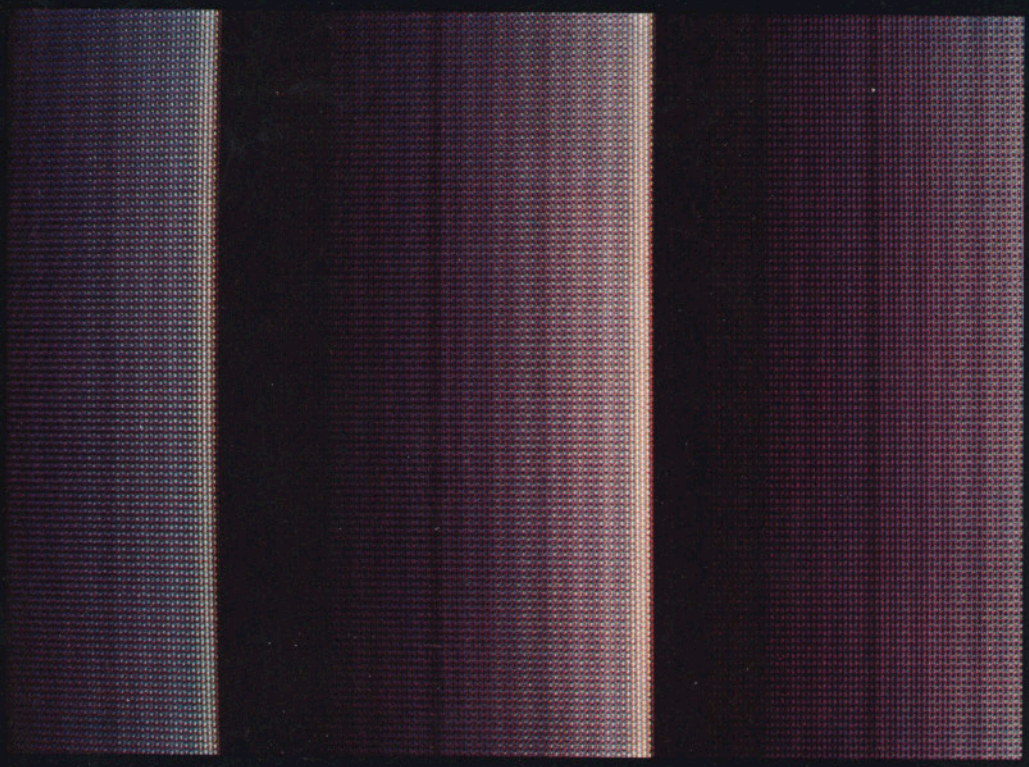
Computer Literacy Skills

GRAPHICS AND ANIMATION ON THE ATARI

800, 400, 1200XL, 800XL, and 600XL



CHRISTOPHER LAMPTON

GRAPHICS AND ANIMATION ON THE ATARI

800, 400, 1200XL, 800XL, and 600XL

BY CHRISTOPHER LAMPTON

A COMPUTER LITERACY SKILLS BOOK

The words *computer graphics* and *animation* conjure up magical images of arcade games, bright colors, flashing lights, and wonderful shapes.

This book shows you how to create graphics on the most popular Atari computers, which excel at producing color graphics. Some knowledge of the BASIC language is needed, but you needn't be an expert programmer.

You'll learn how to create graphics using text and bitmap modes, as well as using sophisticated techniques in which you will create your own character sets. Step-by-step programming instructions are blended with careful explanations of the internal workings and architecture of the computer.

Numerous activities give you a chance to try nearly every technique presented. Suggested projects are given at the end of most chapters, and a comprehensive program listing will enable you to easily create your own character set.

So why depend on commercial programs to make your life more graphic and animated? Let Chris Lampton help you learn how to create your own games, designs, and charts.

GRAPHICS AND ANIMATION ON THE ATARI

800, 400, 1200XL, 800XL, and 600XL

by Christopher Lampton

A Computer Literacy Skills Book
FRANKLIN WATTS 1986
New York London Toronto Sydney

Original designs created by Grant Geyer,
Marshall Geyer, and Jonathan Halpern,
who are students in Staten Island, New York.

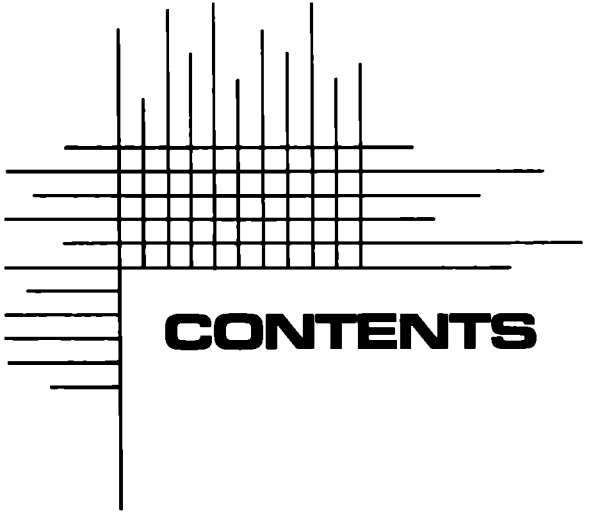
Photographs courtesy of Ira Schulman: vi, 4, 8, 11,
12, 15, 26, 27, 35, 37, 52, 57, 78, 81, 96, 99, 108;
Atari Inc.: p. 3 (top); Electronic Arts: p. 3 (bottom);
Atari Inc. and Star Wars: p. 59.

Library of Congress Cataloging in Publication Data

Lampton, Christopher.
Graphics and animation on the Atari.

(A Computer literacy skills book)
Bibliography: p.
Includes index.
Summary: Instructions for creating simple and
advanced graphics using BASIC on the Atari computer.
Includes suggestions for projects.
1. Atari computer—Programming. 2. Computer graphics.
3. BASIC (Computer program language) [1. Atari computer
—Programming. 2. Computer graphics. 3. BASIC (Computer
program language) 4. Programming languages (Computers)
5. Programming (Computers)] I. Title. II. Series.
QA76.8.A82L36 1986 006.6'765 85-26445
ISBN 0-531-10144-4

Printed in the United States of America
6 5 4 3 2 1



CONTENTS

Original designs created by Grant Geyer,
Marshall Geyer, and Jonathan Halpern,
who are students in Staten Island, New York.

Photographs courtesy of Ira Schulman: vi, 4, 8, 11,
12, 15, 26, 27, 35, 37, 52, 57, 78, 81, 96, 99, 108;
Atari Inc.: p. 3 (top); Electronic Arts: p. 3 (bottom);
Atari Inc. and Star Wars: p. 59.

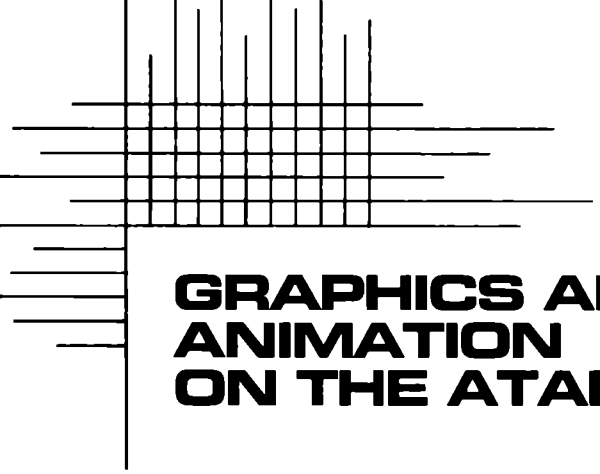
Library of Congress Cataloging in Publication Data

Lampton, Christopher.
Graphics and animation on the Atari.

(A Computer literacy skills book)
Bibliography: p.
Includes index.
Summary: Instructions for creating simple and
advanced graphics using BASIC on the Atari computer.
Includes suggestions for projects.
1. Atari computer—Programming. 2. Computer graphics.
3. BASIC (Computer program language) [1. Atari computer
—Programming. 2. Computer graphics. 3. BASIC (Computer
program language) 4. Programming languages (Computers)
5. Programming (Computers)] I. Title. II. Series.
QA76.8.A82L36 1986 006.6'765 85-26445
ISBN 0-531-10144-4

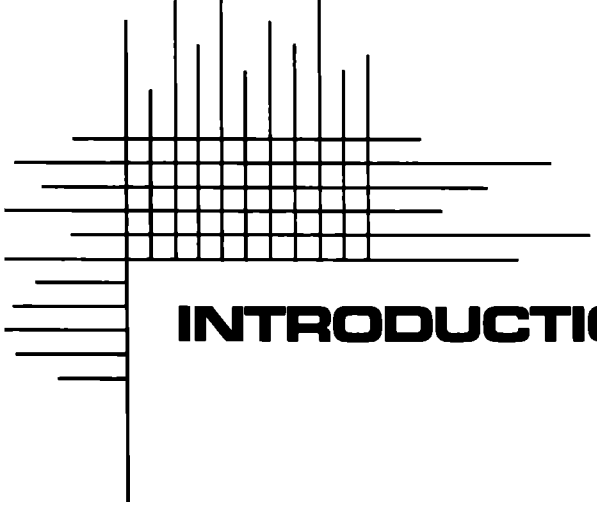
Printed in the United States of America
6 5 4 3 2 1

The Publisher wishes to acknowledge the assistance of Marshall Geyer, Grant Geyer, and Jonathan Halpern in the preparation of this book.



GRAPHICS AND ANIMATION ON THE ATARI

ATARI



INTRODUCTION

Before words, there were pictures. Primitive men and women traced beautiful murals on the walls of caves. Children too young to speak scrawl crayon drawings on paper or any available surface.

There is something in the human mind that responds to pictures as it can never respond to words. Pictures carry information more efficiently and intuitively (if somewhat less precisely) than words. A picture is worth a thousand words. Every picture tells a story. Let me paint you a picture. . . .

The computer, that most modern of information-processing gadgets, can process pictures as efficiently as it processes words and numbers, and in much the same fashion. Through computer programming, the art of writing instructions that can be executed by a computer, we can "paint" pictures on the computer's video display.

Pictures on the video display of a computer are called graphics. This book is about the creation of graphics on the Atari microcomputers—the Atari 800, 400, 1200XL, 800XL and 600XL computers—using the programming language called Atari BASIC. It is assumed in the pages that follow that you know something about programming in BASIC, but not necessarily in Atari BASIC. No previous knowledge of computer graphics is required.

[2]

HARDWARE AND SOFTWARE

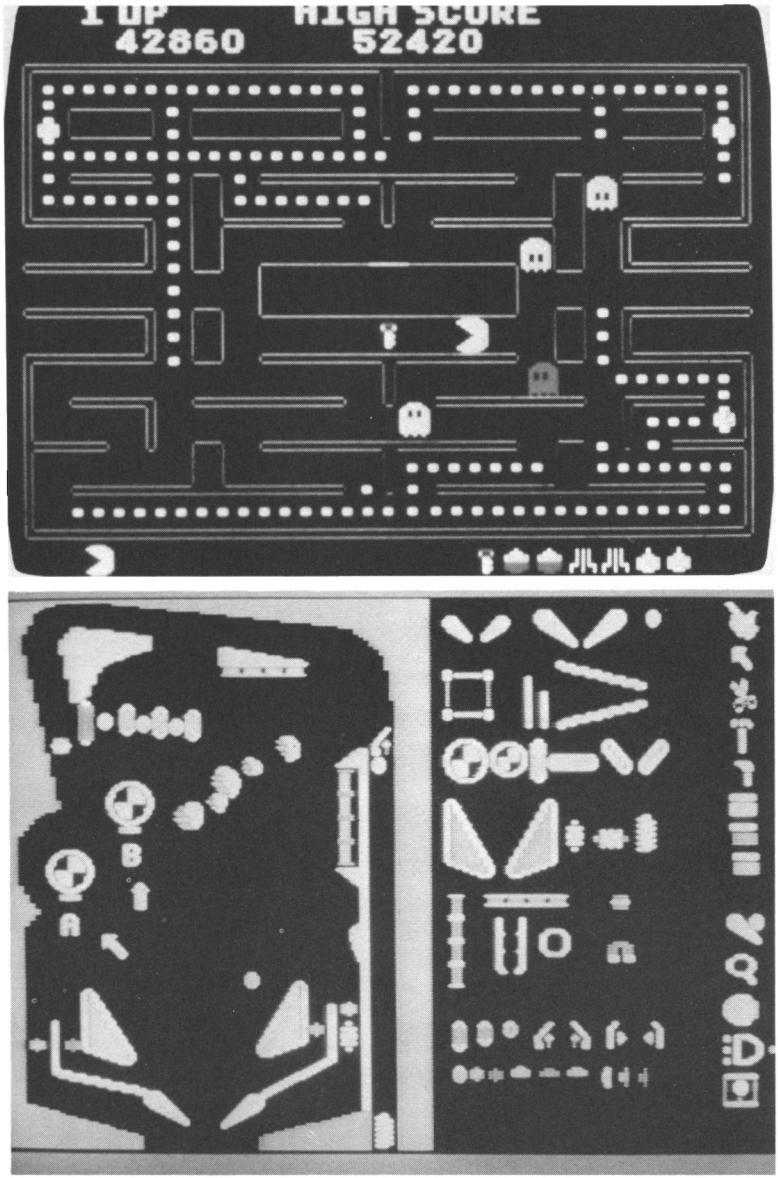
A computer is a piece of *hardware*, that is, a physical unit of machinery. The instructions that we give to the computer are *software*, or programs. The distinction between hardware and software—that is, between the machine and the instructions that we give to the machine—may seem obvious now, but later it may become a little fuzzy.

For instance, graphics are a function of hardware, but we control them through software. When a computer is built, it is given certain capabilities, among them (in most instances) the capability of producing graphic images on a video display. Though we can take advantage of these capabilities in our programs, even the cleverest of programmers cannot stretch these capabilities farther than the builders of the machine intended them to go, without first modifying the hardware (not a trivial task). If a computer has the capacity for producing only low-resolution, black and white images, we cannot write programs that will produce high-resolution, color ones.

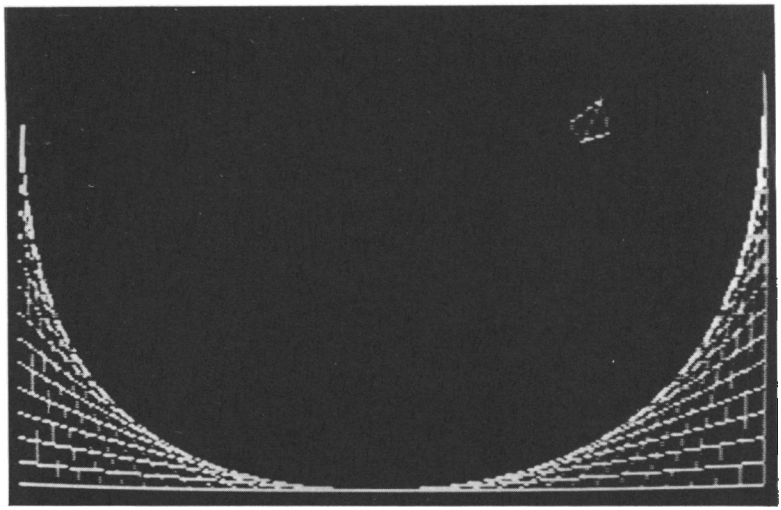
Fortunately, the creators of the Atari chose to bestow on it some fairly sophisticated graphics capabilities, including (but not limited to) the ability to produce high-resolution color images. The key to these capabilities are a pair of electronic circuits—microchips, actually—called Antic and GTIA. (On Ataris manufactured prior to 1982, the second of these chips is called CTIA.) Like miniature computers in their own right, these chips analyze the graphics instructions provided by the programmer and use them to assemble a video signal that can be sent to the video display, producing a picture.

As a potential Atari graphics programmer, you might wonder how we give instructions to Antic and GTIA. The answer depends on what kind of graphics effect you want to produce, and how much work you're willing to put into producing it. It is possible to "talk" directly to the Antic and GTIA chips with your programs, telling them exactly what you want them to do; much later in this book we'll study techniques for doing just that. In many instances, however, there will be an easier way.

Every Atari computer leaves the factory with a built-in program called the operating system, or OS for short. This program is actually made up of a number of smaller programs, or *routines*, each concerned with input and output, that is, the movement of data into and out of the computer. Because graphics are a form of output—the output of pictures to the video display—the OS contains a number of routines for producing graphics. Thus, when we wish to create graphics in our programs we don't have to write out the long series of instructions necessary to persuade Antic or GTIA to produce the desired effect; we simply tell the OS what



Top: PacMan, *the video game that started the whole graphics craze.* Bottom: *Electronic Arts'* Pinball Construction Set, *a "sophisticated recreation." Both products make good use of the Atari's graphics capabilities.*



Original graphics design created on the Atari

we want to do and it in turn relays the proper instructions to the Atari's graphic chips. (Actually, because most of our programs will be written in BASIC, we will give our instructions to another program called the BASIC interpreter, which in turn will pass the instructions to the OS, which will then relay the instructions directly to the appropriate chips. A jury-rigged process—but it works!)

Alas, there is a limit to how much help the Atari OS will give us. Though it contains some fairly sophisticated graphics routines, Antic and GTIA can perform some tricks that even the operating system doesn't know about. These include some of the most advanced graphics techniques that the Atari offers us as programmers. Thus, if we wish to venture out into this territory, we are on our own. We must talk directly to the graphics chips, explaining to them each detail of the operations we want them to perform.

Chapters one through four discuss the relatively "easy" operations that we can ask the operating system to perform for us. The next three chapters explain some of the more complex graphics techniques that we can use if we directly manipulate the Atari hardware. Although some of the advanced techniques may seem intimidating at first, they are not really so difficult once you familiarize yourself with a few essential concepts. A knowledge of these concepts will allow you to produce some spectacular and effective programs.

1 TEXT MODES

There is a time-honored art known as mosaic, where the artist creates pictures out of tiny, colored bits of material such as marble and glass. Though the individual bits of material may be shapeless and uninteresting—in fact, they may be shards thrown away by someone else—a skilled artist can fit them together to form elaborate and beautiful images.

