethoven might have differed with that definition.

Of all instruments, the piano is the most familiar, and so we will relate this discussion to it. The keyboard of a piano is shown in Fig. 10.2. It consists of seven clusters of keys like those shown, plus three at the bottom (left) of the keyboard. Each cluster of keys is called an octave because it consists of eight full steps represented by the white keys

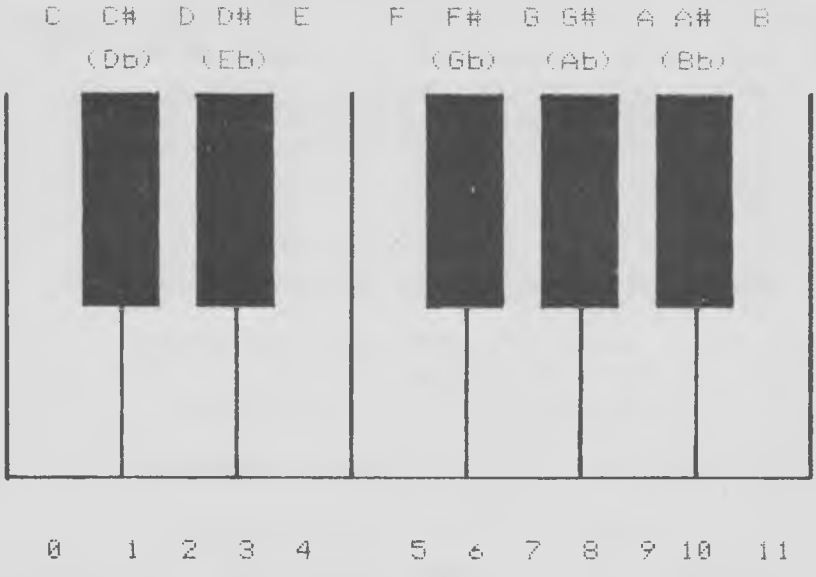


Figure 10.2 Piano keyboard notes

(octave comes from the Latin word for eight). The keys have letters for names, and the names recur in each octave. Usually people think of an octave as going from C to the next higher C, although an octave can span any eight full steps. What makes an octave significant is that, as you progress up the keyboard, the frequency of a given note doubles each octave. "Middle C," the fourth C from the left on the keyboard, has a frequency of 440 hertz; the C above middle C is 880, the one below 220. Thus all the notes of the same name sound the same to us, only higher or lower.

In fact, an octave does not really have eight full steps, but rather twelve half-steps. The "full steps" between E and F and between B and C are actually half-steps, but in our music system they sound full. The Chinese use a different tonal progression, which makes their music sound alien to us, and probably they think ours sounds funny too. At any rate, the black keys represent the half-steps, known as sharps and flats. The black key between C and D can be called either C# or D♭ ("C-sharp" or "D-flat") depending on the mood of the speaker; they are the same note. There is no E# key since a half-step separates E and F, but F is E# and E is F♭ .

The easiest octave goes from C to C because it's the only octave that sounds right without playing any sharps or flats. If you start a scale on G, it only sounds right if you play F# instead of F as you go note by

note. Similarly, if you start the scale on F, you have to play B ♭ instead of B to make it come out right. This is because of the inconsistency of having half-steps between the third and fourth notes and the seventh and eighth. The starting note of the scale is its "key." A composition written with no sharps or flats consistently occurring has C as its reference note and is thus in the key of C. The scale that starts on G has one sharp and is in the key of G; with one flat, in the key of F, etc.

Introduction to computer sound

A music synthesizer doesn't know or care about notes, keys, sharps, flats, and all that stuff. It merely oscillates at a specified frequency, introducing certain sound characteristics as it does so. The synthesizer of the Commodore 64 can produce an infinitely varying tone frequency, with which we will experiment shortly. To produce pitch-true notes that form music, we have to define for it the mathematical relationships among the frequencies representing musical notes. This is not as intimidating as you might think, and in the next section we'll see how to do it. We also have to specify the characteristics of the sounds we want, such as the waveform, attack, decay, and other things that will become clear as we go along.

For this, the Commodore 64 reserves a 24-byte block of memory starting at address 54272. Each of the 24 bytes has an assigned control function, and our desires are communicated to the synthesizer by POKEing numeric values into them. They give us the capability of controlling up to three independent voices simultaneously, of synthesizing the tones of various musical instruments or creating new and never-before-heard (sometimes for good reasons) sounds, of varying the volume, and so on.

The device in the Commodore 64 that produces sounds in response to the values POKEd into its control registers is a cousin of our old friend VIC. Its name is SID, short for Sound Interface Device. SID is a chip that functions independently of the computer's central processor; it can make sounds while the computer goes about other tasks. A sound, once begun by setting the SID control registers, continues until the registers are changed, regardless of what other jobs the machine might be doing. Thus, you control the CPU with your program, and the CPU controls SID by POKEing values into its registers at appropriate times.

Figure 10.3 is a chart showing all the control registers of the sound interface device and their purposes. Much of the terminology it uses will be unfamiliar at this point, of course, but I will frequently refer to the chart throughout this chapter until, by the end, you will fully understand it and, through it, how to control the synthesizer.

Address	54272+	Voice	Control
54272	0	1	Note LF component
54273	1	1	Note HF component
54274	2	1	Pulse rate low component
54275	3	1	Pulse rate high component
54276	4	1	Waveform
54277	5	1	Attack/decay
54278	6	1	Sustain/release
54279	7	2	Note LF component
54280	8	2	Note HF component
54281	9	2	Pulse rate low component
54282	10	2	Pulse rate high component
54283	11	2	Waveform
54284	12	2	Attack/decay
54285	13	2	Sustain/release
54286	14	3	Note LF component
54287	15	3	Note HF component
54288	16	3	Pulse rate low component
54289	17	3	Pulse rate high component
54290	18	3	Waveform
54291	19	3	Attack/decay
54292	20	3	Sustain/release
54293	21	All	Filter low cutoff frequency
54294	22	All	Filter high cutoff frequency
54295	23	All	Resonance control
54296	24	All	Volume control (low nibble) Filter select (high nibble)

Figure 10.3 Control registers for the Sound Interface Device (SID)

Making notes

A sound is caused by the vibration of a substance that sets up waves that move through the air. In the case of the Commodore 64, the "substance" is the speaker of the TV set, whose cone moves in response to the signals generated by any or all of the three oscillators in the SID chip. An oscillator is a circuit that causes a current to fluctuate up and down from 0 volts; the farther the current fluctuates, the louder the sound; the more rapidly it fluctuates, the higher the frequency. Musical notes differ from noise by occurring at expected frequency intervals that our ears mathematically evaluate and find pleasing. To make musical notes with the Commodore 64, then, we need to establish these mathematical intervals.

The frequency of the notes in each octave is twice the frequency of the corresponding notes in the octave below, as we mentioned before.

Though you might not recognize it, this is precisely the concept that underlies binary numbering: doubling value with each step (bit or octave) in a progression. With respect to the zeroth octave, the first octave is twice the frequency, the second four times, the third eight times, etc. Another way to express this, familiar from our bit-setting days, is that the octave frequency is two raised to the power of the octave number, or in BASIC,

```
F = 2^N
```

where F is the octave frequency factor and N is the octave number (offset from zero).

The result is a multiplier that we can apply to low C in order to calculate the frequency of any other C. If low C has a frequency of 55 hertz (cycles per second, abbreviated Hz), then middle C, the start of octave 3, has a frequency of

$$2^3 \times 55 = 440 \text{ Hz}$$

It follows that if we can compute the frequency of any C with respect to low C, we can calculate the frequency of *any* note by using the relative octave and the note's position within it and applying the factor to a common base value. Figure 10.1 gives note numbers (positions) for each of the twelve half-steps that comprise an octave: C is note 0, E is note 4, etc. The position of a note with reference to low C is

```
Note number + (octave × 12)
```

The E above middle C is note 4 within octave 3. According to this formula, its position from low C is

```
NP = 4 + (3 × 12) = 40
```

or, in other words, it's 40 half-steps above low C. The frequency doubles every octave and each note contributes to this by 1/12, so the note factor is 2 raised to the power of NP/12, or

```
NF = 2^(NP/12)
```

in BASIC notation. If 55 hertz is the base value (frequency of low C), the frequency of the E above middle C is

```
FQ = 2^(NP/12) * 55 = 554.37 Hz
```

The Commodore 64's oscillators use numbers that are related to frequency, but not the frequencies themselves. Instead of 55, the common base for developing frequencies with respect to low C is 268.234. To play low C, POKE the value 268; the oscillator value for the E above middle C is calculated as

```
2^(40 / 12) * 268.234 = 2703.63
```

Relating this back to Fig. 10.1, line 110 gives the relative octave RO and the first number of each pair in the DATA statements gives the note. Line 200 then computes the note position and 210 figures out its corresponding oscillator value. In Old MacDonald, the tune covers portions of two octaves, ranging from G (7) up to E (given here as 16). We started exactly one octave above the relative octave RO—in other words, at the start of octave 4—so that we wouldn't have to use negative numbers to represent notes below the starting note and thereby cause the song to end prematurely because of the end-of-data check in line 190. The constant 268.234 provides the proper base and spread for the note frequencies. The relative octave is simply a point of reference that can be changed to suit circumstances. It need not be an integer; you can change it to any fractional value between 0 and 7 and the music tones shift, but their relationships to each other remain correct. Try playing Old MacDonald using RO = π; though the pitch is higher than when RO = 3, it still sounds fine.

If you read Chapter 7 and Appendix M of your *User's Guide,* you will find that the approach given there differs greatly from this one, stressing instead the use of long complicated tables of note values. The method in this book greatly simplifies musicmaking on the Commodore 64 by calculating the music note values for you and letting you concentrate on the creative aspects of the synthesizer rather than on the clerical tedium of table lookups.