Computer graphics are a kind of mosaic. The material out of which computer images are formed is neither marble nor glass; it is the *pixel*, short for *pictorial element*. A pixel is simply a point or rectangle of light on the video display of the computer. As programmers, we can control (within certain limitations) the color, position, and even size of the pixels that appear on the display of our computer. From these pixels we can fashion pictures, in much the same way that the artist builds a mosaic.

That's really all there is to graphics programming—putting the proper pixels in the proper positions on the video display. But your lesson in graphics programming has only just begun. Now you must learn how to tell the computer where to place pixels on the video display, a subject that will fill the rest of this book.

THE GRAPHICS MODES

The Atari is a versatile graphics computer. It offers us sixteen different *graphics modes*. In each of these modes, there are small

(and sometimes not so small) differences in the way we design graphic images. The mode that you choose for your graphics will depend on the specific kind of graphics you want to produce. In some instances, we can even use more than one graphics mode at the same time. (Actually, the number of graphics modes available depends on when your Atari was built. If it was built before 1982—that is, if it uses the CTIA chip instead of the GTIA chip—it will have fewer graphics modes available. We will discuss this problem later in the book, as we examine these "missing modes.")

In Atari BASIC, we select an Atari graphics mode with the statement GRAPHICS. We use this statement like this:

```
GRAPHICS mode number
```

where the mode number is a number between 0 and 63, indicating the graphics mode we wish to use. (Although there are only sixteen graphics modes, we can use larger numbers to indicate variations on these basic modes.) The mode number may be written as a *numeric expression*, that is, as a computation (such as 4 + 6) that evaluates to the proper mode number, or as a BASIC variable (such as A) equal to the proper mode number.

Roughly speaking, we can divide the Atari graphics modes into two categories: text modes and bitmap modes. In a bitmap mode, we can control the position and color of every pixel on the screen, as a mosaic artist controls the position of every fragment of glass or marble in the image that he or she is creating. In a text mode, we can control only the position and color of predefined images called *characters*, as though we were building a larger mosaic out of smaller mosaics.

When you turn on your Atari computer and enter Atari BASIC, you are already in graphics mode 0. This is a text mode; that is, it uses predefined characters. Some of these characters are designed in the images of the letters of the alphabet, which is why we can type words and sentences on the screen in graphics mode 0. Every time we press a key with a letter of the alphabet (or a numeral or a punctuation mark) depicted on it, the Atari's operating system tells the GTIA chip to output a picture of that character to the video display. Thus, our Atari computer can serve as a kind of electronic typewriter, though the medium on which it displays the typed text is a video monitor or television set rather than a piece of paper.

In graphics mode 0, we can display up to 960 characters on the screen at one time. These characters are organized into twenty-four rows of forty characters apiece. That is, we can have no more than forty characters across and twenty-four characters up and down.

Though text modes are, as the name implies, intended for the display of text, we can also use a text mode for creating graphics. In fact, some of the most interesting graphics displays in this book will utilize the Atari's text modes.

EXPLORING THE KEYBOARD

If you have an Atari computer handy, pause now to turn the computer on and prepare it for programming in BASIC, if you have not already done so. If you have an Atari 400 or 800, without the letters XL following the name, you must insert the BASIC programming cartridge before the computer is ready to be programmed; if you have an XL Atari, you need only turn the computer on.

The BASIC interpreter—the computer program that reads our BASIC instructions and translates them into a form that the computer can understand—will announce its presence by printing the word READY on the screen. Below this word will appear a nonblinking cursor. You may begin to type instructions.

Before you do so, however, play with the keyboard a little and watch what happens on the screen as you do so. Occasionally you will receive some odd messages from the BASIC interpreter, but you can ignore these. The interpreter simply thinks that you're trying to give it instructions, and it is having difficulty making sense of what you've typed.

As noted earlier, pressing a key with a letter of the alphabet or a numeral or a punctuation mark on it will cause that character to appear on the display. In addition, certain keys have special uses, often more than one. The key marked SHIFT, for instance, when pressed along with a second key, will often change the meaning of the second key. The key marked TAB causes the cursor to jump forward several spaces on the screen. The key marked CAPS allows you to type both upper- and lowercase letters on the screen. On XL model Ataris, pressing the CAPS key will return the Atari keyboard to its "normal" state, where only uppercase letters can be typed. On earlier-model Ataris, pressing the SHIFT and CAPS keys simultaneously will produce the same result. The key marked RETURN will produce a carriage return, causing the cursor to drop from its current line to the left-hand margin of the following line.

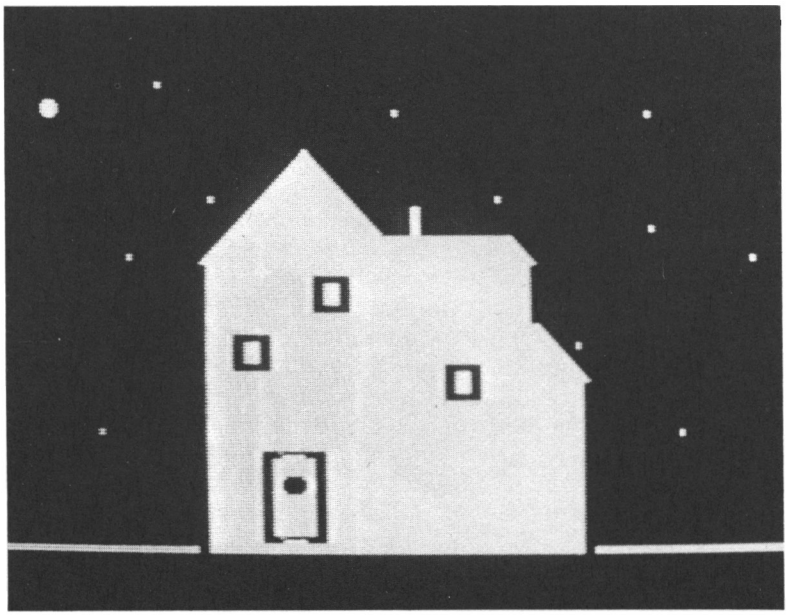
The key marked CONTROL has a very special purpose. Much like the SHIFT key, it can change the meaning of any key pressed simultaneously. For instance, when you press the CONTROL key, the "<" key becomes a CLEAR key and can be used to clear the screen. (In this and other multiple keystrokes, press only the key represented by the symbol within the quotation marks, not the quotation marks.) Try it and see. Bear in mind,

however, that both keys must be pressed at the same time to produce this effect—that is, you must press the CONTROL key and then press the "<" key while the CONTROL key is still being pressed. Similarly, the CONTROL key converts the ">" to an INSERT key, which will create a space between two existing characters to allow new characters to be inserted.

GRAPHICS CHARACTERS

We can use the CONTROL key to create a limited form of graphics on the Atari screen. For example, press the CONTROL key along with the T key. The image of a ball will appear on the display. If you press CONTROL along with the F key, a diagonal line will appear. Similarly, pressing CONTROL along with the G key will cause another diagonal line to appear, but with the opposite slant.

Quite a few of these *graphics characters* are available on the Atari keyboard, through the use of the CONTROL key. With a little imagination, you can combine several of these characters to form larger characters. Using the slanted lines on the F and G keys, for instance, it is possible to create diamond patterns on the screen, or interlocking networks of lines.



Original design created using Atari graphics characters

Not every key will produce a graphics character when pressed along with the CONTROL key. Some keys will produce actions on the screen, such as the screen clearing and character inserting we discussed a moment ago. One of the most important of these "screen actions" is the movement of the cursor. By pressing the CONTROL key along with the "−" key, the "=" key, the "+" key, and the "*" key, we can move the cursor up, down, left, and right, respectively. (For this reason, the designers of the Atari keyboard have placed pictures of tiny arrows on those keys, indicating the direction that the cursor will move when the key is pressed along with the CONTROL key.) In this way, we can position the cursor precisely where we want it on the screen, before we begin to type. Using these arrow keys in combination with the graphics character keys allows us to draw elaborate pictures on the Atari's screen.

(The earlier, non-XL Ataris have a special sketchpad mode activated automatically when no cartridge is present in any of the Atari's slots. If you have one of these computers, you may wish to use the sketchpad mode rather than BASIC for drawing pictures on the screen. The keys on the keyboard will behave in exactly the manner described above.)

THE ESC KEY

The ESC key, in the upper left-hand corner of the keyboard, performs a function similar, but by no means identical, to that performed by the CONTROL key. ESC is short for "escape," because it allows us to escape the normal function of a key. In certain instances, pressing the ESC key alters the meaning of the key pressed immediately after it. (In this case, the keys should *not* be pressed simultaneously.) The keys affected by ESC are primarily those that cause actions to take place on the screen, such as the cursor movement keys. When one of these "action" keys is pressed immediately after the ESC key, the action will not be performed as it normally would be; rather, a special graphics character will be printed on the display.

To demonstrate, press the ESC key, then press CONTROL-"=". The CONTROL-"=" combination—that is, the simultaneous pressing of the CONTROL and "=" keys—ordinarily moves the cursor down one line, as you saw a moment ago. When it is pressed immediately after the ESC key, however, the image of a downward-pointing arrow will appear instead. Similarly, pressing ESC and then CONTROL-"-" will produce the image of an upward-pointing arrow, rather than moving the cursor up one space.

Other keys altered by ESC include CONTROL-"+", CONTROL-"*", CONTROL-"<", CONTROL-">" and CON-

TROL-DELETE. Even the ESC key itself can be altered by the ESC key. Pressing the ESC key twice in a row produces one of the most interesting characters in the Atari character set—a kind of cursive E with a tail. Try it and see.

GRAPHICS

It is all well and good to draw pictures on the Atari video display using the keyboard, but the purpose of this book is to show you how to produce graphics using an Atari BASIC program. It is unlikely that you will want the user of your programs to draw his or her own graphic images using the keyboard; hence, there must be instructions in Atari BASIC that can be used to produce the same kind of effect.

In fact, there are quite a few such instructions. The most common, and versatile, graphics instruction recognized by the Atari is the PRINT command. As a BASIC programmer, you are undoubtedly familiar with the PRINT command as a means of placing text on the video display. However, it can also be used to create graphics.

For instance, to create a simple drawing of a fence on the Atari display, you can type:

```
PRINT "
```

then press the W key several times while holding down the CONTROL key. Finally, you can add another quote and press RETURN. A sequence of fencelike graphics characters will be printed on the display.

In the same way, other graphics characters can be printed simply by placing those characters in quotes after the word PRINT. Although we have demonstrated this technique by using the Atari immediate mode (where instructions are executed as soon as we press RETURN), we can just as easily place these instructions in our programs, by preceding the statements with line numbers.

THE CHARACTER SET

The total set of all characters that can be displayed on an Atari text screen is called the Atari *character set*. Obviously, this character set contains the letters of the alphabet, in both upper and lowercase, the ten numerals from 0 to 9, and a handful of punctuation marks. It also includes a bevy of graphics characters.

Although we see these characters on the display as visual images, they are represented inside the computer as numbers. In fact, all information is represented inside a computer as numbers.

Complete Atari character set

The numbers used to represent characters inside a computer are usually called *ASCII code numbers.* ASCII stands for *American Standard Code for Information Interchange.* However, the numbers that the Atari uses to represent characters are not always quite the same as the standard ASCII code; thus, we refer to these numbers as the ATASCII code.

To determine the ATASCII code for any character in the Atari character set, we can use the ASC function, like this:

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

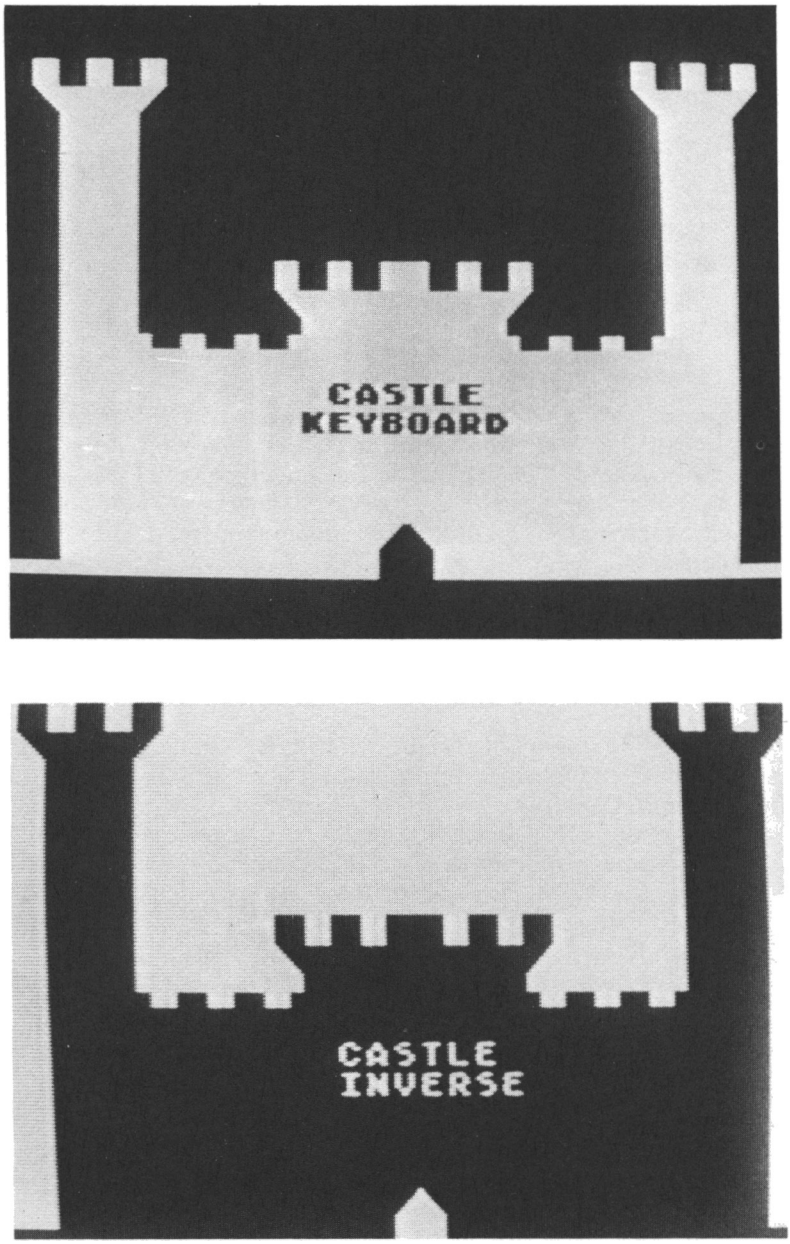
If you type this statement and press RETURN, the computer will respond with the number 65, indicating that the ATASCII code for the capital letter A is 65. If you had placed the slanted line character (CONTROL-F) within the quotes rather than the letter A, the computer would have responded with the number 6, indicating that the ATASCII code for the slanted line character is 6. You are invited to experiment with other characters, using the ASC function to determine their ATASCII codes.

Even certain screen actions are assigned ATASCII code numbers. For instance, moving a cursor up one line is represented by ATASCII code 28. Clearing the screen is represented by ATASCII code 125. And so forth. To determine the ATASCII code for a screen action character, use the ESC method described earlier to place an image of that character between quotes after the ASC function. For instance, to determine the ATASCII code for the CONTROL-">" combination, type

```
PRINT ASC("
```

then press ESC followed by CONTROL-">" and add

```
")
```



Examples of normal and reversed characters and designs

Finally, press RETURN. The ATASCII code for the CONTROL-">" combination (255, as it happens) will be displayed.

THE CHR$ FUNCTION

We can use the ATASCII code numbers to print characters on the display. To do this, we use the CHR$ function. The CHR$ function is the opposite of the ASC function. It produces the character represented by an ATASCII code number. For instance, if we type

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

the capital letter A will be displayed. If we type

```
PRINT CHR$(6)
```

the slanted line character will be displayed.

Interestingly, adding 128 to the ATASCII code of a character (assuming that the code was less than 128 to begin with) will create a negative image of the character. We call these negative characters *reversed characters.* To see the reverse of the letter M, for instance, you could type

```
PRINT CHR$(ASC("M") + 128)
```

You can also produce reversed characters by pressing the key in the lower right-hand corner of the keyboard, which automatically puts the Atari in reversed text mode.

Which method you use to print characters in your programs—typing the graphics characters from the keyboard or using the CHR$ function—will vary according to circumstance and personal taste. However, in this book we will use the CHR$ method to print all graphics characters and screen action characters, because this method is likely to produce less visual confusion as you type the program listings provided here. However, you may use either method in the programs you write yourself.

LARGE TEXT MODES

Graphics mode 0 is not the only text mode offered by the Atari. Two others are graphics modes 1 and 2. As you can probably deduce, we can enter these modes by typing

```
GRAPHICS 1
```

and

GRAPHICS 2

respectively.

Although these additional modes are also used for displaying text, there are certain differences between the way text is displayed in these modes and the way it is displayed in mode 0. Before we try out these new modes, press the RESET key at the upper right-hand corner of the Atari console. This will clear out any side effects left by previous graphics instructions. (If you have just turned on the computer, this won't be necessary.)

Now type

GRAPHICS 1

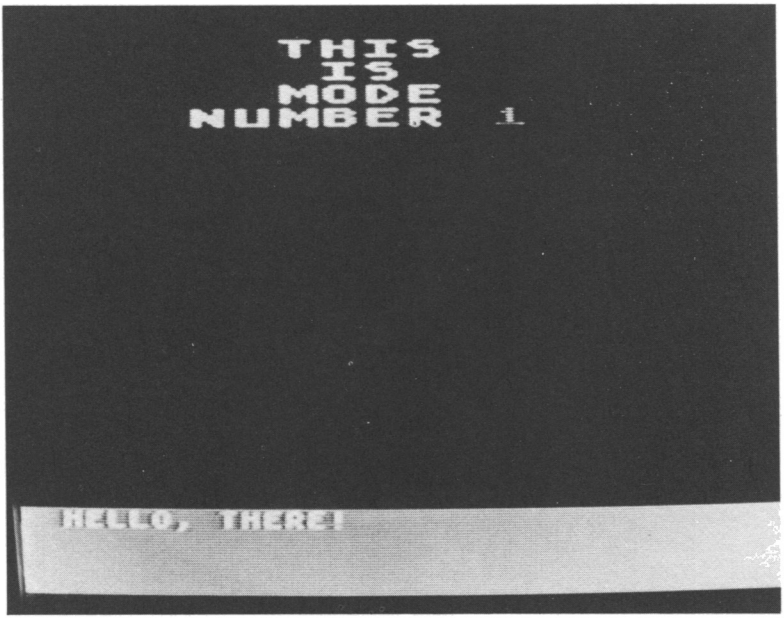
and press RETURN. The display should turn black, except for a small section at the bottom of the screen that will remain the same blue color that you saw while in mode 0. The word READY and a cursor should appear in the blue section of the display. The black portion of the display is in mode 1. The blue portion of the display is in mode 0. Because the display is in two modes at once, we say that we are using a *mixed graphics mode.* The portion of the display in mode 0 is called the *text window,* because it provides a small opening in which we can display ordinary mode 1 text. Later, we will learn how to place the entire display in mode 1, without the text window.

While in mode 1, we can still type BASIC instructions to the computer, although all of our typing will now appear within the limited confines of the text window. For instance, type

PRINT "HELLO, THERE!"

and press RETURN. The words HELLO, THERE! will appear on the display, just as they would in mode 0. However, notice that they appear in the text window and not in the mode 1 portion of the display. Earlier, we said that mode 1 is a text mode. Yet, you might wonder what good it does to have additional text modes if they do not display text.

The PRINT statement, unless we modify it in some way, will always output text to the mode 0 text window. (If there is no mode 0 text window, the OS will automatically place the entire screen in mode 0 when the PRINT statement is executed.) The way we modify the PRINT statement to print on the mode 1 portion of the display is to change it to a PRINT #6 statement. The #6 portion of the statement tells the operating system that we wish to print to the upper portion of the graphics display, no matter what mode it is currently in.



A display in mixed graphics mode

To demonstrate, type

```
PRINT #6; "HELLO, THERE!"
```

and press RETURN. The words HELLO, THERE! will now appear at the very top of the display, in the mode 1 portion. Notice, however, that the characters look quite different than they did on the mode 0 screen. They are larger, or at least wider. And this, in fact, is one of the essential differences between modes 0 and 1 (though not the *only* difference). Mode 1 produces characters twice as wide as those in mode 0. While we can print thirty-eight characters across the display in mode 0, we can print only nineteen characters across in mode 1. On the other hand, both modes allow us to print twenty-four lines of characters—or would, if the mode covered the entire display.

Some uses for mode 1 text should immediately come to mind—the design of eye-catching title screens, for instance, where you may wish to use large and dramatic characters. We can also use mode 1 to increase the size of our graphics characters, which would produce more substantial (if somewhat less detailed) graphic images.

Before you decide to do all title screens and graphics in mode 1, however, let's take a look at mode 2. Type

```
GRAPHICS 2
```

and press RETURN. First, the screen will clear. (This happens automatically when we type the graphics statement, though in a moment we will discover a way to stop it from happening.) Then it will return to a state that looks remarkably like mode 1: a black screen with a blue text window at the bottom. Are we back in mode 1 again?

No, although mode 2 behaves much like mode 1. For instance, PRINT statements will still direct their output to the text window, as before. And we can still print information on the mode 2 screen with PRINT #6. Type

```
PRINT #6; "HELLO, THERE!"
```

and press RETURN. Once again, the words HELLO, THERE! will appear on the upper, mode 2 portion of the display. Now, however, the letters are even larger. Not only are they twice the width of the mode 0 characters, but they are twice the height. Even if mode 2 filled the entire display, we could print only twelve lines of these characters at one time, instead of the usual twenty-four.

Think of the eye-catching title displays you could create with mode 2! Think of the outsized graphic images you could piece together from the enlarged graphics characters!

The text mode you choose to work in, of course, depends on what effect you are trying to achieve. In some cases, mode 2 (or even mode 1) characters may be too large for your purposes; in a few cases, they may even be too small (in which event you are left to your own devices, since this is the largest character mode that the Atari will supply).

When we wish to exit the current text modes and return to mode 0, we can simply type

```
GRAPHICS 0
```

and press RETURN. An easier method, however, is to press the RESET key. This causes the display to revert to mode 0, but does not affect any programs that we have in memory.

REMOVING THE TEXT WINDOW

With a tiny bit of extra effort, we can produce variations on these two new graphics modes. If we wish to eliminate the text window,

for instance, we can add 16 to the mode number in the GRAPHICS statement. The statement

```
GRAPHICS 1 + 16
```

will place the display in mode 1 without a text window. This may also be written as GRAPHICS 17; however, the notation GRAPHICS 1 + 16 makes the intent of the statement clearer to anyone, ourselves included, who attempts to read the program that we have written.

If you attempt to type this statement in the immediate mode, a peculiar thing will happen. The display will be placed ever so briefly in mode 1, then will revert automatically to mode 0. Why does this happen? Earlier, we said that the OS would automatically restore the display to mode 0 if we tried to use an ordinary PRINT statement on a display without a mode 0 text window (unless, of course, the display was in mode 0 to begin with). The same thing happens when BASIC attempts to print the word READY on the display after executing our immediate mode instruction. There is no text window to print it in; therefore the OS automatically reverts the entire display to mode 0.

Thus, when we create a full-screen mode other than mode 0 we must do so in a program, rather than in the immediate mode. To see a mode 1 screen in action without a text window, type this program:

```
10 GRAPHICS 1 + 16
20 FOR I = 1 TO 12
30 PRINT #6; "THIS IS A FULL"
40 PRINT #6; "SCREEN TEXT MODE"
50 NEXT I
60 GOTO 60
```

and RUN it.

What does this program do? Line 10 places the full display in mode 1, without a text window. The loop in lines 20 through 50 fills the display with the words THIS IS A FULL SCREEN TEXT MODE. Finally, line 60 creates an infinite loop that prevents BASIC from printing the word READY on the display and thereby restoring mode 0. To exit from this program, press either BREAK or RESET. (Note that we must be careful not to allow the printing to run off the end of the mode 1 display. Unlike mode 0, the mode 1 display will not automatically scroll up one line when we reach the bottom; rather, we will receive an error message if we attempt to print past the last line of the display.)

Similarly, we can create a full mode 2 display by typing GRAPHICS 2 + 16 (or GRAPHICS 18).

To demonstrate, change lines 10 and 20 in the above program to:

```
10 GRAPHICS 2 + 16
20 FOR I = 1 to 6
```

and RUN it.

UNCLEARING THE SCREEN

Another variation we can play is to enter one of these graphics modes without clearing the screen first. To do this we add 32 to the mode number in the GRAPHICS statement. For instance, to enter mode 1 without clearing the screen, we would type GRAPHICS 1 + 32 (or GRAPHICS 33). Here is an example:

```
10 GRAPHICS 0
20 FOR I = 1 TO 18
30 PRINT "THIS IS MODE 0"
40 NEXT I
50 GRAPHICS 1 + 32
60 PRINT #6; "BUT THIS IS MODE 1"
70 GOTO 70
```

This program fills the screen with mode 0 text, then switches to mode 1 with the text (or as much of it as will fit) still intact, plus one extra statement. During the switch from one graphics mode to another, of course, the text changes size and position on the display.

If we wish to combine both of these variations—that is, to enter a full-screen graphics mode without clearing the display—we add 32 + 16 (or 48) to the mode number. To enter a full-screen mode 2 without clearing the screen we would type GRAPHICS 2 + 32 + 16 (or GRAPHICS 2 + 48 or GRAPHICS 50).

A great deal can be done graphically within the limits of these three graphics modes. Before we examine these modes further, however, let's pause and look at some other modes that offer us a completely different method of putting graphics on the display: bitmap graphics.

Suggested Projects

1. Using the CHR$ function and a FOR-NEXT loop, write a program that will print the entire character set of your computer on the mode 0 display.

2. Alter the above program to print the character set on the mode 1 display. What differences do you see in the character set? How many different characters are displayed? How do some of these characters differ from others?

3. What instruction will cause the OS to put the display into each of the following modes?

   a. Mode 2 display without a text window.
   b. Mode 1 display with a text window but without a cleared screen.
   c. Mode 0 display without a text window or cleared screen.

2 BITMAP MODES

Earlier, we compared computer graphics to the art of mosaic and said that programming computer graphics was primarily a matter of putting colored pixels together to form a picture.

After reading the last chapter, you may wonder if this is really so. We have, until now, spent a lot of time putting characters on the display, but precious little time dealing with pixels. When we create text graphics, are we really manipulating individual pixels?

The answer is a qualified yes. If you look closely at the characters that you type on the Atari display, be they letters of the alphabet or graphics characters or whatever, you will see that these characters are made up of tiny dots. In fact, each character is made up of a matrix of eight dots horizontally and eight dots vertically, for a total of sixty-four dots in all. These dots are pixels. You may not see all sixty-four pixels when you look at a character on the display, but this is simply because some of these pixels are the same color as the background, rendering them effectively invisible.

Thus, when we program text mode graphics we are telling the computer what patterns of pixels we want it to place on the display. However, we are limited to the use of predefined patterns placed in the computer by the manufacturer. (As noted earlier,

there is a way to escape this restriction, which we will study in the section on advanced graphics.)

THE BITMAP REALM

By leaving the text modes altogether, we enter a completely different realm of graphics creation, where there are no predefined patterns of pixels. In this realm we can tell the Atari to place pixels at specified positions on the display, in specified colors. We have complete freedom to produce any combinations of pixels that we wish, within certain limitations that will soon become apparent.

We gain this freedom for a price, however. Displays that would be simple to create in a text mode become surprisingly difficult in one of these bitmap modes. Fortunately, the opposite is true as well. The bitmap modes available with the GRAPHICS statement are modes 4 through 11, 14 and 15.

If you have one of the earlier, pre-1982 Ataris, without the GTIA chip, modes 9 through 11 will not be available. If you are not sure if your machine precedes this date, type this program:

```
10 GRAPHICS 9
20 GOTO 20
```

and press RETURN. If the display turns black, you have the GTIA chip and can use modes 9 through 11. If nothing happens, you have an earlier Atari with the CTIA chip, and will have to sit out the description of these modes.

You will be able to use modes 14 and 15 only if the name of your Atari model contains the letters XL. Owners of pre-XL machines must sit out the description of these modes even if they have the GTIA chip. (However, you'll learn a way of creating these modes without the GRAPHICS statement, in the section on advanced graphics.)

There are two essential differences between the various bitmap modes. One is resolution, that is, the size and shape of the pixels that can be displayed. The other is the number of different colors that can appear on the display at one time. We'll discuss color more fully in the next chapter; for now we'll concentrate primarily on resolution.

GRAPHICS MODE 7

Mode 7 is fairly representative of the Atari bitmap modes, so we will look at this mode in some detail before we study the others. It is a medium-resolution mode, which is to say that we can use it to

produce fairly detailed images, but not the most detailed the Atari is capable of. It is also a four-color mode, which means that we can display four colors simultaneously while in mode 7.

To enter mode 7, type

```
GRAPHICS 7
```

and press ENTER. As in modes 1 and 2, the display clears and turns black, except for a small blue text window at the bottom of the display. In fact, mode 7 looks suspiciously like modes 1 and 2. Could this really be a bitmap mode instead of a text mode?

To find out, type this statement:

```
PRINT #6; "THIS IS NOT A TEXT MODE"
```

in the text window and press RETURN.

Instead of printing the text in the upper portion of the display, the Atari prints a line of dots in response to this command. These dots are pixels. You might wonder what they have to do with the sentence that we asked the computer to print. The answer is: Not much.

The Atari OS will obligingly print text on the display in any graphics mode that we happen to be using; at least, it will *try* to print the text. Unfortunately, text is meaningless in a bitmap mode. Instead of text, Antic and GTIA now expect to receive information concerning the pixels that we wish to draw on the display. The graphics chips make an earnest attempt to interpret our text as a sequence of pixels, but the result usually has nothing to do with the text we wanted to print.

The solution to this problem is simply not to use the PRINT #6 statement while in a bitmap mode. We can still use the ordinary PRINT statement to place text in the text window, assuming there *is* a text window. In addition, Atari BASIC offers us several statements that can be used to place meaningful or interesting patterns of pixels in the bitmap mode portion of the display.

THE PLOT STATEMENT

Though you may not be aware of it, the mode 7 display that you are now looking at is covered with pixels. Unfortunately, these pixels are all the same color. In order to change the color of some of these pixels, we can use the PLOT statement.

The syntax for the PLOT statement is

```
PLOT column,row
```

where "column" and "row" are the horizontal and vertical coor-

dinates, respectively, of the pixel whose color we wish to change. These coordinates are given as numeric expressions.

The Atari display is divided into a matrix of pixels—that is, the dots of color are arranged in horizontal rows and vertical columns. The number of pixels in this matrix depends on what bitmap mode we are using, just as the number of characters that we can place on the display in a text mode depends on what text mode we are using.

In mode 7, the matrix contains 12,800 pixels (if there is a text window present) or 15,360 pixels (in a full-screen mode). These pixels are arrayed in a specific pattern. In mode 7, the display is 160 pixels wide and 80 or 96 pixels high, depending on whether there is a text window. (As before, we can turn off the text window by adding 16 to the mode number in the GRAPHICS statement. There is, however, no text window available in modes 9 to 11.)

The coordinates that we use with PLOT tell the Atari the horizontal and vertical positions within this matrix of the pixel we want to plot. The first number specifies what horizontal row of pixels the pixel is in, where the top row of pixels is row 0. The second number specifies what column of pixels the pixel is in, where the leftmost column of pixels is column 0. In mode 7, the first number may be any number from 0 to 159 and the second number may be any number from 0 to 79 or 95, depending (once again) on whether or not there is a text window.

Before we use the PLOT statement, we must tell the OS what color we wish to change the pixels to. This is done with the COLOR statement. The syntax for the COLOR statement is

```
COLOR color number
```

where "color number" is a number from 0 to 4.

We'll have more to say about color numbers in the next chapter, so for now, simply type

```
COLOR 3
```

while in mode 7 and press RETURN. This tells the OS to plot all subsequent pixels in light blue, until told otherwise. (If you've run any graphics programs other than the ones in this book since you turned on the computer, you might hit the RESET key and issue another GRAPHICS 7 statement before typing this statement, to make sure that color 2 is light blue. More about this later.)

Now type

```
PLOT 80,42
```

This statement tells the computer that the pixel we wish to plot—that is, the pixel whose color we wish to change—is in row 80 and column 42. A light-blue pixel should appear roughly in the center of the display.

DRAWING LINES

That wasn't very hard, was it? However, plotting a single pixel on the Atari display isn't exactly a useful occupation. We need to be able to put together a meaningful picture out of pixels. This generally involves putting a lot of pixels on the screen very quickly, in some sort of pattern. For instance, here is a program that will arrange a sequence of mode 7 pixels in a straight line:

```
10 GRAPHICS 7 + 16
20 COLOR 2
30 FOR COLUMN = 30 TO 130
40 PLOT COLUMN,42
50 NEXT COLUMN
60 GOTO 60
```

This program cycles the value of the variable COLUMN through a series of possible values between 30 and 130. It then plots the pixel at each successive value of COLUMN, 42 in color number 2. This causes a light-green straight line to be drawn across the middle of the display.

(Readers not previously familiar with Atari BASIC may be surprised at the length of the variable name used in this program. Unlike many other versions of BASIC, which recognize only the first two characters of a variable name, Atari BASIC will recognize a full 128 characters in a variable name, allowing us to adopt names that suggest the purpose for which the variable is being used, making our programs easier to read. Note, also, that Atari BASIC requires that we place the name of a variable after the word NEXT in a NEXT statement, unlike other BASICs in which this is optional.)

Here is a program that will arrange a sequence of mode 7 pixels in a near square:

```
10 GRAPHICS 7 + 16
20 COLOR 2
30 FOR ROW = 20 TO 60
40 FOR COLUMN = 60 TO 100
50 PLOT COLUMN,ROW
60 NEXT COLUMN
70 NEXT ROW
80 GOTO 80
```

Similarly, this cycles the values of variables ROW and COLUMN through a series of possible values between 20 and 60 and between 60 and 100, respectively. It then plots the pixel at each successive value of the COLUMN,ROW coordinates. This causes a rectangular area on the display to be filled with pixels.

THE DRAWTO STATEMENT

There is an easier way to achieve these and similar effects, using the DRAWTO statement. The DRAWTO statement plots a straight line of pixels on the display between the pixel most recently plotted and another, specified set of coordinates.

The DRAWTO statement is written almost identically to the PLOT statement, like this:

```
DRAWTO column,row
```

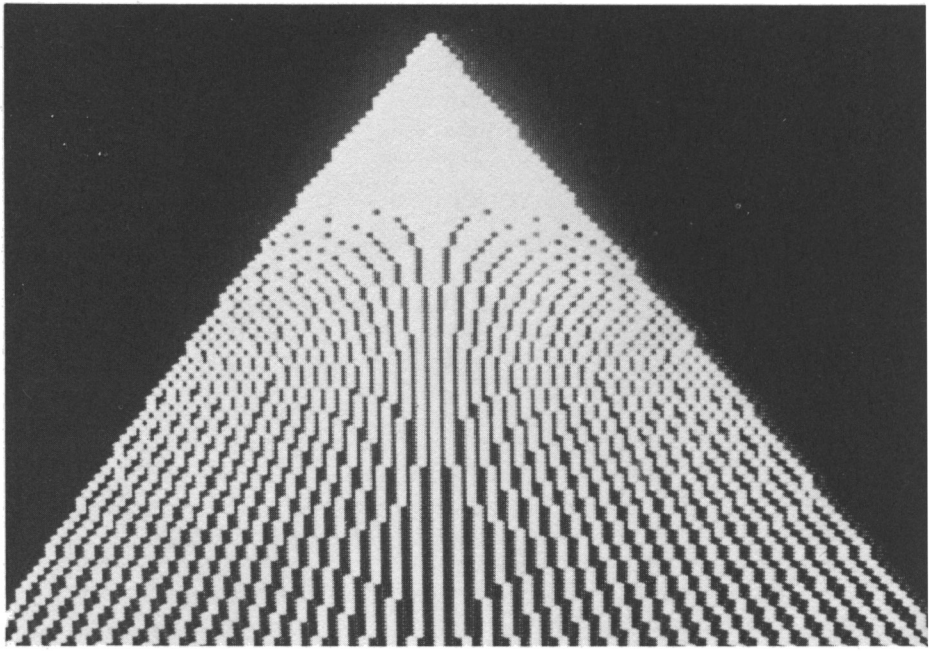
where "column" and "row" are the horizontal and vertical coordinates at which you wish the line to end. The line will begin at the coordinates of the most recently plotted pixel; if we have not previously plotted a pixel before using the DRAWTO statement, the beginning of the line will be assumed to be at coordinates 0,0.