Setting up the note for the oscillator

Once a program has calculated the oscillator setting for a note, it has to communicate that value to the synthesizer. This is done with POKEs into the SID control registers' "note component" locations shown in Fig. 10.3.

The chart indicates that for Voice 1, location 54272 is the note LF (low-frequency) component and 54273 is the note HF (high-frequency) component. These terms are potentially misleading, since a note has only one frequency, and "component" is therefore the operative word. The HF component is the number of times the oscillator setting is divisible by 256 and the LF component is the remainder, computed in BASIC as

```
HF = INT(FQ / 256)
LF = FQ - (HF * 256)
```

The results are two numbers, both less than or equal to 255, that can be POKEd into single bytes. You can find the instructions that do this for Old MacDonald at lines 220 and 230 in Fig. 10.1.

As an example, earlier we calculated the oscillator setting for the E above middle C as 2703. Running FQ = 2703 through these equations,

HF = 10 and LF = 143. You can then set up the Voice 1 oscillator by POKEing them into 54273 and 54272, respectively. If you want to generate this note in Voice 2, POKE them into 54280 and 54279, or into 54287 and 54286 for Voice 3.

Save Old MacDonald on a tape or diskette. We'll rejoin him on the farm presently, but right now we're going to experiment with the almost infinite variability of frequencies that can be produced by the SID oscillators. Type and run Fig. 10.4.

```
100 REM  **  SLIDING TONE
110 S = 54272
120 FOR P = 0 TO 24: POKE S+P, 0: NEXT P
130 POKE S+5, 129        :REM  ATTACK/DECAY
140 POKE S+6, 129        :REM  SUSTAIN/RELEASE
150 POKE S+24, 15        :REM  VOLUME
160 POKE S+4, 17         :REM  TRIANGLE WAVE
170 FOR HF = 0 TO 255: POKE S+1, HF
180 FOR LF = 0 TO 255 STEP 16
190 POKE S+0, LF: PRINT HF, LF
200 NEXT LF: NEXT HF
210 FOR P = 0 TO 24: POKE S+P, 0: NEXT P
220 END
RUN
```

Figure 10.4 Infinitely variable frequencies demonstration

This program immediately begins printing a listing of the HF and LF components while, at the same time, producing the tone they generate from the Voice 1 oscillator. It runs for a long time, at first producing no audible sound but then, as the HF gets above 2, starting to output what sounds like a motorcycle accelerating in the distance. The tone rises fairly rapidly for a time, becoming a whistle, but then the rise in pitch slows until it is scarcely perceptible (when HF is above 120 or so). The frequency itself is rising at a steady rate, but because it is high, it takes increasingly greater changes in frequency to produce a discernible change in pitch. Remember, in each octave the frequency is double what it was in the previous octave. What you are hearing is audible proof of this phenomenon. And though the sound seems to increase steadily in the lower ranges, the LF component is actually STEPping by 16, to give us $16 \times 256 = 4096$ tones. It takes 4 minutes and 9 seconds for the program to run to completion. You can take out the STEP 16 from line 180 to hear every single frequency the SID oscillator is capable of producing,

but that will take 1 hour, 6 minutes, and 24 seconds. Go right ahead if you have the time, inclination, and auditory endurance.

A better idea is to reload Old MacDonald.

Waveforms

The SID oscillators are capable of producing four basic waveforms, which greatly alter the character of sound. As written, the program in Fig. 10.1 uses a pulse wave, which makes a rather sharp twangy sound that emulates a guitar.

Let's try a different waveform. Retype the following lines to read

```
240  POKE S+4, 17
260  POKE S+4, 16
```

and rerun the program. The sound is now more like a marimba. This is because we told the SID to produce a triangular waveform. Now change these lines as follows:

```
240  POKE S+4, 33
260  POKE S+4, 32
```

This is a sawtooth waveform, which sounds like a harpsichord. Each waveform has a distinctive sound that, along with the attack/delay and sustain/release settings we'll discuss shortly, can synthesize an astonishing variety of musical instruments and sound effects.

And speaking of sound effects, there is one additional waveform we haven't yet mentioned, called simply "noise." Change line 240 to POKE the value 129 into S+4 and run the program. Aha! The guy is beating the garbage can again! But listen—he's got several garbage cans that generate different notes, because the crashes alter in pitch to convey the old familiar refrain.

Back to the pulse waveform: this is the most versatile of the waveforms because you can dramatically alter the character of its sound by changing the width of the pulses. The following program changes give a sampling of its diversity:

```
170  FOR R = 0 TO 3
175  POKE S+3, R * 4
240  POKE S+4, 65
260  POKE S+4, 64
```

This time as you run the program, the theme repeats four times due to the loop count in 170. Line 175 alters the width of the pulse in each iteration so that we start out with the sound the program had the first time it ran, and each repeat has a different timbre as the pulse width increases.

Figure 10.5 shows the form of a pulse wave. Unlike most sound waves, this one is square. The "corners" of a sound wave consist of harmonics—multiples of the basic frequency—that make the tone fuller, and because a pulse wave turns four corners during each cycle it generates more harmonics than any other. The width of the pulses affects the sound by governing how much of the harmonic content you hear. When the pulses are evenly spaced, all the harmonics are audible and the sound is at its fullest.

There are two memory addresses for each voice in the SID registers that govern the pulse width. For Voice 1, they are 54274 and 54275, which are designated in Fig. 10.3 as "Pulse rate low component" and "Pulse rate high component." As with the frequency, there is only one pulse rate per voice and too many possibilities to hold in one 8-bit register, so they're split into two. In this case, however, only the lower nibble of the high component is used, giving 8 bits in the lower component and 4 bits in the high for a total of 12 bits, or $2^{12} = 4096$ possible pulse widths. The pulse width is not an absolute number, but rather a ratio of high width to low width. When the pulse width is 0, the pulses have no width and therefore no sound is heard. Similarly, when the pulse width is 4095, there is no sound because the wave is all high pulses with no lows. In the middle (a pulse width value of 2048) the ratio of high pulse width to low pulse width is 1:1, producing a square wave that has the fullest sound because the harmonics are in balance. The sound thins on either side of 2048.

Now that you understand something of the theory, make one more change in the program to hear it in action. Replace line 140 with the following:

```
140  POKE S+2, 0
```

This places a zero in the pulse width low component, so that the pulse width is entirely governed by the high component POKEd in line 175. As the program runs, it makes no sound the first time through the tune, except for a subdued ticking as the synthesizer generates a pulse wave with no pulses. The first repeat makes tones that seem rather full and pleasing, but the third time through, it reaches the peak of fullness, because the width value is 2048 on this pass. On the final repeat, the tone is again thinner because the pulse high phase is 75 percent of the signal.

The sawtooth and triangular waveforms are less complex. You can't vary their characteristics the way you can pulse widths. These waveforms are shown in Fig. 10.6, and it's obvious where they got their names. A sawtooth wave has a twangy, penetrating, sometimes unpleasant quality, while a triangular wave sounds rather hollow and soothing (harpsichord versus marimba).

There are no hard-and-fast rules for choosing one waveform over another. It depends upon your preferences and some experimentation, es-