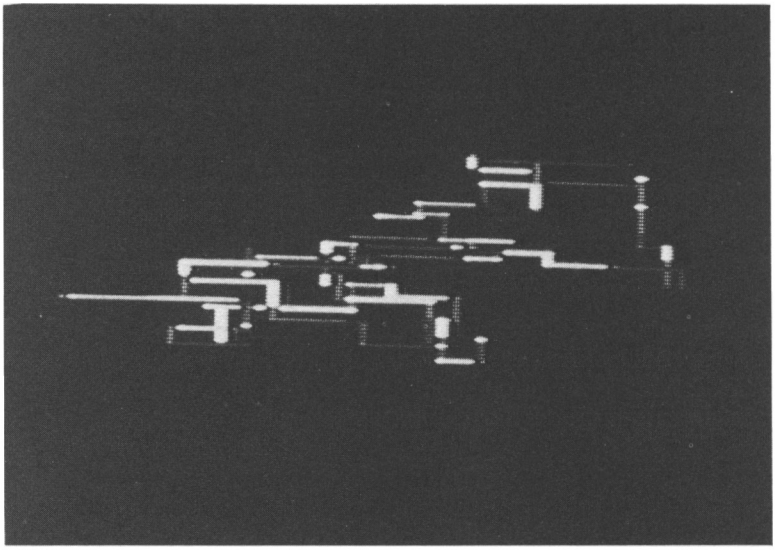
For instance, to draw a line between coordinates 55,16 and 7,62 in mode 7, we could use this program:

```
10 GRAPHICS 7 + 16
20 COLOR 2
30 PLOT 55,16
40 DRAWTO 7,62
50 GOTO 50
```

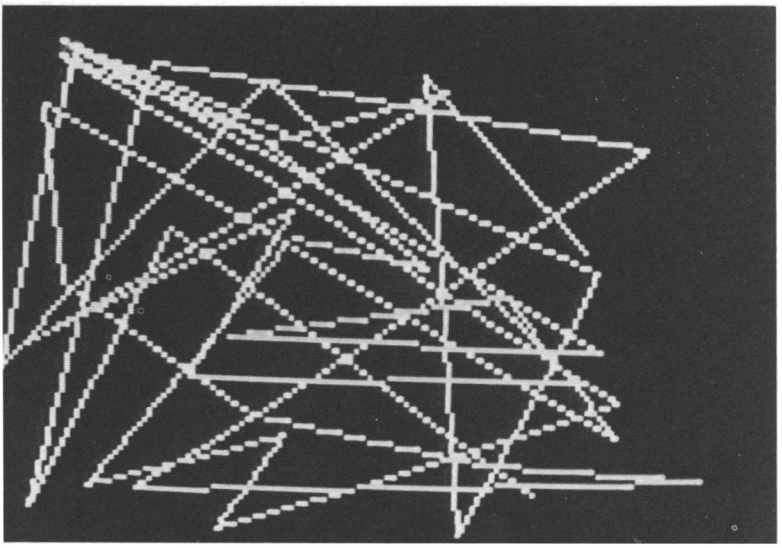
The PLOT statement in line 30 establishes the beginning of the line as 55,16. The DRAWTO statement in line 40 will draw a line from this position to coordinates 7,62. Any subsequent DRAWTO statements will begin at the coordinates where this DRAWTO statement leaves off—i.e., 7,62—unless there are intervening PLOT statements.

Try drawing some lines of your own with DRAWTO; just bear in mind that both endpoints of the line must fall within the coordinate range of the mode 7 screen, or you will be dumped unceremoniously back into mode 0 with an error message. You'll probably notice quickly enough that the diagonal lines drawn by DRAWTO are not precisely straight; a degree of jaggedness creeps in. This is because the pixels generated in mode 7 are not small enough to blend into a perfect diagonal. There is a kind of stairstep effect. The pixels form the line as a series of horizontal





The programs for these designs utilize DRAWTO statements.



RUNning the random-line program in this chapter creates a picture that looks something like this.

or vertical segments, approximating but not quite duplicating a genuinely straight line. Nonetheless, the effect is quite sufficient for most graphics purposes.

Here is a program that draws a random web of lines across the Atari display, each line beginning where the previous line leaves off:

```
10 GRAPHICS 7 + 16
20 PLOT INT(RND(1)*160), INT(RND(1)*96)
30 DRAWTO INT(RND(1)*160), INT(RND(1)*96)
40 GOTO 30
```

Line 20 generates a random starting point for the line with the RND function. The Atari function RND(1) is always equal to a random fraction between 0 and 1. By multiplying this value by 160 and chopping off any fractional values with the INT function, we produce a random whole number between 0 and 160. Similarly, multiplying the RND(1) value by 96 produces a random number between 0 and 96. Line 30 then uses the same method to calculate a random endpoint for a line and draws the line. Line 40 sets up an infinite loop that causes this latter process to repeat again and again. To exit the program, hit the BREAK key.

DRAWING SHAPES

The DRAWTO statement is a powerful graphics tool; it allows us to place many pixels on the screen all at once in a meaningful pattern—a straight line. It shouldn't be hard to see how this could be used to place decorative borders around the screen, for instance, or to create drawings of geometric objects. This program uses DRAWTO to draw a square:

```
10 GRAPHICS 7 + 16
20 COLOR 2
30 PLOT 60,20
40 DRAWTO 100,20
50 DRAWTO 100,60
60 DRAWTO 60,60
70 DRAWTO 60,20
80 GOTO 80
```

Drawing geometric shapes like this is simply a matter of determining the coordinates of the vertices of the shapes—that is, the points at which two lines come together—and using those coordinates as the endpoints of the lines. Sometimes, however, we may wish to express the coordinates as variables rather than numbers, so that the shape can be moved during the course of the program. For instance, here is a variation on the above program that allows us to place the square at any point on the display, depending on how we answer the opening question:

```
10 GRAPHICS 0
20 ? "TYPE THE X,Y COORDINATES OF THE UPPER"
30 ? "LEFT-HAND CORNER OF THE SQUARE"
40 INPUT X,Y
50 IF (X>119) OR (X<0) OR (Y>55) OR (Y<0) THEN ? "BAD
COORDINATES" : GOTO 20
60 GRAPHICS 7 + 16
70 COLOR 2
80 PLOT X,Y
90 DRAWTO X+40,Y
100 DRAWTO X+40,Y+40
110 DRAWTO X,Y+40
120 DRAWTO, X,Y
130 GOTO 130
```

(Note that we use the abbreviation "?" for the PRINT statement in this program. This is a common usage in Atari programs, and we will use it frequently in this book. The BASIC interpreter will treat it exactly as though it were a PRINT statement.)

This program prompts the user to input the horizontal and vertical coordinates of the upper left-hand corner of the square, stores these coordinates in variables X and Y, then checks to see if they will cause the square to run off the edge of the display. If not, it prints the square at that position.

A reading of lines 80 through 120 should make it obvious how the program calculates the vertices of the square. The X coordinate of the upper right-hand corner is 40 greater than the X coordinate of the upper left-hand corner, while the Y coordinate is identical; hence, if X and Y are the coordinates of the upper left-hand corner, then the coordinates of the upper right-hand corner are X+40,Y, no matter what the values of X and Y are. Similar logic is used to calculate the other vertices. Hit BREAK and type RUN to try the program more than once. You are encouraged to try it with many different coordinates for the square.

SCALING AND TRANSLATING

Moving a picture from one position on the screen to another by having the computer recalculate the lines that make up that picture is called *translation*; we have *translated* the picture from one position to another. Similarly, if we were to take the amounts that are added to the X and Y coordinates in the above program and make them into variables too, we would be able to change the size and even the proportions of the square in much the same way; this process is called *scaling,* because it allows us to change the scale of the drawing. To demonstrate, make the following changes and additions to the previous program:

```
42 ? "TYPE HEIGHT OF SQUARE"
44 INPUT HEIGHT
46 ? "TYPE WIDTH OF SQUARE"
48 INPUT WIDTH
50 IF ((X+WIDTH)>159) OR (X<0) OR ((Y+HEIGHT)>195) OR (Y<0)
THEN ? "BAD COORDINATES" : GOTO 20
90 DRAWTO X+WIDTH, Y
100 DRAWTO X+WIDTH, Y+HEIGHT
110 DRAWTO X, Y+HEIGHT
```

Try different values for the WIDTH and HEIGHT, as well as the coordinates of the upper left-hand corner. You can create small rectangles, large rectangles, short, fat rectangles and tall, skinny ones. Be warned, however, that this program does not check to see if any of the lines will run off the edge of the display. If one does, the program will be interrupted, mode 0 restored, and an error message printed. You might want to try your hand at

rewriting line 50 so that it checks for lines that will extend beyond the allowable coordinates of mode 7 (0 to 159 horizontally and 0 to 95 vertically). Hint: add the value of X to WIDTH to get the highest horizontal coordinate and the value of Y to HEIGHT to get the highest vertical coordinate, and check to see if these values (as well as the beginning values of each line) are in range.

Here is a program that plays some variations on the scaling and translation concept by drawing a series of concentric squares in the center of the display:

```
10 GRAPHICS 7 + 16
20 YCENTER = 48 : XCENTER = 80
30 FOR SIZE = 4 TO 44 STEP 4
40 PLOT XCENTER-SIZE, YCENTER-SIZE
50 DRAWTO XCENTER+SIZE, YCENTER-SIZE
60 DRAWTO XCENTER+SIZE, YCENTER+SIZE
70 DRAWTO XCENTER-SIZE, YCENTER+SIZE
80 DRAWTO XCENTER-SIZE, YCENTER-SIZE
90 NEXT SIZE
100 GOTO 100
```

THE XIO STATEMENT

The DRAWTO command helps us get a lot of pixels onto the screen very quickly, and in a meaningful pattern. However, Atari BASIC (with a little help from the OS) can do the DRAWTO command one better with a command that fills entire areas of the screen with pixels in a single sweep. This is the XIO 18 command, sometimes unofficially referred to as XIO (FILL).

XIO is the Atari's all-purpose output command. In effect, it is a nearly direct pipeline to the OS; it allows us to access the routines in the OS with minimal intervention by BASIC. It can be used to duplicate the work of several other Atari BASIC commands, such as PRINT, INPUT, GET, and several others. The number that follows the XIO command indicates which particular operating system routine we want to use.

There is no other BASIC command equivalent to XIO 18. It fills an entire area of the display with pixels. Alas, it is rather awkward to use. It is best for filling rectangular areas, as we did in a much earlier pixel plotting program. We must first draw a portion of that rectangular area with the PLOT and DRAWTO commands, then fill it with XIO 18. Here is the precise sequence of steps necessary to fill a rectangle with pixels:

- PLOT the point at the lower right-hand corner of the rectangle.

- DRAWTO the upper right-hand corner.
- DRAWTO the upper left-hand corner.
- Give the command POSITION X,Y, where X and Y are the coordinates of the lower left-hand corner.
- Give the command POKE 765, **color register**, where color register is the same as the number used in the current COLOR command.
- Type XIO 18, #6, 0, 0, "S:"

There are a (very few) possible variations on these steps, but on the whole they should be performed precisely like this. Even if you don't understand the point of all of these steps, following them letter by letter will produce a rectangular area filled with pixels. Here is a sample program:

```
10 GRAPHICS 7 + 16
20 COLOR 2
30 PLOT 100,60
40 DRAWTO 100,20
50 DRAWTO 60,20
60 POSITION 60,60
70 POKE 765,2
80 XIO 18, #6, 0, 0, "S:"
90 GOTO 90
```

Run the program; the rectangle fills with color much faster than when we filled the earlier rectangle using a pair of loops and PLOT statements.

THE CHANGING PIXEL

All of the tricks we can perform with pixels in mode 7 can be performed in all of the other bitmap modes. All that changes is the number of pixels that we can squeeze on the display at one time and the number of colors in which we can squeeze them.

To demonstrate the differences in resolution between graphics modes 3 through 8, the following is a demo program that draws a line in each mode, printing the number of the mode in the text window at the bottom of the display. Note that the line is identical in each mode; that is, the starting and end coordinates of the line are the same. Here's the program:

```
10 FOR I = 3 TO 8
20 GRAPHICS I
30 PRINT "GRAPHICS MODE ";I
40 COLOR 1
```

```
50 PLOT 0,0
60 DRAWTO 39,23
70 FOR J = 1 TO 1000
80 NEXT J
90 NEXT I
100 GOTO 100
```

In the first mode, mode 3, the line from coordinates 0,0 to 39,23 is huge, blocky looking. It stretches all the way across the screen, from left to right, top to bottom. In mode 4, it shrinks. Now it stretches only halfway across the screen. The line itself is thinner, less blocky, though it still has a pronounced "stairstep" look. In mode 5, there is no apparent change. In mode 6, the line becomes smaller still, and thinner, though it still gives the impression of being strung together out of a number of small beads. It now stretches about one-quarter of the way across the screen. In mode 7 there is, once again, no apparent change. In mode 8, it shrinks away almost to nothing. It stretches no more than one-eighth of the way across the screen. The stairstep look is almost completely gone; the line looks like a genuine diagonal line, just like one you might draw with a pencil, though perhaps somewhat neater.

If you have an XL machine and want to see what the line looks like in modes 14 and 15, change the following program line:

```
10 FOR I = 14 TO 15
```

and RUN the program again. If you have the GTIA chip and want to see what the line looks like in modes 9 through 11, delete line 30 and make the following change:

```
10 FOR I = 9 TO 11
```

We cannot print the number of the graphics mode in the text window in modes 9 through 11 because there is no text window in modes 9 through 11. Note that the line seems to vanish altogether in mode 10; this is normal. If you could see it, it would look pretty much like it does in modes 9 and 11.

The reason that our line looks different in some modes is of course that the size of the pixel changes. It is largest in mode 3, a low-resolution mode. It is somewhat smaller in modes 4 and 5, which are low-to-medium-resolution modes. It is smaller still in modes 6 and 7, which are medium-to-high-resolution modes. And it is smallest in mode 8, the Atari's highest-resolution mode. (Modes 9 through 11 and modes 14 and 15 fall into the middle-resolution range.)

THE PIXEL MATRIX

As the size of the pixel changes, the number of pixels that can be placed on the display at one time changes. This in turn alters the number of pixels in the rows and columns of the pixel matrix. Table 1 lists the number of pixels on the display in each mode (the annotation "split" after the mode number indicates a text window), along with the range of pixels vertically and horizontally.

TABLE 1
NUMBER OF PIXELS ON DISPLAY IN EACH MODE AND RANGE OF PIXELS

Mode	Number of Pixels	Number per Row	Number per Column
3	960	40	24
3 (split)	800	40	20
4	3840	80	48
4 (split)	3200	80	40
5	3840	80	48
5 (split)	3200	80	40
6	15360	160	96
6 (split)	12800	160	80
7	15360	160	96
7 (split)	12800	160	80
8	61440	320	192
8 (split)	51200	320	160
9	15360	80	192
10	15360	80	192
11	15360	80	192
14	30720	160	192
14 (split)	25600	160	160
15	30720	160	192
15 (split)	25600	160	160

You'll notice that several modes—4/5, 6/7, 9/10/11, 14/19—have identical pixel matrices. That is, not only do they put the same number of pixels on the display, but they arrange those pixels in identical patterns. There *are* differences between these modes, however, as we shall see later.

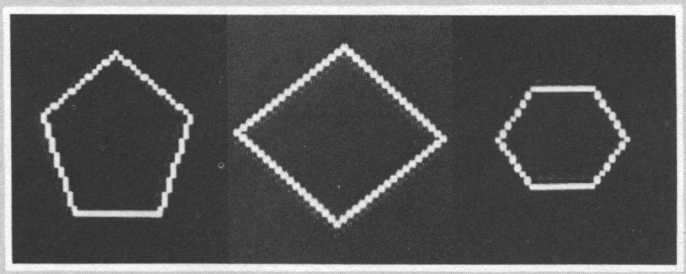
When plotting pixels on the Atari display, we must always be careful not to plot a point off the edge of the matrix. In mode 3, for instance, we could not plot the pixel with coordinates 50,29 because no such pixel exists in mode 3, although we could easily

plot such a point in mode 7. If you should rashly attempt this anyway, the Atari will break into your program with an error message and return you to mode 0. Thus, you should always consult Table 1 or a similar chart before plotting pixels in a given mode. Note, however, that the highest coordinate that can be used for the row or column of a pixel is one *less* than the number of pixels in that row or column. This is because the 0 coordinate is used for the first pixel in each row and column. For instance, in mode 7 we could not plot a pixel at a coordinate position past 159,79 (split screen) or 159,95 (full screen).

Now that we've taken a look at how to change the color of a pixel with the PLOT command, in the next chapter we will look at how we determine the color that the pixel will take once it has been plotted.

Suggested Projects

1. Write programs that draw five-sided, six-sided, and seven-sided figures on the Atari display. Write a program that draws a four-sided, diamond-shaped figure.