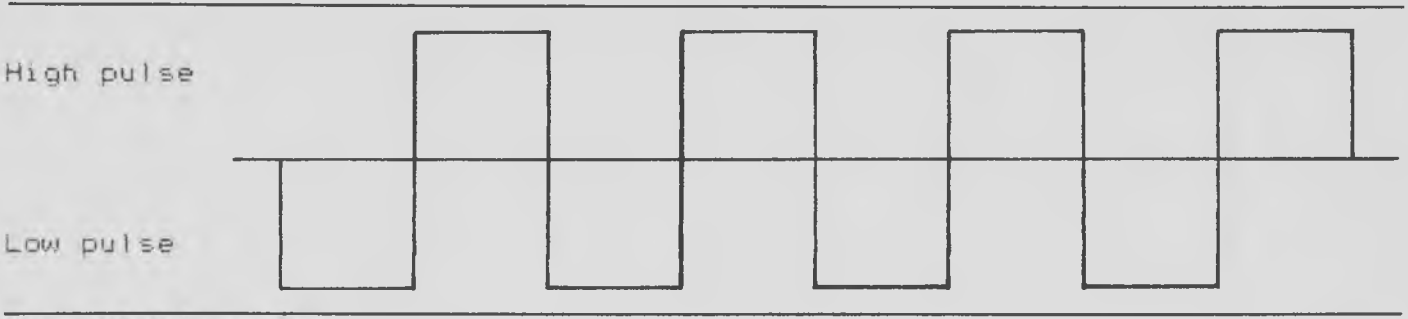


Figure 10.5 Pulse waveform

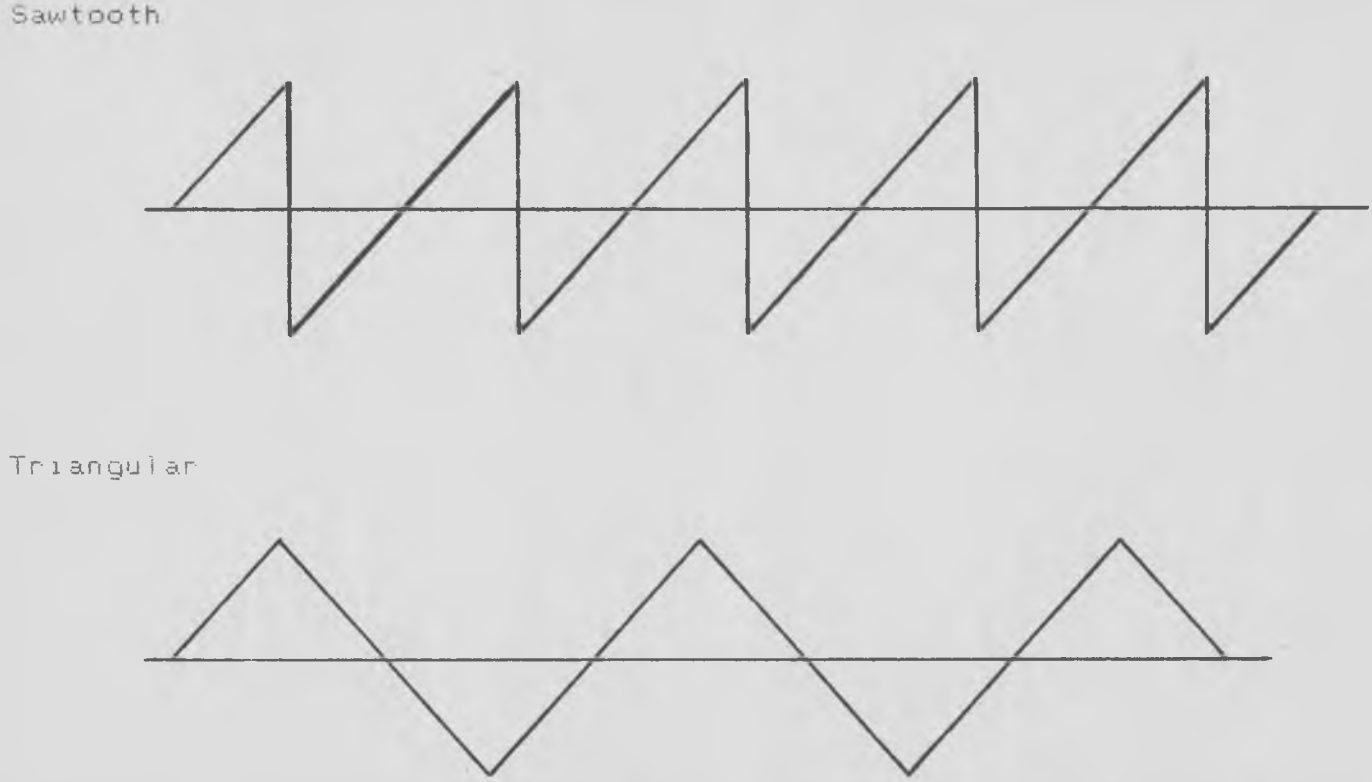


Figure 10.6 Sawtooth and triangular waveforms

pecially with pulse widths. Later I will suggest some waveform choices that simulate different instruments.

The waveform is selected by POKEing a value into the waveform register for the voice that is to sound it: 54276 for Voice 1, 54283 for Voice 2, 54290 for Voice 3. All other parameters for the voice must be already set up, because when you POKE a number into the waveform register, you turn on the sound.

The waveform selection value is one bit in the high-order nibble with the low bit of the byte turned on, giving an odd number. To turn the voice off, you POKE a value into the waveform register that is one less than the waveform selection number (low bit off) (see Fig. 10.7.).

Waveform	Turn on	Turn off
Triangular	17	16
Sawtooth	33	32
Pulse (must also set width)	65	64
Noise	129	128

Figure 10.7 Waveform selection values

The waveform registers work differently than other control bytes in the Commodore 64. As soon as you POKE a value into one, the SID chip copies it elsewhere and resets the register to 0. It continues to play the note with the selected waveform until you place another value into the waveform register. This means, however, that you cannot PEEK at the register to get back the value you POKEd there, because the waveform register always contains 0.

The ADSR envelope

As it plays, a note goes through four phases. These phases occur no matter what the source of the sound (and apply to nonmusical sounds as well), but the intensity of each phase varies according to the characteristics of the source. A trumpet and a harpsichord produce very similar waveforms, but they certainly don't sound much alike because the phases of the notes they make differ greatly.

When the note starts, sound rapidly rises from silence to a peak volume. This is called the *attack*. On a plucked instrument, the attack rate is very fast, giving each note a sharp start; on an air-activated instrument, such as a clarinet or an organ, the attack rate is quite slow.

Having reached peak volume, the sound then falls away to a middle range. The rate at which it drops from the attack peak is called the *decay*.

A guitar has a high decay rate because the string, after a few vibrations, settles down. A horn, on the other hand, has almost no decay because the player is blowing it.

The middle range at which the note is really heard is its *sustain* phase. Sustain comes as soon as the initial attack has decayed, and usually lasts the longest of any of the phases. In a pipe organ, for example, the sustain phase lasts for as long as the player holds down the key, and in a horn for as long as the person playing it has breath.

The final phase is *release*, the rate at which the note drops to silence. In a guitar the release phase is a tapering off of the sound, while in a horn the sound ends the instant the player stops.

Figure 10.8 illustrates the four phases of a sound, which are called its "ADSR envelope" (for Attack/Decay/Sustain/Release).

Just as it greatly altered the sound to change waveforms or pulse widths, so are tremendous variations and shadings of a given waveform created when the ADSR envelope is changed. Line 130 in Old MacDonald sets up the attack/decay parameters for the fastest attack and a medium decay rate. We can change to a medium attack by retyping the instruction to read

```
130  POKE S+5, 8 * 16
```

and running the program again. See what a difference it makes? Each note now slides to its peak and then slides back down to silence, giving them a "yipping" sound caused by a relatively slow attack and rapid decay.

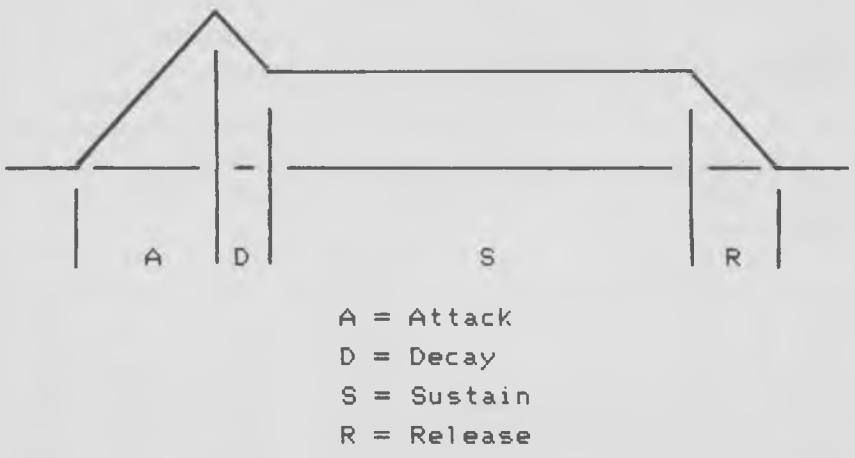


Figure 10.8 The phases of a sound

Also, there is no sustain/release setting, so the note is not held. For that, add the statement

```
130  POKE S+6, 4 * 16
```

which creates a medium sustain level (the note is held after the decay phase at half the volume of the peak that occurs during the attack). The problem with these settings is that the result doesn't sound very pretty. Each note almost seems like two notes played at the same frequency in rapid succession. The reason? We have the fastest decay rate following a fairly slow attack, so that the volume rises in rather leisurely fashion and then abruptly drops to the sustain phase. To correct this, change line 130 to

```
130  POKE S+5, (8 * 16) + 8
```

That's better, since now the level slips off the peak rather than plummeting from it. The notes sound smoother because each phase slides into the next. The sound is improved even more by raising the sustain level close to the attack with

```
135  POKE S+6, 8 * 16
```

No doubt all these numbers we're POKEing are mysterious even though you can hear their effects, so we'll disclose their secrets. Attack/decay is controlled by register S + 5 and sustain/release by S + 6 in Voice 1. These two registers are both divided into nibbles, with attack in the high-order and decay in the low-order nibbles of S + 5, and sustain in the high and release in the low nibbles of S + 6. Thus, by POKEing a value from 0 to 15 into each nibble, we independently control each phase of the notes played in that voice. You'll recall that, to POKE a value into the high-order nibble, you multiply it by 16, which explains the multiplication in the preceding examples.

For attack, decay, and release, the fastest rate is 0, the slowest is 15, and you can vary the rates by selecting a setting anywhere between those extremes. For sustain, the higher the setting, the closer the note's holding level to the peak that occurs during the attack phase. Figure 10.9 summarizes this.

		V1	V2	V3
Attack	(0 = fastest, 15 = slowest) × 16	54277	54284	54291
Decay	(0 = fastest, 15 = slowest)	54277	54284	54291
Sustain	(0 = lowest, 15 = highest) × 16	54278	54285	54292
Release	(0 = fastest, 15 = slowest)	54278	54285	54292

Figure 10.9 ADSR settings

When we first played Old MacDonald, the sustain/release settings were 0, meaning that the notes were not held at all after the decay phase. Furthermore, the attack rate was also 0, giving the sharpest possible attack, and the decay rate was 9, slightly slower than medium. Consequently, the notes had a staccato sound, since they consisted almost entirely of the decay phase. The experiments just performed reset the attack and decay to slower rates and raised the sustain level, which softened the notes and gave the impression of an accordion. This result leads to some general observations about the relationships of the ADSR components in producing "good" sounds:

- The slower the attack rate, the slower the decay should be and the higher the sustain level.
- Conversely, the faster the attack rate, the faster the decay and the lower the sustain level should be.
- Notes played in rapid succession should have a fast release rate (otherwise the synthesizer makes a "poof" sound by cutting off the note before it has fallen silent).

Some typical musical instruments

How an instrument sounds is a function not only of its waveform and its notes' ADSR characteristics, but of the listener's ear as well. You might find that some of the suggested settings in Fig. 10.10 don't quite sound to you like the instruments they synthesize and that's okay. These are not absolute settings, but rather a basis for further adjustment until you get a sound that, to your ear, most closely approximates these familiar instruments.

	Waveform	A	D	S	R
Guitar	Pulse (W = 255)	0	9	0	0
Flute	Triangular	4	12	12	4
Clarinet	Triangular	4	8	8	0
Harpsichord	Sawtooth	0	9	0	0
Marimba	Triangular	0	9	0	0
Pipe organ	Triangular	0	0	15	0
(with echo)	Triangular	0	0	15	8
Accordion	Triangular	6	6	15	4
Brass horn	Sawtooth	3	6	8	0

Figure 10.10 Familiar instrument settings

Multiple voices

I expect we're both getting weary of Old MacDonald, so let's leave the bucolic fellow and venture into greater sophistication with the program in Fig. 10.11.

The values V1 and V3 (for Voice 1 and Voice 3) in line 120 establish the starting points for the lowest and highest voices' control addresses in the SID registers. Line 130 clears the SID registers—always a good practice when starting a sound program—and 140 sets the volume control at maximum. The chord-building loop runs between lines 150 and 250. The "V" in the FOR loop stands for the Voice and, because it steps from 0 to 14 by 7s, provides the offset for addressing voices in the SID registers: first pass addresses Voice 1, second Voice 2, third Voice 3. The note is obtained by READing the next item from the DATA state-

```
NEW
100 REM  **  3-NOTE CHORD
110 S = 54272: RO = 4
120 V1 = 0: V3 = 14
130 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
140 POKE S + 24, 15      :REM  VOLUME
150 FOR V = V1 TO V3 STEP 7
160 READ N
170 NP = N + (RO * 12)
180 FQ = 2^(NP / 12) * 268.234
190 HF = INT(FQ / 256): LF = FQ - (HF * 256)
200 POKE S + V + 0, LF           :REM  NOTE LF
210 POKE S + V + 1, HF           :REM  NOTE HF
220 POKE S + V + 6, 15 * 16      :REM  S/R
230 POKE S + V + 4, 17           :REM  TRIANG WF
240 FOR T = 1 TO 369: NEXT T     :REM  DELAY
250 NEXT V
260 REM  **  DIMINISH VOLUME
270 FOR X = 15 TO 1 STEP -1
280 POKE S + 24, X
290 FOR T = 1 TO 369: NEXT T
300 NEXT X
310 FOR X = 0 TO 24: POKE S + X, 0: NEXT X
320 END
330 DATA 0, 4, 7
RUN
```

Figure 10.11 Singing in harmony

ment, with note numbers corresponding to key numbers from Fig. 10.2. Lines 170–190 calculate the oscillator frequency settings, as discussed earlier in this chapter, and lines 200 and 210 POKE them into the current voice's registers. Line 220 sets the sustain/release levels for a pipe organ and POKEs them into the S/R control location. Finally, line 230 turns on the triangular waveform for the current voice, and the loop repeats after a delay of 1 second (it takes about half a second to process each note, so the timing loop lasts for another half-second). Musicians call the measured building of a chord in this fashion an arpeggio: here it demonstrates a method of controlling multiple voices.

The loop between lines 260 and 300 gives you an idea of the volume range available through the manipulation of address 54296 (S+24). You can use this register to introduce crescendos and decrescendos into music ("dynamics," in musical terms) and to make other sound effects more interesting and realistic. This loop steps downward from 15 to 1, POKEing the current loop count into the volume control each time. Notice that you can hear the volume drop in the lower levels: just as the ear is more sensitive to low-frequency fluctuations, it is also more sensitive to changes in low volumes.

You can play with this program to produce other chords. Change the DATA in line 330 to 4, 7, 10 to hear a diminished seventh chord. You can play in a minor key with 0, 3, 7. It's also possible to change the octave by giving a different value for RO in line 110, and if you change it to a fractional number such as 4.39756, the notes retain their proper musical relationship even though you have not specified a true octave number.

Using the jiffy clock to control tempo

The Commodore 64 has a built-in "jiffy clock" that you can use to time events. The reading of this clock is available from BASIC by sensing the value of the standard variable TIME. To save the current time, for example, you can issue the statement

```
CT = TIME
```

and CT will then contain a number that came from the jiffy clock. This clock "ticks" 60 times a second, meaning that every 1/60 of a second its reading increases by one. If you want to time how long an event lasts, you can start the instructions with T1 = TIME and end them with T2 = TIME, then find the duration with T2 - T1, which figures the time in increments of 1/60 second. To reduce the elapsed time to whole seconds, divide by 60.

Similarly, you can use TIME in a loop that delays action for a specified minimum interval (the "beat"). In a music program:

- Get the TIME("T1") when the note starts;
- While the note plays, READ the next note and perform necessary calculations;
- Enter a loop that repeatedly compares T1 to the current TIME until the interval necessary to maintain the tempo has elapsed;
- Set up and start the next note, then repeat.

The usual method for establishing the tempo is to decide on the number of beats per minute. A quarter-note gets one beat, an eighth-note a half-beat, a whole-note four, etc. With a tempo of 60 (beats per minute), there is a beat per second and the jiffy clock advances by 60; with a tempo of 120, the clock advances by 30 for each beat.

Using this reasoning, you can set the tempo for a composition by giving to the variable BT the number of "ticks" of the jiffy clock per beat: for a tempo of 120, BT = 30; for 60, BT = 60; etc. Each note is then given a duration value based upon its length in comparison with one beat. A half-note (gets 2 beats) has a duration of 2; an eighth-note (gets a half-beat) has a duration of 0.5. You can then apply this duration value in the loop that checks the elapsed time. The following example demonstrates this in a practical application.

Controlling multiple voices

The availability of three voices might seem rather limiting to those with a musical background, and, in fact, it is a pity Commodore didn't include a fourth voice: choirs sing in four-part harmony, rounds almost invariably consist of four parts, a full chord has four notes, etc. Nevertheless, it's astonishing what this synthesizer can do with three, as the simple composition that follows demonstrates.

Before we become awestruck, however, we need to discuss some elements of control when using multiple voices. First, you should always set up all the parameters, including the note frequency components, *before* POKEing the waveform for the voice (since the waveform turns on the voice). Second, don't alter anything except the note and waveform from one note to the next; it is unnecessary to reset ADSR values once they have been POKEd into the voice's registers. Third, if two or three notes all start at the same time, calculate and POKE the frequency components for all three, and then POKE the waveform for all three. If you start one voice, then work up and POKE the frequency and start the next, the second note will begin noticeably later than the first.

The musical score is contained in either DATA statements or a file. It's essential to structure the data in some sort of consistent fashion so that the program can act on them properly, and so that you can encode them fairly easily. One way, which we'll use in the forthcoming program, involves setting up records of four elements each in which:

- Item A indicates the voice that is to play the note;
- Item B is the note number;
- Item C gives the note's duration;
- Item D is an action indicator (0 means get the next note before playing, nonzero means play now).

This is not a comprehensive music control structure, because it makes no provision for changing ADSR settings, waveforms, volume, etc., "on the fly" (as the music is playing). Those things could be included by using false voice numbers in Item A, such that a 4 would signal the new volume level contained in Item B, a 5 would indicate a change in ADSR for Voice 1, and so on. I chose not to do that here in order to avoid undue complication.

Figure 10.12 gives the general outline of the music-playing program developed in Fig. 10.13.

If you don't read music, you may be unfamiliar with the concept of a *rest*, and since that plays a part in the following program, we'll digress briefly to discuss it. A rest occurs when a voice is silent, or in other words, it's a note without a sound. Just like played notes, it has a specific duration measured by the beat of the music. Since music is a matter of timing, rests synchronize the voices. In this piece, Voice 1 begins by itself with a two-beat note. At the start of the second beat, while Voice 1 is still holding the note, Voice 2 joins in. Thus, Voice 2 begins the song with a one-beat rest. This is a frequent pattern throughout the piece and gives it a syncopated feel. To create a rest, then, the DATA statements occasionally include a Voice 2 note number of 0, and one of the calculation steps sets the frequency to 0 in that case. You can see this in the very first DATA statement of the program with the data 2, 0, 1, 9, which means "Voice 2, Note 0, 1 beat, play."

The piece we'll play here is Melody in F, by Anton Rubinstein (Fig. 10.13). You might also recognize it by the title of the popular song "Welcome, Sweet Springtime," which uses the same melody. It's a pretty piece whose theme is played by a flute with a guitar accompanying (to show that the voices need not have the same waveforms or characteristics). Don't be daunted by its length. It takes a lot of DATA statements to make music, and, in fact, the data portion is longer than the program itself.

1. Set up arrays NN(3) for new-note flags, N(3,4) to hold information related to the current note in each voice.
2. Declare constants for the program, clear SID registers, set volume and ADSR parameters for all voices.
3. Loop for each beat of the music:
   a. Clear the new-note flags
   b. Read the next record and:
      (1) If at the end of data, go to 4 below.
      (2) Set the new-note flag for the voice in which the note plays.
      (3) Calculate the frequency components and save in array N(V,x).
   c. Delay further action for the balance of the duration of the shortest note now playing.
   d. For each new note:
      (1) Turn off the voice.
      (2) POKE the new note frequency components.
   e. For each old note (not replaced by a new one):
      (1) Decrease the remaining duration by the amount of time elapsed since the most recent beat interval began.
      (2) If the remaining duration has dropped to 0 or less, turn off the voice for this note.
   f. For each new note, turn on its voice.
   g. Get the time from the jiffy clock.
   h. Find the shortest remaining duration (that is greater than 0) among the three voices.
   i. The beat delay D is this value multiplied by the number of jiffy clock ticks per beat.
   j. Repeat from 3.a above.
4. At the end of data:
   a. Hold the playing notes until their duration has expired.
   b. Clear the SID registers.
   c. Stop the program.