Designs created for Project 1

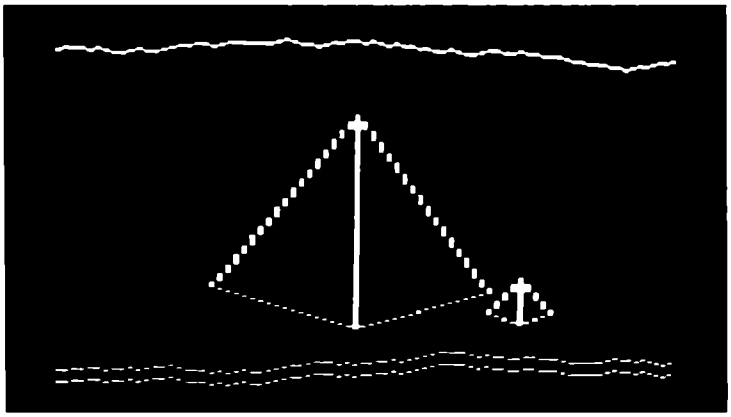
2. Write a program that will ask for a pair of coordinates on the display, then draw a six-sided figure starting at that point. Rewrite the program to allow the user to modify the lengths of the sides of the figure.

3. Write a program that PLOTs a pixel in the middle of the display. Use the XIO 18 technique described earlier to draw a box around this pixel, and fill the box with colored pixels. What happens when the fill color reaches the pixel in the middle of the box?

4. Write a program that uses the PLOT command to draw a border around the edges of the mode 3 display. Rewrite the program to draw borders around the modes 5, 7, and 8 displays. Rewrite each of these programs to draw the same border using the DRAWTO command rather than the PLOT command.

3
COLOR

The Atari graphics chips, in cooperation with the Atari operating system, will generate sixteen different colors. On most computers, this would be considered an adequate range of colors. The Atari, however, also allows us to display each color in eight different luminances—that is, levels of brightness—for a total of 128 different color/luminance combinations.



Original design created on the Atari

Here is a program that will cycle the Atari display through all 128 of these, so you can see what they look like:

```
10 FOR I = 0 TO 15
20 FOR J = 0 TO 14 STEP 2
30 SETCOLOR 2,I,J
40 FOR K = 1 TO 100
50 NEXT K
60 NEXT J
70 NEXT I
```

There is no way to get all 128 of these colors on the display at the same time, not without some extremely advanced programming tricks. Most of the Atari graphics modes only allow two or four colors to appear on the display simultaneously.

THE SETCOLOR STATEMENT

Before we plot a pixel on the display, we must tell the OS what color we wish to plot it in. However, we don't specify the actual color we wish to use; rather, we specify the number of a *register* identified with that color.

A register is simply a location within the computer's memory. This location contains a color number, a number indicating which color and luminance are to be identified with that particular register. There are five color registers, numbered 0 through 4. We place color numbers in those registers with the SETCOLOR statement. The syntax for the SETCOLOR statement is:

```
SETCOLOR register,color,luminance
```

where register is a numeric expression (a number or a variable or a combination thereof) representing the desired color register, and color and luminance are arithmetic expressions representing the color and luminance that we wish to identify with that register.

The register value after SETCOLOR must fall in the range from 0 to 4. In the last chapter, we saw that we could specify a plotting color with the COLOR statement, which consisted of the word COLOR followed by a color-register number. Oddly, this number is not the same as the register number in the SETCOLOR statement, although it refers to that number. The SETCOLOR register referred to by the number in the COLOR statement varies depending on which graphics mode we are using. Table 2 lists the correspondences between the COLOR and SETCOLOR register numbers for modes 3 through 8. Because the

COLOR statement behaves somewhat differently in the other modes, they have been left off Table 2.

TABLE 2
CORRESPONDENCES BETWEEN COLOR AND SETCOLOR REGISTERS FOR MODES 3 THROUGH 8

Mode	Setcolor Register	Color Number
3,5,7,15	0	1
	1	2
	2	3
	3	0
4,6	0	1
	4	0
8	1	1
	2	0
14	0	1
	4	0

As should be obvious from that chart, not every register is used in every mode. Table 3 specifically lists which registers are used in which modes, and what they are used for. The foreground register(s) is the register typically used for the plotting color (in pixel modes) or the character color (in text modes). The background register contains the color that you see after you clear the screen, though in most instances this register can also be used for plotting. The border color determines the border around the edges of the display. (In most modes this is controlled by the same register as the background.) Modes 9 through 11 are omitted because they use the registers in an unusual manner. Modes 0 and 8 also deviate from the normal register use pattern, as we shall see later.

TABLE 3
CORRESPONDENCE BETWEEN MODES AND REGISTERS

Mode	Foreground Register	Background Register	Border Register
0,8	1	2	4
1,2	0,1,2,3	4	4
3,5,7,15	0,1,2	4	4
4,6	0	4	4
14	0	4	4

The color number in the SETCOLOR statement must be between 0 and 15. Table 4 lists the colors represented by these numbers.

TABLE 4
RELATIONSHIP BETWEEN COLOR NUMBER AND COLOR

Color Number	Color
0	Gray
1	Gold
2	Orange
3	Red
4	Pink
5	Violet
6	Blue-Purple
7	Blue
8	Light Blue
9	Blue-Green
10	Aqua
11	Green-Blue
12	Green
13	Yellow-Green
14	Orange-Green
15	Orange

The luminance value in the SETCOLOR statement must be an *even* number from 0 to 14. This number determines the brightness with which the color represented by the color number will be displayed. The lowest luminance number, 0, will produce the darkest color; the highest number, 14, the lightest color.

As an example, the statement

```
SETCOLOR 1,10,2
```

will place color 10 (aqua) and luminance 2 (quite dark) in SETCOLOR register 1.

To see how this color register system works on your computer, first push the RESET button. This will assure that your computer is in graphics mode 0 with the default color settings (that is, the color register values automatically determined by the Atari when it is turned on.)

Type

```
SETCOLOR 2,9,4
```

and press RETURN. Although SETCOLOR register 2 is the register that determines the background color in mode 0, nothing visible should happen. This is because color 9, luminance 4 is the normal background color setting for mode 0.

Now type

```
SETCOLOR 2,9,6
```

and press RETURN. The background should lighten slightly. We have just changed the luminance of the current color without changing the color itself. Now type:

```
SETCOLOR 2,1,6
```

and press RETURN. The entire screen should turn gold.

Take some time and experiment with SETCOLOR register 2 in mode 0. Set the register to any different combinations of color and luminance (within the legal ranges) that come to mind. Notice the results. Check them with what you see in Table 2. Be aware that colors have a tendency to vary between different television sets and monitors.

PLOTTING IN COLOR

All done? OK, now let's do some color plotting. We'll use our old friend mode 7. Type

```
GRAPHICS 7
```

and press RETURN. As in the last chapter, we are going to plot a pixel directly in the middle of the display. However, this time we want to plot the pixel in red. Can you figure out the sequence of steps necessary to do this?

Before we give any commands to the computer, we must consult Table 4, which tells us that the color number for red is 3. Looking at Table 3, we see that registers 0, 1, and 2 can be used for foreground plotting. We'll arbitrarily choose register 1. To get a particularly vivid shade of red, we'll use a luminance number of 0. If you've been paying attention, you'll recall that we assign a color of 3 and a luminance of 0 to register 1 with the statement

```
SETCOLOR 1,3,0
```

Type this statement in the text window and press RETURN. You'll notice a rather odd change in the colors within the text window when this command is executed; ignore it. Now, to

establish that we will be plotting with SETCOLOR register 1, we consult Table 2. It tells us to use COLOR number 2.

We now type:

```
COLOR 2
```

and press RETURN. Once we have established the plotting color, we can go ahead and plot the pixel like this:

```
PLOT 80,42
```

Type this command and press RETURN. The pixel will appear in the middle of the display.

Or will it? Actually, there doesn't seem to be anything on the display. The reason is that the dark red we have chosen to plot in is too much like the naturally black background color of mode 7. We must change the background in order to make it visible. The background SETCOLOR register for mode 7 is 4. We can change it to a light orange background by typing

```
SETCOLOR 4,15,2
```

Ah! There's our red pixel, directly in the middle of the display. This should prove the point that the *combination* of background and foreground colors that we choose for the Atari screen is as important as the individual colors themselves. Some combinations work better together than others.

If you want another example of this, try typing

```
SETCOLOR 4,0,16
```

and press RETURN. This sets the background to a dazzling shade of white, so dazzling, in fact, that you may have trouble making out the contents of the text window in the glare. Furthermore, our red dot seems black by contrast. Not exactly what we had intended.

If you follow the information in the tables, you should have no trouble using the color registers in modes 3 through 8 and 14 through 15. Remember that in all but one of these modes, SETCOLOR register 4 is used to control the color of the background. In mode 8, which behaves a little differently from the others, register 2 controls the background color. This is also true of mode 0, which resembles mode 8 in many ways, except that it is a text mode instead of a pixel mode. In fact, the size of the individual pixels in mode 0 is precisely the same as the size of the pixels in mode 8; only, mode 0 does not give us control over these pixels,

only over the character patterns that are made up of these pixels. Note also that register 4 is used as the border color in all modes. Since this is the same register used for the background color in certain modes, the background and border will always be the same color in these modes.

To plot in the background color, we always use COLOR 0, no matter what the SETCOLOR register is for background in the current mode. (See Table 2 to verify this.) Why would we want to plot in the background color? Usually, to erase pixels that we've previously plotted in a foreground color, as a way of turning the pixels back "off," in other words.

Certain modes, such as 0, 4, 6, 8, and 14, feature only one foreground color register. These are called the *two-color modes*, because they allow us to place only two colors on the display at the same time—the foreground color and the background color. Other color registers are ignored. (Modes 0 and 8 also allow a separate border color.) Several additional modes feature three foreground color registers. These are called the *four-color modes*, because they allow us to place four colors on the display at the same time—the three foreground colors and the background color.

The foreground color in modes 0 and 8 is handled somewhat differently from the other modes. Once we have set the color and luminance of the background register using the SETCOLOR statement, we cannot reset the color of the foreground register; it must always be the same color as the background. We can control only the luminance of the foreground register. Thus, the pixels that we plot in mode 8 and the characters that we print in mode 0 will always be the same color as their backgrounds, but they should have different luminances.

For instance, if we set the background color to violet by typing

```
SETCOLOR 2,5,0
```

the Atari will automatically set the foreground register (SETCOLOR register 1) to the same color.

We can use the SETCOLOR statement to change the luminance of the foreground, however, so that the characters will contrast properly with the background. This is done with a statement like this:

```
SETCOLOR 1,5,10
```

Note that we have used the same color as in the SETCOLOR statement: color 5, or purple. Even if we had attempted to use a

different color, however, it would have been ignored by the OS.

For instance, the statement

```
SETCOLOR 1,9,10
```

would have had exactly the same effect. When we set the foreground register in mode 0, the OS pays attention only to the register and luminance number; the color number is meaningless.

THE GTIA MODES

So far we have scarcely mentioned GTIA modes 9 through 11. There is a reason for this. These modes behave somewhat differently from the way in which the other modes behave.

In each of these modes, the display resolution is 80 X 192—that is, the pixel matrix is made up of 80 columns and 192 rows. The difference between these modes is in the number of colors that we can place on the display at one time.

In mode 9, we can have one color, but we can display it at sixteen different luminances.

In mode 11, we can have sixteen different colors, but we must display them all at the same luminance.

In mode 10, we can have eight different color/luminance combinations.

First, we'll examine mode 9, where we can have one color in sixteen different luminances. We set the single color of mode 9 using SETCOLOR register 4. This register sets both the background color—the normal function of register 4—and the foreground colors. It is the only register that we use in this mode. The syntax for SETCOLOR in mode 9 is

```
SETCOLOR 4, color, background-luminance
```

where "color" is the single color that we will work with in mode 9 and "background-luminance" is the luminance that we wish to assign to the background.

For example, the statement

```
SETCOLOR 4,1,2
```

sets register 4 to color 1 (gold) at luminance 2. Color 1 now becomes the only color that we may use in mode 9. (We can issue another SETCOLOR statement, of course, but that will change all the colors on the display.) The luminance number will set the luminance of the background.

The COLOR statement is used in an unusual fashion in mode 9. Rather than determining the color in which we will be plotting, it determines the luminance. The syntax for the COLOR statement in mode 9 is

COLOR luminance

where "luminance" is (as before) an even number from 0 to 14.

For instance, if we wish to draw a light aqua line on a dark aqua background, we could use this program:

```
10 GRAPHICS 9
20 SETCOLOR 4,10,0
30 COLOR 8
40 PLOT 7,10
50 DRAWTO 65,167
60 GOTO 60
```

Line 10 puts us in mode 9. Line 20 establishes the mode 9 color as 10 (aqua) and the background luminance as 0 (very dark). Line 30 establishes the plotting luminance as 8 (much lighter). Line 40 starts the line, and line 50 finishes. Line 60 establishes an infinite loop to prevent the OS from returning us to mode 0. (Remember that no text window is available in mode 9.)

To prove that sixteen luminances are indeed available in mode 9, here is a program that prints all of them on the display at one time, then cycles through the sixteen different colors. Notice the subtle, almost three-dimensional shading effect that can be created with this wide range of luminances:

```
10 GRAPHICS 9
20 SETCOLOR 4,0,0
30 FOR I = 0 TO 4 : REM REPEAT 4 TIMES
40 FOR J = 0 TO 15 : REM ALL 16 LUMINANCES
50 COLOR J
60 PLOT I*16+J, 0 : REM DRAW A LINE
70 DRAWTO I*16+J, 191 : REM IN CURRENT LUMINANCE
80 NEXT J
90 NEXT I
100 FOR I = 0 TO 15 : REM CYCLE THROUGH ALL 16 COLORS
110 SETCOLOR 4,I,0
120 FOR J = 1 TO 500 : REM BRIEF DELAY
130 NEXT J
140 NEXT I
150 GOTO 100 : REM DO IT UNTIL SOMEBODY HITS BREAK
```

Save this program on disk or tape; we'll be coming back to it in a moment.

THE RAINBOW MODE

Much as mode 9 allows us to use sixteen luminances in the same color, mode 11 allows us to use sixteen different colors, but they must all be at the same luninance. Once again, we establish this luminance with the SETCOLOR statement, using register 4, with the syntax:

```
SETCOLOR 4, background-color, luminance
```

For instance, the statement

```
SETCOLOR 4,15,4
```

tells the OS that we wish to work with a mode 11 luminance of 4, and that the background should be set to color 15 (orange). We then select the colors for plotting with the COLOR statement. In mode 11 the syntax for the COLOR statement is

```
COLOR color
```

where "color" is a number from 1 to 15 that represents the color we wish to plot with. We have no further control over the luminance.

For instance, if we wish to draw an orange square on a blue-green background, both at luminance 8, we could write

```
10 GRAPHICS 11
20 SETCOLOR 4,9,8
30 COLOR 15
40 PLOT 20,40
50 DRAWTO 60,40
60 DRAWTO 60,160
70 DRAWTO 20,160
80 DRAWTO 20,40
90 GOTO 90
```

Line 10 puts us in mode 11. Line 20 establishes the luminance as 8 and sets the background color to 9 (blue-green). Line 30 establishes the plotting color as 15 (orange). Lines 40 through 80 draw the square, and line 90 loops until we press BREAK or RESET.

To see all sixteen colors on the display at one time in mode 11, make the following changes to the program that we used to demonstrate the sixteen luminances in mode 9:

```
10 GRAPHICS 11
40 FOR J = 0 TO 15 : REM ALL 15 PLOTTING COLORS
100 FOR I = 0 TO 14 STEP 2
110 SETCOLOR 4,0,I
```

As this program proceeds, you will notice several dark bands extending vertically across the display. These bands are in the background color, which is the sixteenth color available in mode 11. It is the only one of the sixteen colors not affected by changes in luminance, always remaining at a luminance of 0.

THE MOST VERSATILE MODE

Mode 10 is in some ways the most atypical of Atari graphics modes, but it may also be the most versatile. It allows us to display eight different color/luminance combinations. In mode 10, the COLOR statement functions in its normal capacity, as a specifier for the color register. However, it now can specify any register from 0 to 7, to accommodate the wide range of colors available in this mode. You may recall an earlier statement that there are only five Atari color registers. This is true, except in mode 10. However, because the SETCOLOR statement will function with only five registers—SETCOLOR registers 0 through 4—it is necessary to find another way to place values in the remaining registers.

Although we can use SETCOLOR to set five out of the eight color registers used in mode 10, it is sometimes best to avoid the use of SETCOLOR at all, as there is another method we can use to set all eight registers: the POKE command.

The POKE command alters the contents of the Atari's memory. Since the color registers are locations in memory, we can use the POKE command to change them much as we use the SETCOLOR command. Here is the syntax for using the POKE command to set a color register in mode 10:

```
POKE register+704, color*16+luminance
```

where "register" is the register number that would be used in the corresponding COLOR statement and "color" and "luminance" are the numbers used to specify (what else?) the color and luminance. The register number must be in the range 0 to 7. (If you should exceed this range, BASIC won't tell you that you've made an error, but you could destroy data and programs stored in the computer's memory, though not damaging the computer itself.)

For instance, if we wish to set register 6 to color 4 (pink) at a luminance of 12, we would write

```
POKE 6+704, 4*16+12
```

We could also write this as

```
POKE 710, 76
```

but the former version is clearer to the initiated reader, and is easier to modify should we change our mind about the color and/or luminance we wish to place in that register. To plot in the color that we stored in register 6, we would, of course, write

```
COLOR 6
```

and begin plotting.

Here is a program that uses the RND(1) function to place a series of randomly colored stripes on the display, then randomly alters the values in the color registers to produce a shifting color effect:

```
10 GRAPHICS 10
20 FOR I = 0 TO 7
30 POKE 704+I, INT(RND(1)*16)*16+INT(RND(1)*16)
40 NEXT I
50 FOR I = 0 TO 79
60 COLOR INT(RND(1)*8)
70 PLOT I,0
80 DRAWTO I,191
90 NEXT I
100 POKE 704 + INT(RND(1)*8), INT(RND(1)*16)*16+INT(RND(1)*16)
110 GOTO 100
```

Don't stare at the screen for *too* long; the effect is hypnotic.

TEXT PLOTTING

The COLOR statement, as well as several of the graphics commands we studied in our discussion of the bitmap modes, can also be used with text graphics, but to somewhat different effect. For instance, it is possible to PLOT on the display while in modes 0, 1, and 2. However, instead of plotting pixels, we plot text characters. And the COLOR statement determines which characters we PLOT with.