Figure 10.12 Plan for a music program

```
NEW
100 REM  **  WELCOME, SWEET SPRINGTIME
110 DIM NN(3), N(3,4): CLR
120 BT = 20: RO = 3: S = 54272
130 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
140 POKE S+24, 12
150 N(1,4) = 17: N(2,4) = 65: N(3,4) = 65
160 POKE S+5, (4 * 16) + 8       :REM V1 A/D
170 POKE S+6, (8 * 16) + 0       :REM V1 S/R
180 POKE S+9, 255: POKE S+12,9   :REM V2 PW, A/D
190 POKE S+16,255: POKE S+19,9   :REM V3 PW, A/D
200 REM  **  SET UP NEXT NOTE(S)
210 FOR X = 1 TO 3: NN(X) = 0: NEXT X
```

```
220 READ V, NV, N(V,3). A: NN(V) = 1
230 IF A < 0 THEN 590
240 IF NV = 0 THEN FQ = 0: GOTO 270
250 NP = NV + (RO * 12)
260 FQ = 2^(NP/12) * 268.234
270 N(V,2) = INT(FQ/256): N(V,1) = FQ-(N(V,2) * 256)
280 IF A = 0 THEN 220
290 REM  **  HOLD NOTES TO END OF BEAT
300 IF (TIME - T1) < D THEN 300
310 REM  **  PROCESS NEW NOTES
320 FOR V = 1 TO 3: VL = (V - 1) * 7
330 IF NN(V) = 0 THEN 370
340 POKE S + VL + 4, N(V,4) - 1
350 POKE S + VL + 0, N(V,1)
360 POKE S + VL + 1, N(V,2)
370 NEXT V
380 REM  **  DECR DURATION OF OLD NOTES
390 FOR V = 1 TO 3: VL = (V - 1) * 7
400 IF NN(V) <> 0 THEN 440
410 N(V,3) = N(V,3) - (D / BT)
420 IF N(V,3) > 0 THEN 440
430 POKE S + VL + 4, N(V,4) - 1
440 NEXT V
450 REM  **  START NEW NOTES
460 FOR V = 1 TO 3: VL = (V - 1) * 7
470 IF NN(V) = 0 THEN 490
480 POKE S + VL + 4, N(V,4)
490 NEXT V
500 REM  **  GET TIME FOR TEMPO
510 T1 = TIME
520 REM  **  FIND SHORTEST DURATION
530 D = 999
540 FOR V = 1 TO 3
550 IF N(V,3) <= 0 THEN 570
560 IF N(V,3) < D THEN D = N(V,3)
570 NEXT V
580 D = D * BT: GOTO 200
590 REM  **  STOP THE MUSIC
600 IF (TIME - T1) < D THEN 600
610 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
620 END
999 REM  ----------
1000 DATA 1,12,2,0, 2,0,1,9
```

(continued)

```
1001 DATA 2,9,1,0,  3,5,1,9
1002 DATA 1,11,1,9
1003 DATA 1,12,1,9
1004 DATA 1,12,2,0, 2,0,1,9
1005 DATA 2,10,1,0, 3,7,1,9
1006 DATA 1,11,1,9
1007 DATA 1,12,1,9
1008 DATA 1,17,2,0, 2,0,1,9
1009 DATA 2,9,1,9
1010 DATA 1,16,1,9
1011 DATA 1,17,1,9
1012 DATA 1,24,4,0, 2,0,1,9
1013 DATA 2,6,1,9
1014 DATA 2,15,1,9
1015 DATA 2,6,1,9
1016 DATA 1,22,2,0, 2,0,1,9
1017 DATA 2,7,1,9
1018 DATA 1,21,1,0, 2,9,1,9
1019 DATA 1,19,1,0, 2,10,1,9
1020 DATA 1,21,2,0, 2,12,1,9
1021 DATA 2,16,1,9
1022 DATA 1,19,2,0, 2,10,1,9
1023 DATA 2,16,1,9
1024 DATA 1,17,2,0, 2,0,1,9
1025 DATA 2,9,1,9
1026 DATA 1,16,1,0, 2,8,1,9
1027 DATA 1,14,1,0, 2,11,1,9
1028 DATA 1,16,2,0, 2,7,1,9
1029 DATA 2,10,1,9
1030 DATA 1,14,2,0, 2,4,1,9
1031 DATA 2,10,1,9
1032 DATA 1,12,2,0, 2,0,1,9
1033 DATA 2,9,1,0, 3,5,1,9
1034 DATA 1,11,1,9
1035 DATA 1,12,1,9
1036 DATA 1,12,2,0, 2,0,1,9
1037 DATA 2,10,1,0, 3,7,1,9
1038 DATA 1,11,1,9
1039 DATA 1,12,1,9
1040 DATA 1,17,2,0, 2,0,1,9
1041 DATA 2,9,1,9
1042 DATA 1,16,1,9
1043 DATA 1,17,1,9
1044 DATA 1,26,2,0, 2,0,1,9
```

```
1045 DATA 2,6,1,9
1046 DATA 1,24,2,0, 2,15,1,9
1047 DATA 2,6,1,9
1048 DATA 1,22,2,0, 2,0,1,9
1049 DATA 2,7,1,9
1050 DATA 1,18,1,0, 2,9,1,9
1051 DATA 1,19,1,0, 2,10,1,9
1052 DATA 1,21,2,0, 2,12,1,9
1053 DATA 2,16,1,9
1054 DATA 1,19,2,0, 2,10,1,9
1055 DATA 2,16,1,9
1056 DATA 1,17,4,0, 2,0,1,9
1057 DATA 2,9,1,9
1058 DATA 2,12,1,9
1059 DATA 2,9,1,9
1060 DATA 1,29,2,0, 2,9,2,0, 3,5,2,9
9999 DATA 0,0,0,-1
RUN
```

Figure 10.13 Melody in F, by Rubinstein, by synthesizer

Save this program on tape or disk so that later you can get it back for experimentation. You might also wish to use it as the basis for your own musical arrangements.

Filtering

A sound is a highly complex structure comprised of a fundamental frequency plus the harmonics, which are its multiples and submultiples. The more harmonics, the fuller the tone sounds. Also, a tone with many harmonics has more sound and is therefore louder. That's why, in Melody F, the guitar sounded louder than the flute, as though it were closer to the microphone; the guitar uses a pulse wave, which generates more harmonics than the triangular wave of the flute.

The process of stripping away certain frequency components of a sound, such as harmonics, is called filtering. As the name suggests, the sound passes through a device—a circuit, in this case—that stops certain frequencies and lets others pass. This alters the character of the tone.

Filtering is a selective process that allows you to pick cutoff points. A high-pass filter permits frequencies and their harmonics that are above the cutoff to pass and stops those below it. A low-pass filter has the opposite effect, allowing only sounds below the cutoff to pass. A bandpass filter passes only a selected frequency and no others, so if the cutoff is

set for the frequency that is playing, you hear that tone with no harmonics. These are the three filters of the SID chip. You can combine the low-pass and high-pass filters to produce the opposite effect of the band-pass in what is called *notch-reject filtering*, which will stop frequencies in a narrow band and pass all others. Using notch filtering you can hear all the harmonics of a frequency without hearing the fundamental tone. Filtering can thus produce some strange sounds. Inevitably, because the filter removes some of the sound components, a filtered tone has a lower volume than when unfiltered.

Make sure you've saved Melody in F, and then we'll conduct some experiments in filtering by building on Fig. 10.14.

```
NEW
100 REM  **  FILTERING EXPERIMENT KIT
110 WF = 65                          :REM  WAVEFORM
120 HF = 26: S = 54272
130 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
140 POKE S+1, HF                     :REM  TONE FREQ
150 POKE S+3, 8                      :REM  MED PULSE WIDTH
160 POKE S+6, (15 * 16)              :REM  S/R
170 POKE S+24, 15                    :REM  VOLUME
180 POKE S+4, WF                     :REM  TURN ON VOICE
190 FOR T=1 TO 2200: NEXT T          :REM  HOLD 3 SEC.
200 REM  ----------
999 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
RUN
```

Figure 10.14 Filtering demonstration

The initial run of this program produces a midrange tone using the fullest (medium) pulse width and lasts for 3 seconds. This is an unfiltered tone that will sound at the start of each of the following experiments as an audible point of reference.

In order to invoke filtering, it is necessary to set two switches in the SID registers and provide a cutoff frequency (refer to Fig. 10.3, bytes 21–24). The anatomies of these registers are further detailed by Fig. 10.15.

Use the chart in Fig. 10.15 as follows to turn on filtering:

1. Select the voice to be filtered by setting its corresponding bit in S+23. You can select more than one voice for filtering.
2. Select the filter mode by setting the mode bit in S+24, added to the volume level. You can select more than one filter mode.
3. Set the cutoff frequency by POKEing the high-frequency component ("HF") into S+22.

Address	54272+	Bits	Purpose
54295	23	0	Voice 1 filter: 1 = yes, 0 = no
		1	Voice 2 filter: 1 = yes, 0 = no
		2	Voice 3 filter: 1 = yes, 0 = no
		4–7	Filter resonance level (0–15)
54296	24	0–3	Volume control: 0–15 (loudest)
		4	Low-pass filter: 1 = on, 0 = off
		5	Bandpass filter: 1 = on, 0 = off
		6	High-pass filter: 1 = on, 0 = off
		7	Voice 3 output: 1 = off, 0 = on
54294	22	0–7	Filter cutoff frequency (high byte)

Figure 10.15 Filter control registers

The following experiments all utilize this procedure.

The first filter we'll listen to is the high-pass, which passes all frequency components above the cutoff. Set this up by adding Fig. 10.16 to the program.

```
210 REM  **  FILTERED PORTION
220 POKE S+23, 1                    :REM  FILTER VOICE 1
230 POKE S+24, (15 + 2^6)           :REM  HIGH-PASS
240 POKE S+22, HF                   :REM  CUTOFF
250 POKE S+4, WF                    :REM  TURN ON VOICE
260 FOR T = 1 TO 2200: NEXT T       :REM  HOLD 3 SEC
RUN
```

Figure 10.16 High-pass filter demonstration

The sound that follows the initial unfiltered tone is very similar, but softer. It cuts out the harmonics below the cutoff and passes only the fundamental tone frequency and its upper harmonics. You can reverse this and hear only the lower harmonics by changing line 230 to read

```
230  POKE S+24, (15 + S^4)
```

which selects the low-pass instead of the high-pass filter. Now the filtered tone is nearly inaudible, demonstrating that most of the content of a sound consists of its fundamental frequency and the upper harmonics.

This is an opportune point to digress for a moment on *resonance*, another characteristic of sound. Resonance is the relative loudness of low-frequency harmonics; the higher the resonance level, the richer the timbre.