For instance, we saw earlier in this chapter that the SETCOLOR statement in mode 0, when used with register 2, controls both the foreground and background colors; as a result, the SETCOLOR statement, used with register 1, controls only the luminance of the characters placed on the text screen. Thus, it would hardly be necessary to use COLOR to determine color, when all of the colors are predetermined.

However, we can combine the COLOR and PLOT statements to place individual characters at specified coordinates on the mode 0 display, just as we use them in the pixel modes to place colored pixels at specified coordinates. The COLOR statement in mode 0 has this syntax:

```
COLOR atascii-code
```

where "atascii-code" is the ATASCII code number of the character we wish to place on the display. The PLOT statement can then be used to place the character on the display, using coordinate positions within the twenty-four rows of forty characters, just as we would use pixel coordinates to plot in a pixel mode.

For instance, if we wish to place the letter A in the third column of the twentieth row, we could write

```
10 GRAPHICS 0 : REM CLEAR THE DISPLAY
20 COLOR ASC("A") : REM GET ATASCII FOR "A"
30 PLOT 3,20 : REM PUT IT ON THE DISPLAY
40 GOTO 40 : REM HOLD IT THERE
```

RUN this program. Sure enough, the letter A appears in row 3 of column 20. (Bear in mind that the display starts with row 0 and column 0.) But what are those other two blocks of color spoiling up our otherwise neat display?

Those two unsightly blocks are images of the Atari cursor, which doesn't always turn itself off when it should. How do we get rid of these images? There is a special instruction that we can use to turn off the cursor when it finds its way into modes where it does not belong. It is

```
POKE 752,1
```

Add this statement to the above program as line 15, like this:

```
15 POKE 742,1
```

and RUN the program again. Now, one of the two cursor images has been removed—but one remains. Getting rid of it is tricky.

One way is to clear the screen forcibly *after* entering mode 0. (Though the OS performs an automatic screen clear when the GRAPHICS 0 statement is executed, the cursor sometimes finds its way through subsequently.) We can forcibly clear the screen with the statement

```
PRINT CHR$(125);
```

CHR$(125) is the ATASCII code produced by the CONTROL-"<" (or CLEAR) key. It can also be typed, within quotes, as ESC CONTROL-"<". To add this statement to the program, write

```
17 ? CHR$(125);
```

RUN the program again. Now both unwanted cursor images are removed.

LINES OF TEXT

You might wonder what good it does us to PLOT letters in the middle of the display. Admittedly, this application is of limited utility. However, the DRAWTO statement can be used in text mode to plot entire lines of text characters onto the display, from one specified set of coordinates to another, a capability that can be used quite effectively with graphics characters. For instance, here is a program that uses the inverse space character (ATASCII 160) to draw a border around the display. Ordinarily, we could produce such an effect with a set of FOR-NEXT loops, a process that would be awkward and slow. Here we accomplish it with a PLOT and four DRAWTOs:

```
10 GRAPHICS 0
20 COLOR 160
30 PLOT 0,0
40 DRAWTO 39,0
50 DRAWTO 39,23
60 DRAWTO 0,23
70 DRAWTO 0,0
80 GOTO 80
```

To reduce the dimensions of the borders, simply change the coordinates. And, of course, characters other than reverse spaces can be used for plotting, and in patterns other than rectangles. Use your imagination. PLOT and DRAWTO are powerful methods of getting a lot of characters onto the mode 0 screen, very quickly, in attractive patterns.

The COLOR, PLOT, and DRAWTO statements work in much the same fashion in text modes 1 and 2, though the dimensions of the display have changed: twenty columns across in modes 1 and 2, and twelve rows in mode 2. However, these modes allow us to place more colors on the display at one time than does mode 0. You might wonder how we can specify which of these colors we wish to use, if the COLOR statement is used to determine characters.

Good question. And it has a rather complicated answer. In modes 1 and 2 the characters themselves determine the colors in which they will be printed. This is a concept more easily explained if you understand binary numbers, a subject that will be touched on in the section on advanced graphics. For now, here is a chart that tells us which characters will be displayed in which colors. Characters are given by ATASCII code. Equivalent color registers are also given, along with a notation indicating whether the character displayed in that register's color will be displayed normally or reversed.

ATASCII VALUES	COLOR REGISTER
32 to 90	Normal: 0
160 to 218	Reverse: 2
91 to 122	Normal: 1
225 to 250	Reverse: 3

Hence, a character with an ATASCII value of 63 will be displayed in normal video with the color in register 0. A character with an ATASCII value of 240 will be printed in reverse video with the color in register 3.

A major drawback of text modes 1 and 2 is that the actual character printed will always be printed as though it were a character in the range 32 to 90; only the color will vary if the ATASCII value of the character falls outside of this range. CHR$(225), for instance, will be printed as the uppercase letter A, as will CHR$(65), but the former will be reversed and in one color while the latter will be normal and in another. This is why we are limited to only sixty-four different characters. (This may not quite add up, but bear in mind that some of the characters in these ranges are control characters, which can be printed only using the ESC method, and are not included in the chart.)

Here, for instance, is a program that prints a mode 2 sentence in two different colors:

```
10 GRAPHICS 2
20 SETCOLOR 0,15,2
30 SETCOLOR 1,0,6
40 PRINT #6; "Now iS tHe TiMe FoR aLl GoOd MeN aNd        WoMeN
tO cOmE tO tHeAiD oF tHeIr PaRtY."
```

The lowercase letters, belonging to a different ATASCII range from their uppercase counterparts, print in a different color, although all characters will print as though they were uppercase. (We can't print lowercase in modes 1 and 2, because it

falls out of the allowed range.) To add to the variety of colors, you might try throwing in some reversed characters, both uppercase and lowercase. (Press the reverse key in the lower right-hand corner of the keyboard to get reversed characters.) This will bring in yet another range of ATASCII codes and can result in as much as four different colors being displayed at one time.

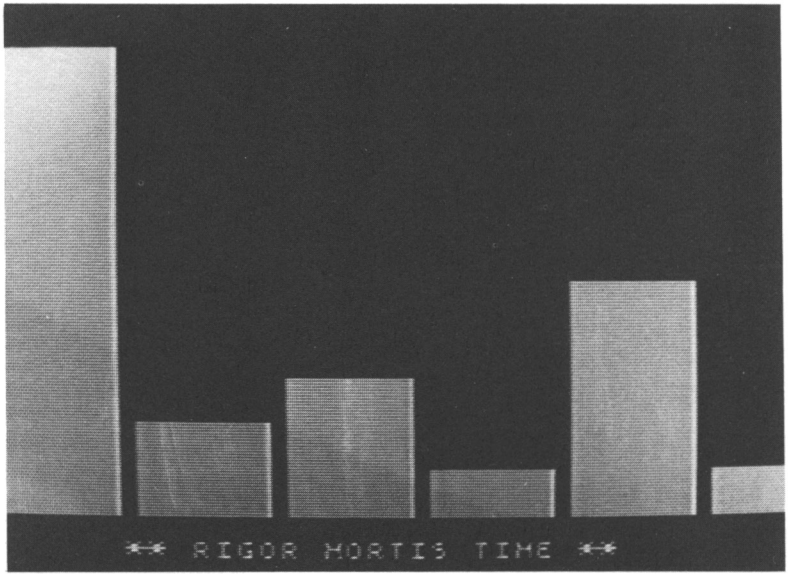
THE BAR CHART PROGRAM

Thus far, you may be tempted to believe that the primary use of colored graphics is to place pretty patterns on the display. You would be excused for thinking this, but it isn't true.

There are some very serious uses for colored graphics, for instance, the creation of bar charts.

A bar chart is a graphic representation of numeric data that allows immediate visual comparison of related numbers. It is used commonly in business to illustrate such difficult to grasp statistics as relative sales figures and economic growth.

A bar is simply a filled rectangle, like the ones we drew in Chapter Two, the size of which represents an important quantity. By comparing the size of several such bars representing related quantities, we can have an immediate, visceral sense of which quantity is larger and which is smaller. And if we have several



Bar chart created using the program in this chapter

bars, representing the same quantity at different points in time, we can instantly see if the quantity is rising or falling. Such charts give us a much more immediate perception of certain trends.

Here is a bar chart program for your Atari, written in mode 8. Mode 8 was chosen because its high resolution gives us the ability to display small differences in size between the quantities represented by two bars. Of course, owners of GTIA Ataris should recognize that we can get a similar fine discrimination in the height of a bar using modes 9, 10, or 11, each of which has the same vertical resolution (192 pixels) as mode 8. We would then have the advantage of being able to use several different colored bars on the display at one time. Nonetheless, we have written the program in mode 8 so that even those without GTIA computers can make use of it. It is not a finished, polished program, but it could provide the basis for one. You are encouraged to customize and expand it, to optimize it for any needs you may have for such a program. We will offer some suggestions for customization at the end of the chapter.

Here's the program:

```
5 REM *** BAR CHART PROGRAM ***
6 REM
10 DIM LENGTH(32), TITLE$(36)
15 OPEN #1, 4, 0, "K:"
20 HIGHBAR = 1
30 GRAPHICS 1
32 SETCOLOR 1,0,8
34 SETCOLOR 2,0,0
36 SETCOLOR 4,0,0
40 POSITION 3,9
50 ? #6; "BAR CHART DEMO"
60 ? "TITLE OF CHART";
70 INPUT TITLE$
80 ? "NUMBER OF BARS DESIRED"
90 ? "(2 - 32)"
100 INPUT NUMBER
110 IF (NUMBER > 32) OR (NUMBER < 2) THEN 80
120 FOR BAR = 0 TO NUMBER - 1
130 ? "LENGTH OF BAR NUMBER "; BAR + 1;
140 ? " IN STANDARD UNITS";
150 INPUT LNGTH
160 IF LNGTH < 1 THEN ? "BAD LENGTH" : GOTO 130
170 IF HIGHBAR < LNGTH THEN HIGHBAR = LNGTH
180 LENGTH (BAR) = LNGTH
190 NEXT BAR
200 GRAPHICS 8
210 SETCOLOR 2,0,0
```

```
215 COLOR 1
220 WIDTH = 320/NUMBER
230 UNIT = 158/HIGHBAR
240 FOR BAR = 0 TO NUMBER-1
250 PLOT BAR*WIDTH+WIDTH-5, 159
260 DRAWTO BAR*WIDTH+WIDTH-5, 159-UNIT*LENGTH(BAR)
270 DRAWTO BAR*WIDTH, 159-UNIT*LENGTH(BAR)
280 POSITION BAR*WIDTH, 159
290 POKE 765,1
300 XIO 18, #6, 0, 0, "S:"
310 NEXT BAR
320 POKE 752,1
330 ?
340 FOR I = 1 TO 18 - (LEN(TITLE$)/2 + 3)
350 ? " ";
360 NEXT I
370 PRINT "** ";TITLE$;" **"
380 GET #1, K
390 GOTO 20
```

This listing is complex enough that we should go over it line by line. However, you'll probably want to type the program first and RUN it, to see what it does. Initially, the program will prompt you to enter the name of the chart. Secondly, it will ask how many bars you wish to show on the display. Finally, it will ask you, one at a time, what length each bar should have in "standard units." By standard units, we simply mean that you should have some unit of measurement in mind for the bars and use it consistently on each bar.

For instance, if you are graphing the population of a country, you may want to use millions of people as the unit, while if you are graphing the sales of a small business, you might want to use thousands of dollars as the unit. There is virtually no limit to the range of numbers that you can use for input, except that negative numbers are not allowed. The program will take the tallest bar in the chart and "normalize" it to the height of the display, then adjust all shorter bars accordingly. Thus, only the relative sizes of the bars matter, not the specific unit that you use. Once the bars have been drawn and you are tired of looking at them, press any key and you will be prompted to design another chart.

Here's a line-by-line description of the program:

Line 10—Dimensions the array (LENGTH) that will hold the lengths of the bars and the string TITLE$ that will hold the title of the chart.

Line 15—Opens the keyboard as a direct input channel. On the Atari, it is necessary to do this before we can use the GET state-

ment to obtain individual characters typed at the keyboard, as we will do later in the program.
Line 20—Initializes variable HIGHBAR, which is used to hold the height of the highest bar in the chart. It is important not to initialize this variable to 0, because under certain circumstances it might provoke a divide-by-zero error later in the program.
Line 30—Puts the display in text mode 1, for the program title.
Lines 32-36—Set the color registers for the opening title.
Line 40—Sets the cursor at the proper position to print the title. We'll look at the POSITION command in the next chapter.
Line 50—Prints title.
Line 60—Prompts user (in text window) for title of chart.
Line 70—Inputs title into TITLE$.
Lines 80-90—Prompt for number of bars in chart.
Line 100—Inputs number of bars.
Line 110—Checks for valid range.
Line 120—Starts loop to get lengths of all bars.
Lines 130-140—Prompt for length of each bar.
Line 150—Inputs length of current bar.
Line 160—Rejects negative lengths.
Line 170—Sets HIGHBAR equal to length of current bar, if current bar is larger then the length of the previous largest bar.
Line 180—Sets LENGTH(BAR), which holds the length of the current bar, equal to LNGTH. (Atari BASIC will not allow direct INPUT of values to array elements.)
Line 190—If more bars, get'em.
Line 200—Sets GRAPHICS 8.
Line 210—Sets background color.
Line 220—Calculates width of each bar.
Line 230—Calculates standard unit in pixels.
Line 240—Starts loop that will draw each bar.
Line 250—Plots lower right-hand corner of current bar.
Line 260—Draws line to upper right-hand corner of current bar.
Line 270—Draws line to upper left-hand corner of current bar.
Line 280—Establishes position of lower left-hand corner.
Line 290—Establishes fill color.
Line 300—Fills the bar with color.
Line 310—If more bars, draw'em.
Line 320—Kills the cursor.
Line 330—Prints a blank line in text window.
Line 340—Centers title in text window.
Line 350—Blank spaces from margin to text.
Line 360—Ready for title.
Line 370—Print title.
Line 380—Wait for a key to be pressed. (This is why we opened the keyboard for input in line 15).
Line 390—Do it all over again.

Suggested Projects

1. Although the bar chart program listed in this chapter may lack the bells and whistles of a full-fledged professional bar chart program, it offers the essential core of such a program. Feel free to customize it in any way you would like. For instance, you might want to add a feature that will print the size of each bar, in standard units, directly under that bar; similarly, you might want to give each bar a name (although this will somewhat limit the number of bars that you can put on the display, since each must be wide enough to have its name printed underneath it).

Once you have created a thoroughly presentable bar chart program, you can make two more changes:

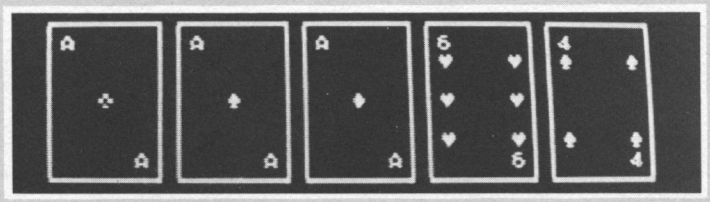
a. Alter the program to read the number of bars, length of each bar, etc., from a file on disk or tape. (You'll need to write a second program that will store the appropriate values in the file.) Then rewrite the program so that it will automatically load the data and display a series of charts for an audience.

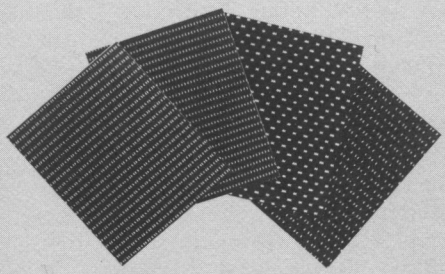
b. If you have a printer that will print graphics, add a routine that will print a hard copy of each bar chart

in a format similar to that used on the display. Read the printer manual to learn the appropriate instructions. If your printer will print color graphics, change the routine to print bars in several different colors. (If you have a GTIA Atari, you might rewrite the main bar chart routine to print a multicolored chart in GRAPHICS 10.)

Once you have customized the program, put on a demonstration for your family and friends, or even for your entire class. Use data in your charts that will be meaningful to your audience: the scores of a school team over their last several games, the amount of money that you (or someone else) have earned in a part-time job, the changes in temperature that you have recorded over the previous month, etc.

2. Write a program using mode 0 that draws five playing cards on the screen, side by side. Use the PLOT and DRAWTO commands to draw the borders of the cards, using graphics characters. Find characters in the Atari character set that can be used to represent the card suits: hearts, spades, clubs, diamonds. See if you can use this graphic effect to create a program that will play a card game with the user.





Project 2 asks you to design five playing cards like these. You also could design the backs of your computer cards.

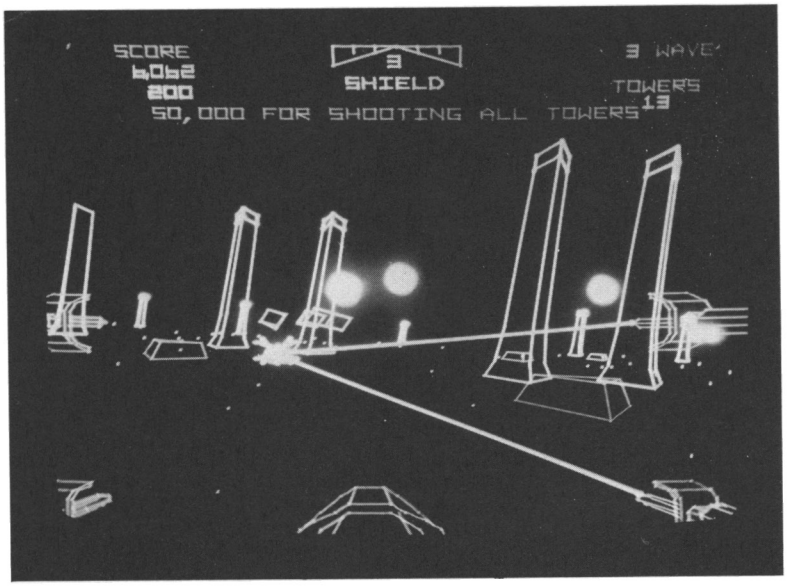
4
ANIMATION

In Chapter Two, we saw a program that used the PLOT statement and a FOR-NEXT loop to draw a line across the display. As it turned out, this program was quite unnecessary, because the DRAWTO statement can accomplish the same thing with less programming effort and at greater speeds. Nonetheless, the technique that we used to draw that line, with a single modification, can be used to create an exciting effect: *video animation.* That is, we can use it to make things move.