Figure 10.15 shows that the upper nibble of S+23 controls the resonance introduced by the filter. In the low-pass demonstration, the resonance was set to 0, and as a result, the low-frequency tones scarcely sounded at all. We can set resonance to maximum (15) by changing line 220 to read

```
220  POKE S+23, 1 + (16 * 15)
```

What a difference it makes! The low-frequency harmonics are now amplified fully, so that the volume level is the same for both tones, but distinctly richer—more resonant—in the second.

Now let's hear the effect of bandpass filtering, in which only the fundamental frequency passes and the filter removes all the harmonic components. For that we have to turn off the resonance control and select switch 5 in S+24 with the following:

```
220  POKE S+23, 1
230  POKE S+24, (15 + S^5)
```

The result is a quiet yet harsh tone, unsoftened by harmonics. Using this filter and a sawtooth waveform, you could probably find a frequency that would literally shatter glass with its pure and penetrating tonal quality.

The last filter we'll listen to is the notch-reject setting, which is the opposite of bandpass. It lets the harmonics pass, but not the fundamental frequency. You get it with

```
230  POKE S+24, (15 + 2^6 + 2^4)
```

which switches on both the high- and low-pass filters. Notice how limp and dispirited the filtered tone becomes. That's because its driving force is absent, and you hear only the sound by-products.

It is not possible to have different filter settings for each voice, but it is possible to have each voice either filtered or unfiltered by setting the select bits in S+23. Thus you can set up the filter and then either have it influence or not influence each voice, and this gives a broad span of possibilities to your sound effects and music.

Play with this program for a while, trying it with different waveforms, resonances, frequencies, etc. It gives you a convenient laboratory in which to learn firsthand about the many interesting effects you can create with filters.

Reload Melody in F and we'll filter the guitar to tone it down. Three additional instructions accomplish this:

```
195  POKE S+23, 2^1 + 2^2
197  POKE S+24, 12 + 2^6
365  POKE S+22, N(V,2)
```

Line 195 selects filtering for Voices 2 and 3, which play the guitar notes, and 197 turns on the high-pass filter. Line 365 dynamically sets the cutoff of the filter to the frequency of the lowest note played, blocking lower harmonics. Run it and you'll hear that the guitar is more subdued, as an accompanying instrument ought to be, letting the melody dominate. The effect, though subtle, is noticeable.

Nonmusical sound effects

You can use the synthesizer to make all sorts of sounds besides music. The possibilities are limited only by the time you devote to experimentation. Here we will present a sampler of useful and interesting sound effects to give you some ideas and a library of noises to build into games and other action programs.

The first is a siren so realistic it'll make the family and the neighbors wonder where the fire is. To produce it, type Fig. 10.17.

```
NEW
100 REM  ** EMERGENCY!
110 S = 54272: FQ = 12000: WF = 33
120 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
130 POKE S+24, 15
140 POKE S+6, (15 * 16)
150 POKE S+4, WF
160 BF = FQ: TF = FQ * 2: SV = (TF - BF)/128
170 FOR F = BF TO TF STEP SV
180 GOSUB 300
190 NEXT F
200 GET X$: IF X$ <> "" THEN 999
210 FOR F = TF TO BF STEP -SV
220 GOSUB 300
230 NEXT F
240 GET X$: IF X$ = "" THEN 170
250 GOTO 999
300 REM  ** SIREN SOUND
310 HF = INT(F/256): LF = F - (HF * 256)
320 POKE S, LF: POKE S+1, HF
330 FOR T = 1 TO 20: NEXT T
340 RETURN
999 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
RUN
```

Figure 10.17 Siren

The sound sweeps up and down through one high-frequency octave. When it reaches the highest or lowest point, it stops if there has been a signal from the keyboard; otherwise, it goes through another half-cycle. The 128 steps specified in line 160 give it the sliding progression characteristic of sirens.

Oddly enough, we can use almost exactly the same instructions to produce a storm at sea. This entails changing the waveform to white noise and applying the bandpass filter in Fig. 10.18.

```
110 S = 54272: FQ = 12000: WF = 129
125 POKE S+23, 1 + (15 * 16)
130 POKE S+24, (15 + 2^5)
315 POKE S+22, HF
RUN
```

Figure 10.18 Blowin' up a nor'easter

It's not difficult, listening to the shrieking of the wind and the thunder of waves, to get a little seasick and start visualizing Hornblower gathering his cloak about him and holding down his tricorn cap as the spray blows and the deck rolls underfoot.

And from the ship at sea in a storm, it's only a short swim to the surf rolling in and breaking on the shore, a sound familiar to those of us who live by the sea. Simply remove lines 200–230 from the program, which deletes the downward portion of the rise-and-fall cycle and simulates the tumbling and shoreward rush of successive combers.

You can make a lot of other interesting and realistic sounds with the noise waveform. Figure 10.19 is a steam locomotive starting up and gathering speed. The acceleration occurs in line 210, where the delay factor R decreases by 10 with each chug. It stabilizes at 90 iterations per timing loop, giving the train a steady top speed.

Some strange things happen to sounds when you combine two waveforms using *ring modulation*, which produces the nonharmonic overtones of a bell or gong. If you POKE a frequency for Voice 3 but don't turn it on with a waveform, Oscillator 3 vibrates inaudibly at that frequency. By turning on Voice 1 or Voice 2 with a waveform value plus 4 (which sets bit 2 of the waveform register), the SID chip combines that voice's wave with the frequency of Oscillator 3 and you hear a ring-modulated tone. Figure 10.20 simulates a mantel clock striking midnight.

You can change the deliberate, dignified rate at which the clock chimes by giving R a different value in line 110—the higher the value, the slower it rings. Line 130 sets up the frequencies of the two voices. The trick is to use frequencies that have no easy common denominator so that

```
NEW
100 REM  **  CHOO-CHOO
110 S = 54272: R = 800
120 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
130 POKE S+1, 50
140 POKE S+5, (8 * 16) + 8
150 POKE S+24, (15 + 2^4)
160 POKE S+23, (15 * 16) + 1
170 POKE S+22, 128
180 POKE S+4, 129
190 FOR T = 1 TO R: NEXT T
200 POKE S+4, 128
210 R = R - 10: IF R < 90 THEN R = 90
220 GET X$: IF X$ = "" THEN 180
230 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
RUN
```

Figure 10.19 Old-time railroading

the two frequencies beat against each other. Change the Voice 3 frequency to 30 and you'll discover that much of the chime quality is lost; 90 is a multiple of 30 and the frequencies don't conflict. A change in either frequency greatly alters the character of the bell sound. Try different settings to hear the rich tonal possibilities of ring modulation.

```
NEW
100 REM  **  MIDNIGHT
110 S = 54272: R = 900
120 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
130 POKE S+1, 90: POKE S+15, 37
140 POKE S+5, 9
150 POKE S+24, 15
160 FOR C = 1 TO 12
170 POKE S+4, 17 + 4
180 FOR T = 1 TO R: NEXT T
190 POKE S+4, 16 + 4
200 NEXT C
210 FOR X = 0 TO 24: POKE S+X, 0: NEXT X
RUN
```

Figure 10.20 Chiming clock

Ordinarily the chiming of midnight signals the end of a day. In this case, it signals the end of our journey together through the Commodore 64's myriad sound and graphics capabilities. Lacking an infinite amount of time and space, we have not explored every nook and cranny, nor have we exploited every combination of sprites, high-resolution graphics, sound, color, and all the other tremendous, exciting potentials this machine possesses. Instead, we have laid a solid base of fundamentals upon which to build. The rest is up to you, and I wish you well.

APPENDIX A

SCREEN DISPLAY CODES

The following chart lists all of the characters built into the Commodore 64 character sets. It shows which numbers should be POKEd into screen memory (locations 1024–2023) to get a desired character. Also shown is which character corresponds to a number PEEKed from the screen.

Two character sets are available, but only one set at a time. This means that you cannot have characters from one set on the screen at the same time you have characters from the other set displayed. The sets are switched by holding down the SHIFT and C= keys simultaneously.

From BASIC, POKE 53272,21 will switch to uppercase mode and POKE 53272,23 switches to lowercase.

Any number on the chart may also be displayed in REVERSE. The reverse character code may be obtained by adding 128 to the values shown.

If you want to display a solid circle at location 1504, POKE the code for the circle (81) into location 1504: POKE 1504,81.

There is a corresponding memory location to control the color of each character displayed on the screen (locations 55296–56295). To change the color of the circle to yellow (color code 7) you would POKE the corresponding memory location (55776) with the character color: POKE 55776,7.

NOTE: The following POKEs display the same symbol in set 1 and 2: 1, 27–64, 91–93, 96–104, 106–121, 123–127.

SCREEN CODES

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	←		31	>		62
A	a	1	SPACE		32	?		63
B	b	2	!		33			64
C	c	3	"		34		A	65
D	d	4	#		35		B	66
E	e	5	$		36		C	67
F	f	6	%		37		D	68
G	g	7	&		38		E	69
H	h	8	'		39		F	70
I	i	9	(		40		G	71
J	j	10	)		41		H	72
K	k	11	*		42		I	73
L	l	12	+		43		J	74
M	m	13	,		44		K	75
N	n	14	–		45		L	76
O	o	15	.		46		M	77
P	p	16	/		47		N	78
Q	q	17	0		48		O	79
R	r	18	1		49		P	80
S	s	19	2		50		Q	81
T	t	20	3		51		R	82
U	u	21	4		52		S	83
V	v	22	5		53		T	84
W	w	23	6		54		U	85
X	x	24	7		55		V	86
Y	y	25	8		56		W	87
Z	z	26	9		57		X	88
[		27	:		58		Y	89
£		28	;		59		Z	90
]		29	<		60			91
↑		30	=		61			92

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
		93			105			117
		94			106			118
		95			107			119
SPACE		96			108			120
		97			109			121
		98			110			122
		99			111			123
		100			112			124
		101			113			125
		102			114			126
		103			115			127
		104			116			

Codes from 128-255 are reversed images of codes 0-127.