Animation is an important aspect of computer graphics. Computer animation makes arcade game programs possible. Computers are used in Hollywood to create special effects for movies like *Tron* and *The Last Starfighter.* And the Atari computer is capable of very sophisticated animation.

We will start, however, with some very unsophisticated animation. Here is a program, similar to the one in Chapter Two, that draws a line across the display:

```
10 GRAPHICS 3 + 16
20 SETCOLOR 4,12,0
30 SETCOLOR 0,13,8
40 COLOR 1
50 FOR I = 0 TO 39
60 PLOT I,12
```



Graphics developed on the Atari for the movie Star Wars

```
70 NEXT I
80 GOTO 80
```

The technique we use for drawing the line is simple. We PLOT a sequence of pixels on the display, repeatedly increasing the value of the horizontal coordinate by 1 until we have stepped it through the full range of horizontal coordinates (0 to 39) available in mode 3. At the same time, we leave the vertical coordinate unchanged. The result is a sequence of pixels all with the same vertical coordinate but consecutive horizontal coordinates, i.e., a horizontal line.

We can also create a vertical line like this:

```
10 GRAPHICS 3 + 16
20 SETCOLOR 4,12,0
30 SETCOLOR 0,13,8
40 COLOR 1
50 FOR I = 0 TO 23
60 PLOT 20,I
70 NEXT I
80 GOTO 80
```

The only difference between this and the previous program is

that we now step the vertical coordinate through its entire range of values while leaving the horizontal coordinate unchanged.

THE MOVING PIXEL

By making one small change to these programs, we can alter the effect considerably. Suppose that we erase each pixel immediately after we draw it. What good would this do us? Try this program and see:

```
10 GRAPHICS 3 + 16
20 SETCOLOR 4,12,0
30 SETCOLOR 0,13,8
40 FOR X = 0 TO 39 : REM STEP THROUGH HORIZONTAL
COORDINATES
50 COLOR 1 : REM USE COLOR 1
60 PLOT I,12 : REM TO PLOT PIXEL
70 FOR J = 1 TO 50 : REM DELAY BRIEFLY
80 NEXT J
90 COLOR 0 : REM USE BACKGROUND COLOR
100 PLOT I,12 : REM TO ERASE PIXEL
110 NEXT I : REM GET NEXT HORIZONTAL COORDINATE
120 GOTO 40 : REM REPEAT INDEFINITELY
```

This is essentially our horizontal line drawing program, with a couple of changes. We now erase each pixel before we draw the next. (Actually, we plot the pixel in COLOR 1, then replot it in the background color, COLOR 0, effectively erasing it.) When you run the program, you will no longer see a line drawn across the display. Instead, you will see a single mode 3 pixel float from one side of the display to the other. We have, in effect, animated the pixel.

Notice that there is a delay loop in lines 70 to 80, directly between the instructions that plot the pixel (lines 50 to 60) and the instructions that erase it (lines 90 to 100). This delay loop serves two purposes. The first is to slow down the motion with which the pixel appears to move. The second is to hold the pixel on the display long enough for us to see it clearly before it is erased.

Remove lines 50 to 60 to see what happens. The pixel will move across the display so quickly that it is barely visible, and the interval between the plotting and erasing of the pixel will be so small that it will spend most of its time erased. By selectively varying the limit value of the FOR-NEXT loop in line 70, we can slow down or speed up the pixel. As the program is shown here, we are using a value of 50 for the limit. A larger value will cause the pixel to move more slowly; a smaller value will speed it up.

By making a similar modification to the vertical line drawing program, we can cause the pixel to move from the top of the display to the bottom, like this:

```
10 GRAPHICS 3 + 16
20 SETCOLOR 4,12,0
30 SETCOLOR 0,13,8
40 FOR I = 0 TO 23 : REM STEP THROUGH VERTICAL
COORDINATES
50 COLOR 1 : REM USE COLOR 1
60 PLOT 20,I : REM TO PLOT PIXEL
70 FOR J = 1 TO 50 : REM DELAY BRIEFLY
80 NEXT J
90 COLOR 0 : REM USE BACKGROUND COLOR
100 PLOT 20,I : REM TO ERASE PIXEL
110 NEXT I : REM GET NEXT VERTICAL COORDINATE
120 GOTO 40 : REM REPEAT INDEFINITELY
```

By simultaneously changing both the horizontal and vertical coordinates of a pixel, we can cause the pixel to seem to move diagonally, like this:

```
10 GRAPHICS 3 + 16
20 SETCOLOR 4,12,0
30 SETCOLOR 0,13,8
40 X = 0 : REM INITIALIZE HORIZONTAL COORDINATE
50 FOR Y = 0 TO 23 : REM STEP THROUGH VERTICAL
COORDINATES
60 COLOR 1 : REM USE COLOR 1
70 PLOT X,Y : REM TO PLOT PIXEL
80 FOR I = 1 TO 50 : REM DELAY BRIEFLY
90 NEXT I
100 COLOR 0 : REM USE BACKGROUND COLOR
110 PLOT X,Y : REM TO ERASE PIXEL
120 X = X + 1 : REM GET NEXT HORIZONTAL COORDINATE
130 NEXT Y : REM AND NEXT VERTICAL COORDINATE
140 GOTO 40 : REM REPEAT INDEFINITELY
```

In this example, the value of the vertical coordinate is increased by a FOR-NEXT loop, while the value of the horizontal coordinate is increased by an X = X + 1 statement in line 120. This situation could be reversed, of course, by placing the horizontal coordinate in the FOR-NEXT loop and incrementing the vertical coordinate with Y = Y + 1. To change the angle of the line, we can change the amount that is added to the vertical or horizontal coordinate. However, neither coordinate should ever be increased by more than 1, unless you want the pixel to jump

across more than one coordinate position at a time, producing a less than smooth animation.

THE LOCATE STATEMENT

Once we have animated an object, it is a fairly simple matter to make that object interact with other objects on the display. This is facilitated with the Atari LOCATE statement. LOCATE can be used to examine a given coordinate position on the display to determine if there is an object at that position, so that we can detect interactions between more than one object.

The syntax for LOCATE is

```
LOCATE column,row,numeric-variable
```

"Column" and "row" are arithmetic expressions representing the horizontal and vertical coordinates, respectively, of the position we wish to examine on the display. "Numeric-variable" is a standard Atari numeric variable, a variable that can be set equal to a number. The LOCATE statement assigns a value to this numeric variable. In a text mode, the value assigned to this variable is a number between 0 and 255, indicating the ATASCII value of the character displayed in the indicated position on the screen. In a pixel mode, the value assigned to this variable is a number between 0 and 7, indicating the color register used for the pixel at that position.

For instance, the following program places the words NOW IS THE TIME on the top row of the mode 1 display, determines what character is in the third position of that row using the LOCATE statement, then prints both the ATASCII value of that character and the character itself in the text window:

```
10 GRAPHICS 1
20 PRINT #6; "NOW IS THE TIME"
30 LOCATE 2,0,CHAR
40 PRINT "ATASCII-"; CHAR; " CHARACTER-"; CHR$(CHAR)
```

Of course, when you run this program the ATASCII value printed will be 87 and the character will be W. If you change the sentence printed by line 20, however, this value will change, to whatever character you place in the third position of the line.

OBJECTS IN COLLISION

When we have more than one object on the display and at least one of them is moving, LOCATE can be used to detect collisions

between objects. Here is a program that uses LOCATE to bounce a pixel back and forth between two walls:

```
10 GRAPHICS 3 + 16
20 SETCOLOR 4,0,0
30 SETCOLOR 0,8,8
40 SETCOLOR 1,8,8
50 COLOR 1 : REM COLOR FOR WALL
60 PLOT 10,0 : REM DRAW LEFT WALL
70 DRAWTO 10,23
80 PLOT 30,0 : REM DRAW RIGHT WALL
90 DRAWTO 30,23
100 DIRECTION = 1 : REM HORIZONTAL DIRECTION OF PIXEL
110 HORIZ = 20 : REM INITIAL HORIZONTAL COORDINATE
120 HORIZ=HORIZ+DIRECTION : REM GO TO NEXT HORIZONTAL
POSITION
130 COLOR 2 : REM COLOR OF PIXEL
140 PLOT HORIZ,12 : REM DRAW PIXEL
150 LOCATE HORIZ+DIRECTION, 12, COLOUR : REM CHECK NEXT
POSITION
160 IF COLOUR = 1 THEN DIRECTION = -DIRECTION : REM IF
WALL, THEN REVERSE DIRECTION
170 FOR DELAY = 1 TO 10 : REM DELAY BRIEFLY
180 NEXT DELAY
190 COLOR 0 : REM BACKGROUND COLOR
200 PLOT HORIZ,12 : REM ERASE PIXEL
210 GOTO 120 : REM CONTINUE INDEFINITELY
```

In this program, we no longer use a FOR-NEXT loop to move the pixel. Instead, we place the horizontal coordinate of the pixel in variable HORIZ and add to or subtract from it a value of either 1 or −1 on each pass through the loop, depending on whether we want the pixel to move right or left. This value is contained in variable DIRECTION. If the value of DIRECTION is 1, then the horizontal coordinate will move to the right (i.e., toward higher coordinate values). If the value of DIRECTION is −1, then the horizontal coordinate will move to the left (i.e., toward higher coordinate values).

Line 120 actually adds the value of DIRECTION to HORIZ. Then lines 130 to 140 plot the pixel at HORIZ,12. (The Y coordinate of the pixel is always 12.) Line 150 uses the LOCATE statement to check the value of the *next* position at which the pixel will be plotted—position HORIZ+DIRECTION, 12. Line 160 checks to see if the COLOR register being used at this position is register 1. If so, we can assume that the pixel is about to strike the wall. In this event, the value of DIRECTION is

reversed—if it is 1 then it becomes −1, and if it is −1 it becomes 1. The pixel then appears to move in the opposite direction, as though it had bounced on contact with the wall. The final line of the main loop jumps back to the beginning of the animation process, so the process will continue until either the BREAK or RESET key is pressed.

We can make the animation even more sophisticated by moving the pixel in both the horizontal and vertical directions, and giving it both horizontal and vertical walls to bounce off. Here is a program that bounces our pixel around inside a rectangle:

```
10 GRAPHICS 3 + 16
20 SETCOLOR 4,0,0
30 SETCOLOR 0,8,8
40 SETCOLOR 1,8,8
50 COLOR 1 : REM COLOR FOR RECTANGLE
60 PLOT 0,0 : REM DRAW RECTANGLE
70 DRAWTO 39,0
80 DRAWTO 39,23
90 DRAWTO 0,23
100 DRAWTO 0,0
110 HDIREC = 1 : REM HORIZONTAL DIRECTION
120 VDIREC = 1 : REM VERTICAL DIRECTION
130 HORIZ = 20 : REM INITIAL HORIZONTAL COORDINATE
140 VERTI = 12 : REM INITIAL VERTICAL COORDINATE
150 HORIZ = HORIZ + HDIREC : REM MOVE HORIZONTALLY
160 VERTI = VERTI + VDIREC : REM MOVE VERTICALLY
170 COLOR 2 : REM PIXEL COLOR
180 PLOT HORIZ, VERTI : REM PLOT PIXEL
190 LOCATE HORIZ+HDIREC, VERTI, COLOUR : REM CHECK NEXT
POSITION IN HORIZONTAL DIRECTION
200 IF COLOUR = 1 THEN HDIREC = −HDIREC : REM IF BORDER,
REVERSE HORIZONTAL DIRECTION
210 LOCATE HORIZ, VERTI+VDIREC, COLOUR : REM CHECK NEXT
POSITION IN VERTICAL DIRECTION
220 IF COLOUR = 1 THEN VDIREC = −VDIREC : REM IF BORDER,
REVERSE VERTICAL DIRECTION
230 FOR DELAY = 1 TO 10 : REM DELAY BRIEFLY
240 NEXT DELAY
250 COLOR 0 : REM BACKGROUND COLOR
260 PLOT HORIZ,VERTI : REM ERASE PIXEL
270 GOTO 150
```

The essential principles behind this program are the same as those behind the last, except that the direction in which the pixel moves is now held in two variables. HDIREC contains the direc-

tion of horizontal motion (equivalent to variable DIRECTION in the last program), and VDIREC contains the direction of vertical motion (which was assumed to be 0 in the last program). The current horizontal and vertical coordinates of the pixel are stored in variables HORIZ and VERTI, respectively. To move the pixel, we add the value of HDIREC to HORIZ and VDIREC to VERTI. This is done in lines 150 to 160. The pixel is drawn by lines 170 to 180. Then lines 190 to 200 check to see if we are about to hit the border of the rectangle in the horizontal direction. If so, the value of HDIREC is reversed. Lines 210 to 220 check to see if we are about to hit the border of the rectangle in the vertical direction. If so, the value of VDIREC is reversed. Note that if we are about to hit the border in *both* directions (i.e., if the pixel is headed into a corner), both variables will be reversed, causing the pixel to bounce straight back the way it came. Otherwise, it will bounce at an angle. This adds a realistic, if somewhat idealized, touch to the animation. Run it and see.

READING THE JOYSTICK

We can also add excitement and realism to our animation by allowing the user of our program to control objects on the display. In Atari games, such control is commonly exercised by means of the joystick.

We can "read" the movement of the joystick from within a BASIC program by using the function STICK. This function is always equal to a number indicating the direction in which the joystick is being pushed. We indicate which joystick we wish to read by placing the joystick number in parentheses immediately after the word STICK like this:

```
STICK(1)
```

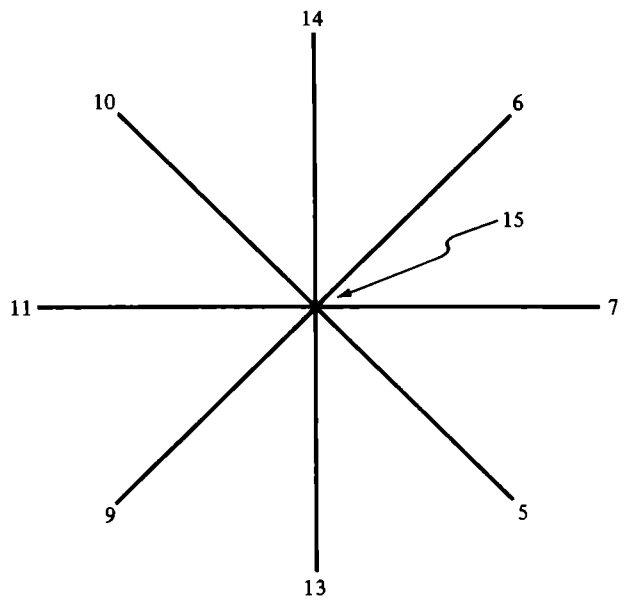
This says that we wish to read joystick number 1. On the pre-XL Ataris, there are four joysticks that we can read, numbered 0 to 3. On the XL Ataris, there are two joysticks, numbered 0 to 1.

We use the STICK function in a numeric expression just as we would use a number or a variable. Typically, we might write:

```
A = STICK(0)
```

This would set variable A equal to the current value of STICK(0). Variable A could then be used to determine the motion of joystick number 0.

Here is a diagram indicating how the value of STICK(X) corresponds to the position of joystick number X:



Values for STICK (X) Based on Joystick Positions

For instance, if STICK(X) is equal to 14, then joystick number X is pointed directly forward. If it is equal to 11, joystick number X is pointed to the left. If it is equal to 15, then joystick number X is centered (i.e., it is not being pointed at all).

ATARI PING-PONG

Here is a program that uses the STICK function to move a pixel back and forth with the motion of joystick number 0:

```
10 GRAPHICS 3 + 16
20 SETCOLOR 4,0,0
30 SETCOLOR 0,8,8
40 DIRECTION = 0 : REM HORIZONTAL DIRECTION
50 HORIZ = 20 : REM HORIZONTAL COORDINATE
60 COLOR 1 : REM COLOR FOR PIXEL
70 HORIZ = HORIZ + DIRECTION : REM MOVE PIXEL
80 PLOT HORIZ,12 : REM DRAW PIXEL
90 DIRECTION = 0 : REM STOP PIXEL MOTION
100 JOY = STICK(0) : REM READ JOYSTICK
110 IF JOY = 11 AND HORIZ>0 THEN DIRECTION = -1 : GOTO 140
: REM IF LEFT, THEN MAKE DIRECTION -1
```

```
120 IF JOY = 7 AND HORIZ<39 THEN DIRECTION = 1 : GOTO 140 :
REM IF RIGHT THEN MAKE DIRECTION 1
130 GOTO 60 : REM IF NO MOTION, REPEAT
140 COLOR 0 : REM ELSE GET BACKGROUND COLOR
150 PLOT HORIZ,12 : REM AND ERASE PIXEL
160 GOTO 60 : REM REPEAT
```

Here is a program that combines the bounce technique described earlier with a joystick reading routine. The result is a one-player Ping-Pong game in graphics mode 5:

```
5 REM *** PING-PONG PROGRAM ***
6 REM
10 DIM A$(5)
20 GOSUB 1000
30 YPAD = 17 : REM Y COORDINATE FOR TOP OF PADDLE
35 GOSUB 3000 : SCORE = 0
40 XBALL = 15 : YBALL = 20 : REM STARTING COORDINATES FOR
BALL
50 LASTX = XBALL : LASTY = YBALL
60 YDIR = 1
70 XDIR = 1
80 A = STICK(1)
90 IF A = 13 AND YPAD < 33 THEN COLOR 0 : PLOT 3,YPAD :
COLOR 2 : PLOT 3,YPAD+6 : YPAD = YPAD+1
100 IF A = 14 AND YPAD > 1 THEN COLOR 0 : PLOT 3,YPAD+5 :
COLOR 2 : PLOT 3,YPAD-1 : YPAD = YPAD-1
110 COLOR 0 : PLOT LASTX,LASTY
120 COLOR 2 : PLOT XBALL,YBALL
130 LASTX = XBALL : LASTY = YBALL
140 LOCATE XBALL +XDIR,YBALL,Z : IF Z = 2 THEN XDIR = -XDIR
150 LOCATE XBALL,YBALL+YDIR,Z : IF Z = 2 THEN YDIR = -YDIR
160 XBALL = XBALL+XDIR : YBALL = YBALL + YDIR
170 IF XBALL = 5 AND XDIR = 1 THEN SCORE = SCORE + 1
175 IF XBALL > 1 THEN 80
180 PRINT "GAME OVER!"
185 ? "YOUR SCORE IS ";SCORE
190 ? "PLAY AGAIN (Y/N)"; : INPUT A$ : IF A$ = "Y" THEN ?
CHR$(125) : GOTO 20
200 GRAPHICS 0 : END
1000 GRAPHICS 5 : POKE 752,1 : SETCOLOR 0,1,0
1010 SETCOLOR 1,10,5
1020 SETCOLOR 2,0,0
1030 SETCOLOR 4,0,0
2000 COLOR 2
2010 PLOT 2,0
2020 DRAWTO 75,0
```

```
2030 DRAWTO 75,39
2040 DRAWTO 2,39
2050 RETURN
3000 COLOR 2 : PLOT 3,YPAD : DRAWTO 3,YPAD+5
3010 RETURN
```

To play the game, plug the joystick into port number 2 and RUN the program. The "paddle" will appear toward the left-hand side of the display; the ball will materialize somewhat to its right. The ball will bounce off the walls bordering the playfield, then return in the direction of the paddle. You must hit the ball with the paddle each time it bounces in your direction, or it will disappear off the edge of the display and the game will end. When this happens, you will be given a score, based on the number of times you have successfully returned the ball.