Appendix 1 Codes from 128-255 are reversed images of codes 0-127.

APPENDIX B

ASCII AND CHR$ CODES

This appendix shows you what characters will appear if you PRINT CHR$(X), for all possible values of X. It will also show the values obtained by typing PRINT ASC("x"), where x is any character you can type. This is useful in evaluating the character received in a GET statement, converting upper-/lowercase, and printing character-based commands (like switch to upper-/lowercase) that could not be enclosed in quotes.

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0		11		22	!	33
	1		12		23	"	34
	2	RETURN	13		24	#	35
	3	SWITCH TO LOWER CASE	14		25	$	36
	4		15		26	%	37
	5		16		27	&	38
	6		17		28	'	39
	7		18		29	(	40
DISABLES	8		19		30	)	41
ENABLES	9		20		31	*	42
	10		21	SPACE	32	+	43

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
,	44	L	76		108	f8	140
-	45	M	77		109	SHIFT RETURN	141
.	46	N	78		110	SWITCH TO UPPER CASE	142
/	47	O	79		111		143
0	48	P	80		112	BLK	144
1	49	Q	81		113	CRSR	145
2	50	R	82		114	RVS OFF	146
3	51	S	83		115	CLR HOME	147
4	52	T	84		116	INST DEL	148
5	53	U	85		117	Brown	149
6	54	V	86		118	Lt. Red	150
7	55	W	87		119	Grey 1	151
8	56	X	88		120	Grey 2	152
9	57	Y	89		121	Lt. Green	153
:	58	Z	90		122	Lt. Blue	154
;	59	[	91		123	Grey 3	155
<	60	£	92		124	PUR	156
=	61	]	93		125	CRSR	157
>	62	↑	94		126	YEL	158
?	63	←	95		127	CYN	159
@	64		96		128	SPACE	160
A	65		97	Orange	129		161
B	66		98		130		162
C	67		99		131		163
D	68		100		132		164
E	69		101	f1	133		165
F	70		102	f3	134		166
G	71		103	f5	135		167
H	72		104	f7	136		168
I	73		105	f2	137		169
J	74		106	f4	138		170
K	75		107	f6	139		171

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	172		177		182		187
	173		178		183		188
	174		179		184		189
	175		180		185		190
	176		181		186		191

Appendix B1

CODES	192-223	SAME AS	96-127
CODES	224-254	SAME AS	160-190
CODE	255	SAME AS	126

APPENDIX C

ASSEMBLY-LANGUAGE SOURCE LISTINGS

The following are source listings for the machine-language subroutines presented in this book. The machine language code is read from DATA statements and POKEd into specific memory locations by BASIC. Each routine is then called from the BASIC program by issuing a SYS instruction that refers to the starting address of the routine. Some routines require that the program first POKE variable information into given locations where the machine language expects to find it, or that other conditions be established prior to the call. These prerequisites are described where appropriate.

Routine CLRHRG

Purpose: Clear high-resolution graphics screen memory between 8192 and 16191 by filling with binary 0s.

Remarks: Begins at location 49152, occupies 19 bytes (through 49170). No setup is required after POKEing into memory and before using. Call with SYS 49152.

```
Listing:        Assembly language:          Decimal:
                ------------------          --------
                LDA     #$00                169, 0
                LDY     #$00                160, 0
                LDX     #$20                162, 32
        LOOP:   STA     $2000,Y             153, 0, 32
                INY                         200
                BNE     LOOP                208, 250
                INC     $C008               238, 8, 192
                DEX                         202
                BNE     LOOP                208, 244
                RTS                         96
```

Routine CLRCM

Purpose: Loads color memory for HRG (1024–2023) with the color byte specified in address 49172.

Setup: POKE desired color combination into 49172, then call with SYS 49171.

Remarks: Begins at 49171, occupies 34 bytes (through 49204). If no value is POKEd to 49172, will set up a black screen with white foreground (first time through) or the last combination POKEd in 49172 (subsequent times through).

```
Listing:        Assembly language:          Decimal:
                ------------------          --------
                LDA     #2                  169, 2
                STA     2                   133, 2
        ENT:    LDA     #4                  169, 4
                STA     $97                 133, 151
                LDA     #$E7                169, 231
                STA     $A3                 133, 163
                LDA     $FC                 165, 252
                STA     $FE                 133, 254
                ADC     #3                  105, 3
                STA     $A4                 133, 164
                LDY     #0                  160, 0
                LDA     ($FB),Y             177, 251
                STA     255                 133, 255
                INY                         200
```

```
LP1:    LDA     ($FB),Y         177, 251
        DEY                     136
        STA     ($FD),Y         145, 253
        INY                     200
        BNE     LP2             208, 2
        INC     $FE             230, 254
LP2:    INY                     200
        BNE     LP1             208, 243
        INC     $FC             230, 252
        DEC     $97             198, 151
        BNE     LP1             208, 237
        LDA     255             165, 255
        STA     ($A3),Y         145, 163
        DEC     2               198, 2
        BEQ     RET             240, 5
        LDA     #$D8            169, 216
        STA     $FC             133, 252
        BNE     ENT             208, 198
RET:    RTS                     96
```

Routine MOBIUS

Purpose: Fast spiral scrolling from left to right and upwards. The byte in the upper-left corner wraps to the lower-right corner. Scrolls character-mode screen memory and color memory.

Remarks: Begins at location 49205, occupies 61 bytes (through 49265). Required setup is to POKE the screen-memory page number (SM/256) into address 252, then call with SYS 49205.

```
NEW
10 A$ = "MULTICOLOR SPRITE MAKER"
20 PRINT CHR$(147): PRINT A$
30 PRINT: PRINT
40 INPUT "NUMBER FOR COLOR 1"; C1
50 INPUT "NUMBER FOR COLOR 2"; C2
60 INPUT "NUMBER FOR COLOR 3"; C3
70 POKE 53285, C1: POKE 53286, C3
80 POKE 53287, C2
100 REM  ××  WORK GRID
110 POKE 53269, 0: PRINT CHR$(147)
120 FOR Y = 1 TO 21
```

(continued)

```
130 FOR X = 0 TO 23 STEP 2: PRINT ". ";: NEXT X
140 PRINT: NEXT Y
150 PRINT CHR$(19)
160 PRINT TAB(28) "STEP 1:"
170 PRINT TAB(28) "RUN 200 NEXT"
180 FOR Y = 3 TO 20: PRINT: NEXT Y: END
190 REM ----------
200 REM  ¥¥  STEP 2 BUILDS SPRITE
210 SL = 14336: SP = SL / 64: PRINT CHR$(19)
220 PRINT TAB(28) "STEP 2:"
230 PRINT TAB(28) "K IF OK     "    [5 spaces]
240 PRINT TAB(28) "N IF NOT": PRINT TAB(28);
250 FOR P = 0 TO 63: POKE SL+P, 0: NEXT P
255 POKE 53276, 1
260 POKE 2040, SP: POKE 53287, 1
270 POKE 53248, 255: POKE 53249, 200
280 POKE 53269, 1: E = 256: B = 0
290 FOR Y = 1 TO 21
300 FOR X = 0 TO 23 STEP 2: P = 1024 + X + (Y ¥ 40)
310 E = E / 4: V = PEEK(P)
320 IF V = 46 THEN 340
325 V = V AND 3
330 POKE SL + B, PEEK(SL + B) OR (E ¥ V)
340 IF E = 1 THEN E = 256: B = B + 1
350 NEXT X: NEXT Y
360 INPUT X$
370 IF X$ = "K" THEN 410
380 PRINT CHR$(19)
390 FOR X = 1 TO 5: PRINT TAB(28) " [12 spaces] "
400 NEXT X: GOTO 160
410 REM  ¥¥  PRINT VALUES FOR SPRITE
420 POKE 53269, 0: PRINT CHR$(147)
430 FOR Y = 0 TO 20
440 FOR X = 0 TO 23: P = X + (Y ¥ 40)
450 PRINT PEEK(SL + P),
460 NEXT X
470 PRINT: NEXT Y
RUN
```

APPENDIX D

LISTINGS OF THE SKELETON PROGRAMS

This appendix contains listings of the major "skeletons" developed in the book as aids to assist you in writing your own programs. The purpose of the skeletons is to relieve you of as many technical burdens as possible so that you can concentrate on the particulars of the problem at hand. Thus, each skeleton comprises an environment in which the essential services are provided by means of subroutines. Instructions for using these subroutines are, to a great extent, the contents of the chapters where the skeletons are presented, and are not included here.

Each skeleton has been structured so that you can start your program at line 110. Lines preceding 100 do the setup needed for establishing the machine context of the mode of operation. Lines 1000 and up furnish services through subroutines that you call from your program. Line 999 is a GOTO that protects your programs from inadvertently wandering into the subroutine area. Your program, then, can occupy all the line numbers from 101 through 998.

The listings as shown here represent the skeletons at the stage of full development, that is, as they are at the ends of the chapters in which they are presented. Fields that you may optionally wish to alter or fill in are indicated in the listings with lowercase descriptions of the item that goes there (for an example, see line 35 in the HRG skeleton). Machine-language subroutines given by DATA statements at the tops of the skeletons are further described in Appendix C.

The HRG Skeleton

This skeleton is presented in Chapter 5. It provides a programming environment for standard bit-mapped high-resolution graphics. The listing is annotated.

```
1 REM  **  HRG SKELETON
5 FOR A = 49152 TO 49204
6 READ B: POKE A, B: NEXT A
7 DATA 169, 0, 160, 0, 162, 32, 153, 0, 32
8 DATA 200, 208, 250, 238, 8, 192
9 DATA 202, 208, 244, 96
10 DATA 169, 16, 160, 232, 162, 4, 141, 0, 4
11 DATA 238, 26, 192, 208, 3, 238, 27, 192
12 DATA 136, 208, 242, 202, 208, 239, 169, 0
13 DATA 141, 26, 192, 169, 4, 141, 27, 192, 96
19 POKE 53265, PEEK(53265) OR 32
20 POKE 53272, PEEK(53272) OR 8
30 SYS 49152
35 POKE 49172, (note 1): POKE 53280, (border color)
40 SYS 49171
50 GOTO 100
60 REM  **  SET BIT FOR XY COORDINATES
65 IF X < 0 OR X > 319 THEN 90
70 IF Y < 0 OR Y > 199 THEN 90
75 BA = (INT(Y/8)*320) + (INT(X/8)*8) + (Y AND 7)
   + 8192  (note 2)
80 BP = 7 - (X AND 7)
85 POKE BA, (PEEK(BA) OR 2^BP)
90 RETURN
100 REM  **  PROGRAM STARTS HERE

        (your program occupies lines 101 - 998)

999 GOTO 9960
1000 REM  **  GRID CONVERSION
1010 X = INT(XF + (H / HD) + .5)
1020 Y = INT(199 - YF - (V / VD) + .5)
1030 GOSUB 60
1040 RETURN


(you can insert your own additional subroutines in the
        lines between 1050 and 8499)
```

```
8500 REM  **  CHANGE FOREGROUND
8510 CB = 1024 + (Y * 40) + X     (note 2)
8520 POKE CB, (PEEK(CB) AND 15) + (CN * 16)
8530 RETURN
8540 REM ----------
8600 REM  **  CHANGE BACKGROUND
8610 CB = 1024 + (Y * 40) + X     (note 2)
8620 POKE CB, (PEEK(CB) AND 240) + CN
8630 RETURN
8640 REM ----------
9000 REM  **  PAINT ONE LINE
9010 CX = SX
9020 IF SX <= EX THEN 9040
9030 SX = EX: EX = CX: CX = SX
9040 REM  **  START OF LINE
9050 X = CX: GOSUB 60
9060 IF BP > (EX - SX + 1) THEN 9180
9070 PV = 2^(BP + 1) - 1: POKE BA, PEEK(BA) OR PV
9080 CX = CX + BP + 1: BA = BA + 8
9090 REM  **  WHOLE BYTE
9100 BR = EX - CX + 1
9110 IF BR < 8 THEN 9140
9120 POKE BA, 255
9130 BA = BA + 8: CX = CX + 8: GOTO 9090
9140 REM  **  END OF LINE
9145 IF BR < 0 THEN BR = 0
9150 PV = 256 - 2^(8 - BR)
9160 POKE BA, PEEK(BA) OR PV
9170 RETURN
9180 REM  **  BIT PLOT
9190 FOR X = SX TO EX: GOSUB 60
9200 NEXT X: RETURN
9210 REM ----------
9960 GET X$: IF X$ = "" THEN 9960
9970 POKE 53265, PEEK(53265) AND 223
9980 POKE 53272, PEEK(53272) AND 247
9990 PRINT CHR$(147): END
```

Note 1: Foreground color times 16 plus background color.

Note 2: When working with multiple screens, add the base address of the Bank and change the literal number (1024 or 8192 in these cases) to the starting address of the screen memory unit in use.

The MHRG Skeleton

This skeleton is presented in the second part of Chapter 6. It provides a programming environment for pixel-mapped multicolor high-resolution graphics. The listing is annotated.

```
1 REM  **  MHRG SKELETON
5 FOR A = 49152 TO 49204
6 READ B: POKE A, B: NEXT A
7 DATA 169, 0, 160, 0, 162, 32, 153, 0, 32
8 DATA 200, 208, 250, 238, 8, 192
9 DATA 202, 208, 244, 96
10 DATA 169, 16,  160, 232, 162, 4, 141, 0, 4
11 DATA 238, 26, 192, 208, 3, 238, 27, 192
12 DATA 136, 208, 242, 202, 208, 239, 169, 0
13 DATA 141, 26, 192, 169, 4, 141, 27, 192, 96
18 POKE 53270, PEEK(53270) OR 16
19 POKE 53265, PEEK(53265) OR 32
20 POKE 53272, PEEK(53272) OR 8
30 SYS 49152
35 POKE 49172, (note 1): POKE 53280, (border color)
40 SYS 49171
50 GOTO 100
60 REM  **  COLOR PIXEL FOR XY COORDINATES
65 IF X < 0 OR X > 159 THEN 90
70 IF Y < 0 OR Y > 199 THEN 90
75 BA=(INT(Y/8)*320)+(INT((X*2)/8)*8)+(Y AND 7)+8192
   (note 2)
80 BP = 3 - (X AND 3): PM = 4^BP: PP = 255 - (PM * 3)
85 POKE BA, ((PEEK(BA) AND PP) OR (CN * PM))
90 RETURN
100 REM  **  PROGRAM STARTS HERE

        (Your program occupies lines 110 - 998)

999 GOTO 9960
1000 REM  **  GRID CONVERSION
1010 X = INT(XF + (H / HD) + .5)
1020 Y = INT(199 - YF - (V / VD) + .5)
1030 GOSUB 60
1040 RETURN

  (Your additional subroutines can occupy lines 1050
          through 8499)
```

```
8500 REM  **  CHANGE FOREGROUND
8510 CB = 1024 + (Y * 40) + X    (note 2)
8520 POKE CB, (PEEK(CB) AND 15) + (CN * 16)
8530 RETURN
8540 REM  ----------
8600 REM  **  CHANGE BACKGROUND
8610 CB = 1024 + (Y * 40) + X    (note 2)
8620 POKE CB, (PEEK(CB) AND 240) + CN
8630 RETURN
8640 REM  ----------
9000 REM  **  PAINT COLORED LINE
9010 CX = SX
9020 IF SX <= EX THEN 9040
9030 SX = EX: EX = CX: CX = SX
9040 REM  **  START OF LINE
9050 X = CX: GOSUB 60
9060 IF BP > (EX - SX + 1) THEN 9160
9070 IF BP = 3 THEN 9100
9075 IF BP = 0 THEN CX = CX + 1: BA = BA + 8: GOTO 9100
9080 BR = BP: FOR PB = BR TO 0 STEP-1: X = CX: GOSUB 60
9090 CX = CX + 1: NEXT PB: BA = BA + 8
9100 REM  **  WHOLE BYTE
9110 PB=0: FOR PV = 3 TO 0 STEP-1: PB = PB
     + (CN * 4^PV): NEXT PV
9120 BR = EX - CX + 1: IF BR < 4 THEN 9140
9130 POKE BA, PB: BA = BA + 8: CX = CX + 4: GOTO 9120
9140 REM  **  END OF LINE
9150 IF BR <= 0 THEN 9180
9160 REM  **  BIT PLOT
9170 FOR X = CX TO EX: GOSUB 60: NEXT X
9180 RETURN
9190 REM  ----------
9960 GET X$: IF X$ = "" THEN 9960
9970 POKE 53265, PEEK(53265) AND 223
9975 POKE 53270, PEEK(53270) AND 239
9980 POKE 53272, PEEK(53272) AND 247
9990 PRINT CHR$(147): END
```

Note 1: Foreground color times 16 plus background color.

Note 2: When working with multiple screens, add the base address of the Bank and change the literal number (8192 or 1024 in these cases) to the starting address of the screen memory unit in use.

The Multicolor Spritemaker

This program is an enhancement of the standard (monochrome) spritemaker shown in Fig. 8.13. It permits you to build multicolor sprites interactively, seeing the results as you go. The operating instructions for the program are found under the discussion of multicolor sprites in Chapter 8. This is a complete stand-alone program.

```
Listing:        Assembly language:          Decimal:
                -----------------           --------
                LDA     #$10                169, 16
                LDY     #$E8                160, 232
                LDX     #$04                162, 4
        LOOP:   STA     $0400               141, 0, 4
                INC     $C01A               238, 26, 192
                BNE     EOL                 208, 3
                INC     $C01B               238, 27, 192
        EOL:    DEY                         136
                BNE     LOOP                208, 242
                DEX                         202
                BNE     LOOP                208, 239
                LDA     #$00                169, 0
                STA     $C01A               141, 26, 192
                LDA     #$04                169, 4
                STA     $C01B               141, 27, 192
                RTS                         96
```

INDEX

COMPUTER • Z5490 • $9.95
CANADA • $12.50

THE EASY-TO-USE KEYS TO THE WORLD OF COMPUTER CREATIVITY

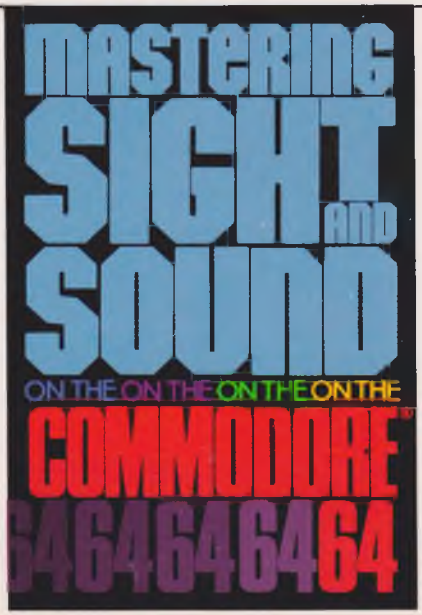
The amazingly high-performance, low-priced Commodore 64 home computer is extraordinary in capabilities that allow you to produce your own graphic animations, color compositions, and musical creations. Now Kent Porter, who has written some of the best—and most intelligible—guides to computer uses, offers both computer veterans and novices:

An overview of the Commodore 64's full range of powers, and the special features that deal with sight and sound.

The full spectrum of visual potential at your fingertips, from creating game graphics to producing works of computer art.

The exciting secrets of synthesizing sounds, whether as special effects or as full-fledged musical compositions.

Complete with computer codes, program illustrations, and a host of other uniquely helpful features, this reader-friendly book tells you everything you need to know to get the most out of your Commodore 64 home computer.



0-452-25490-6