Here is a line-by-line explanation of the program:

Line 10—Dimensions A$ for later use.
Line 20—Calls initialization subroutine to draw display and set color registers.
Line 30—Establishes location of paddle in variable YPAD, which is set equal to the Y coordinate of the upper end of the paddle.
Line 35—Draws paddle. Sets SCORE equal to 0.
Line 40—Sets XBALL and YBALL equal to the horizontal and vertical coordinates of the ball, respectively.
Line 50—Sets LASTX and LASTY equal to the most recent positions of XBALL and YBALL.
Line 60—Sets YDIR equal to the vertical direction of the ball.
Line 70—Sets XDIR equal to the horizontal direction of the ball.
Line 80—Reads joystick into variable A.
Line 90—Checks to see if joystick is pointed downward. If so, and if the paddle is not yet at the bottom of the playfield (YPAD = 33), then it moves the paddle down one position.
Line 100—Checks to see if the joystick is pointed upward. If so, and if the paddle is not yet at the top of the playfield (YPAD = 1), then it moves the paddle up one position.
Line 110—Erases image of ball at LASTX and LASTY coordinates.
Line 120—Draws new image of ball at XBALL and YBALL coordinates.
Line 130—Sets LASTX and LASTY to the most recent positions of XBALL and YBALL.
Line 140—Checks next ball coordinate in horizontal direction. If it is either wall or paddle (i.e., if LOCATE indicates that color register 2 is in use), the ball bounces in the horizontal direction.

Line 150—Checks next ball coordinate in vertical direction. If it is either wall or paddle, the ball bounces in the vertical direction.
Line 160—Moves the ball in the horizontal and vertical directions.
Line 170—Checks to see if ball has been hit by paddle (the only situation in which XBALL = 5 and XDIR = 1 will both be true). If so, adds one to SCORE.
Line 175—Verifies that ball has not run off playfield. If so, repeat the main loop (lines 80 to 175).
Line 180—If ball is off playfield, game is ended.
Line 185—Score is announced.
Line 190—Player is offered a chance to play again. Response is input to variable A$. If the response is Y then display is cleared and game begun again.
Line 200—Otherwise, program terminates. Graphics 0 is restored.
Lines 1000 to 2050—Subroutine to set colors and graphics mode (mode 5), and to draw walls around playfield.
Lines 3000 and 3010—Subroutine to draw the paddle.

Although this is not the fastest moving, most challenging game you will ever play on your Atari, it illustrates the principles on which many animation games are based. The player controls the movement of at least one animated object with a joystick, while objects on the display interact in a series of collisions, which creates changes in the motions of those objects. Points are awarded for certain beneficial interactions, and a predetermined condition terminates play.

THE SOUND STATEMENT

One additional feature that might make our game more interesting would be sound effects. Atari BASIC generates sound effects with the SOUND statement. The syntax for the SOUND statement is

```
SOUND voice,pitch,distortion,volume
```

Voice is a numeric expression indicating which of the Atari's four sound generators (or "voices") you wish to use to generate the effect. (You may, if you wish, use more than one of these voices at a time, by activating each in a separate sound statement.) The voices are numbered 0 through 3; any of the four will produce identical results.

Pitch is a numeric expression evaluating to a number from 0 to 255, where the lower numbers produce higher-pitched sounds. *Distortion* is a number from 0 to 14 establishing the distortion of

the tone. Volume is a number from 0 to 15 that establishes the loudness of the sound. High numbers are louder than low numbers. Volume 0 produces no sound at all.

Since the subject of this book is graphics and not sound, we will not further explain the theory of the SOUND statement. However, the interested reader is invited to experiment with its use. Note that once you have turned on a sound via the sound statement, you can turn it back off only by setting the volume back to 0, like this:

```
SOUND 1,A,B,0
```

where A and B are any values in the appropriate range for the pitch and distortion expression. The 0 volume will turn off the sound from voice 1 until another sound statement is executed.

To add sound to our Ping-Pong program, make the following changes:

```
140 LOCATE XBALL+XDIR, YBALL, Z : IF Z = 2 THEN XDIR =
-XDIR : GOSUB 4000
150 LOCATE XBALL, YBALL+YDIR, Z : IF Z = 2 THEN YDIR =
-YDIR : GOSUB 4000
4000 SOUND 1, 100, 10, 15
4010 FOR I = 1 TO 2 : NEXT I
4020 SOUND 1,100,1,0
4030 RETURN
```

The subroutine at lines 4000 to 4030 turns on voice 1 and leaves it on just long enough to produce a bouncing noise when the ball strikes the border or the paddle. Try it and see.

Suggested Projects

1. We can produce animation with PRINT graphics as well as PLOT graphics, the primary difference being that with PRINT graphics we change the coordinates of a POSITION statement, rather than the coordinates of a PLOT statement. Rewrite the Ping-Pong program in this chapter using PRINT graphics rather than PLOT graphics. Use a graphics character for the ball. Remember that the LOCATE statement functions somewhat differently in text mode: it returns the ATASCII code number of the character in the specified position, rather than the color register in use.

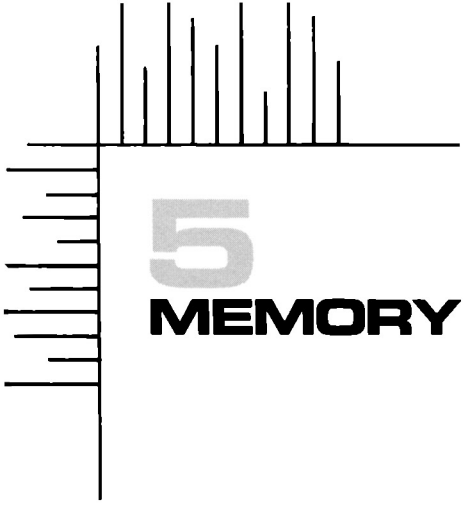
2. If you are successful with the previous exercise, try a more ambitious animated game using PRINT graphics. For instance, you might try a Space Invaders–style game where you control a laser at the bottom of the display that is under attack by a randomly moving spaceship at the top of the display, which is bombarding you with bombs. To fire lasers form your base, use the BASIC function STRIG(X), which reads the trigger button on joystick X. If STRIG(X) is equal to 1, then the button on

joystick X is being pressed; if STRIG(X) equals 0, it is not being pressed.

3. Rewrite the Ping-Pong game using the lower resolution graphics mode 3. Note that the screen coordinates will be different in this mode. How does the use of lower resolution change the action of the game? Does it speed it up? Slow it down? Is the game more exciting? Is it more interesting visually?

4. The Ping-Pong game presented in this chapter lacks certain random factors. The pattern of the ball is always the same, because it starts in the same place and bounces in the same pattern every time it strikes a wall or a paddle. Add a random factor to the game using the RND function. Start the ball at a random position, heading in a random direction. Change the way the ball bounces depending on what portion of the surface of the paddle it strikes.

5. Rewrite the Ping-Pong game so that it can be played by two players using two joysticks. One player will have a paddle at the left end of the playfield while the other will have a paddle at the right end of the field. Keep a separate score for each player, based on how many times his or her opponent has missed the ball, as in a real Ping-Pong game. Terminate the game when one player reaches an appropriate score, such as 21.



5 MEMORY

As we saw in the Introduction, graphics on the Atari computers are controlled by two integrated circuits—chips, if you will—inside the computer. These integrated circuits are called ANTIC and GTIA.

ANTIC and GTIA take information passed to them by the operating system (or, in some instances, by the programmer) and convert that information into a video signal, which is output to the video monitor (or television set) that you are using as a display. It is these chips that create the images of the graphics characters, the text characters, and any other characters that we place on the video display. They determine the color, shape, and position of these images. They determine the mode in which these images are output. They translate information provided by our PLOT statements into illuminated pixels on the display. In short, it is ANTIC and GTIA that are ultimately responsible for everything that you see on the screen.

Up until now, we have not worked directly with these chips; we have allowed the OS to do this for us. If we are to employ more complex graphic techniques, however, it will be necessary for us to open a direct line of communication with these chips, so that we can tell them precisely what sorts of graphic images we want. And the channel that we will use for this communication is *memory*.

THE ANATOMY OF MEMORY

What is memory? It is a series of electronic circuits, inside the computer, that can be used to store numbers in an encoded form. Since all information processed by a computer must first be converted into numbers (a process that happens automatically as we enter data into the computer), memory can be used to store any kind of information, including programs, that we put into it.

When you type a computer program on the Atari—or load a computer program from a disk or tape—that program is stored in memory as a series of numbers. When, in turn, you use a program to save other types of data, such as word processing documents, spreadsheet figures, even graphics, that information is also stored in memory as a series of numbers.

Specifically, computers store information as *binary numbers.* Binary is a numbering system similar in principle to the decimal numbering system we ordinarily use for counting, but different in several particulars. For instance, binary uses only two different digits (0 and 1), rather than the ten different digits (0 through 9) used by the decimal numbering system. All binary numbers are made up of combinations of these two digits. Here are some examples of binary numbers:

```
1
110
101101
11101101
```

The first of these, 1, is equivalent to the decimal number 1. The second, 110, is equivalent to the decimal number 6. The third, 101101, is equivalent to the decimal number 45. And the fourth, 11101101, is equivalent to the decimal number 237.

In computer parlance, a single binary digit is called a *bit.* Each memory circuit in the Atari is constructed in such a way that it can hold no more than eight bits, or a *byte,* in computer lingo. This means that the smallest number that can be held in a single memory circuit is 00000000, equivalent to the decimal number 0, and the largest number that can be held in a memory circuit is 11111111, equivalent to decimal 255. Thus, we can say that each memory circuit can hold a number in the range 0 to 255.

A computer such as the Atari is designed in such a way that it can have 65,536 memory circuits active at any one time. This is equivalent to 64 *kilobytes,* or *K,* for short. (A kilobyte is equal to 1,024 bytes.) The Atari 800XL is a 64K computer, though some of the earlier Ataris contained less than this theoretical maximum. Each of these memory circuits is given an identifying num-

ber, called an *address*. The first memory circuit in the machine is given an address of 0, the second an address of 1, all the way to the final address of 65,535. (The 65,536th address is 0.)

This assigning of addresses makes it easier for the programmer to store values at specific addresses and retrieve those values later—a common activity in machine language programming. (Machine language is the native language of the computer, consisting of numeric commands directly executed by the *central processing unit*, or *CPU*, of the computer.)

THE PEEK FUNCTION

You can see the actual contents of your computer's memory, from Atari BASIC, by using the PEEK(X) function. The PEEK(X) function is automatically set equal to the value stored in memory address X. For instance, this statement

```
PRINT PEEK(49152)
```

will print the contents of memory address 49152. And this statement:

```
C = PEEK(G+5)
```

will set variable C equal to the contents of the address obtained by adding the value of variable G to 5.

By placing the PEEK function inside a loop, we can print out an entire range of memory. Here, in fact, is a program that will print out the entire 64K of the computer's memory:

```
10 FOR I = 0 TO 65535
20 PRINT PEEK(I) ;
30 NEXT I
```

Before you run this program, be warned that it takes several minutes to complete its task—and the numbers that it will print will not be especially meaningful to you, though they are very meaningful to the computer.

THE POKE STATEMENT

The BASIC command POKE can be used to alter the contents of memory: it places a specified value at a specified memory address. It is used in this form:

```
POKE address, value
```

where "address" is the memory address to be changed and "value" is the value you wish to place in that address. The value may not be greater than 255.

For example, this command

```
POKE 14400,C
```

will place the value represented by variable C in memory location 14400. If variable C is greater than 255 (or less than 0), Atari BASIC will return an error message. Note that you must be extremely cautious when poking values into memory, as you may disrupt important information contained there. In fact, a misguided POKE can cause the computer to freeze up, so that you will have to turn it off and back on again to regain control. This won't hurt the computer, but it will destroy any important information (such as laboriously constructed programs) that may have been stored there. It is a good idea to save programs and other data to disk or tape before poking around in memory. We'll be using both the PEEK and POKE commands frequently in the remainder of this book, to create graphics.

As a rule, there are two types of memory in every computer: *random access memory* (or *RAM*) and *read-only memory* (or *ROM*). RAM is sometimes called "user memory," because it is the memory that you use to store programs and the data processed by those programs. RAM can be altered, by the POKE command or by machine language instructions. ROM cannot be altered, but can only be read—that is, values stored in ROM may be retrieved but not changed. ROM is generally used for storing very important programs and data that will frequently be used in the normal functioning of the computer.

The Atari computers use ROM memory to store the operating system and the character set, as well as the BASIC interpreter. On the pre-XL Atari computers, this BASIC ROM was installed inside a cartridge, so that it could be removed from the computer when not in use. On the more recent XL models, this BASIC ROM is built into the machine, although you can remove it from the computer's "memory space" by holding down the OPTION key when you turn on the computer, which causes the BASIC ROM to be replaced by 8K of RAM memory.

Although memory is primarily used for storing data and programs, there is a third use to which it can be put: *memory mapped input/output.* This simply means that we can use the computer's memory to pass information to external devices, such as the video display or the disk drives.

This is how we will communicate with ANTIC and GTIA. We will store values in memory, and these values will be passed automatically to these chips. Depending on the specific memory

address in which we store these values, they will be interpreted in different manners, but all of these messages that we will send to the graphics chips will concern the graphic images we wish to place on the computer display.

VIDEO MEMORY

The most common method of passing information to the graphics chips is through *video memory*. In text mode, video memory is a series of memory addresses, each of which corresponds to one of the character positions on the video display. By placing a special code number for a character in a given memory address within video memory, we are effectively telling the graphics chips to display that character in that position on the screen. (As we'll see in a minute, these code numbers are not quite the same as the ATASCII code.)

Ordinarily, this is done automatically by the OS. For instance, when we tell the BASIC interpreter to PRINT "WORD", the BASIC interpreter tells the operating system (which is, you will recall, a large machine language program) to put the codes for the characters W, O, R, and D into four successive addresses in video memory, beginning with the address equivalent to the current cursor position on the display. This, in turn, tells the graphics chips to output the images of these characters to the monitor.

If we wish to do so, we can place ATASCII values directly into video memory with the POKE statement, bypassing the OS altogether. This isn't as simple as it sounds, since video memory can be placed almost anywhere in the Atari memory. If we do not like the current location of video memory, we simply tell the graphics chips to move it to a different address. Otherwise the operating system will put video memory into the best location it can find for it. In this book, we will simply let the OS do this work for us.

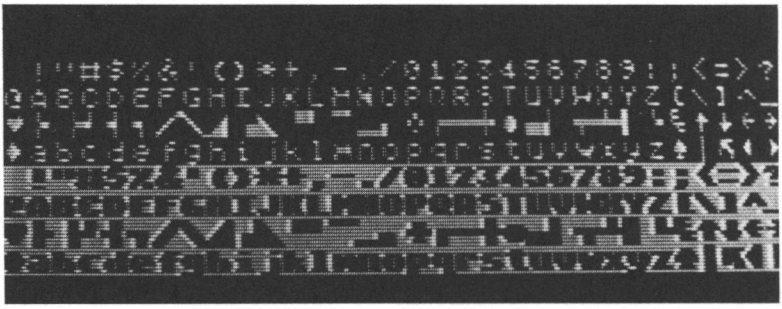
How do we know where to find video memory? This information is stored by the OS at memory addresses 88 and 89. We can determine the starting address of video memory with this statement:

```
VIDMEM = PEEK(88) + 256 * PEEK(89)
```

This sets VIDMEM equal to the address of the video memory location that controls the upper left-hand corner of the display.

As an experiment, clear the display by typing CONTROL-"<". Type the following in the immediate mode:

```
VIDMEM = PEEK(88) + 256 * PEEK(89) : POKE VIDMEM,1
```



The Atari character set

and press RETURN. In the upper left-hand corner, an exclamation point ("!") will appear.

Why an exclamation point? The ATASCII code for "!" is 33, yet the number we POKEd into the first position of video memory was 1. What a curious result!

As mentioned briefly before, the codes used for placing characters in video memory are not the same as the ATASCII codes. This is somewhat confusing. The codes used in video memory are called the Atari *screen codes*. Fortunately there is a fairly simple correspondence between the ATASCII codes and the screen codes. ATASCII codes can be converted to screen codes with the following subroutine, where ACODE equals the ATASCII code to be converted and SCODE is its screen code equivalent:

```
1000 IF ACODE<32 THEN SCODE = ACODE + 64 : RETURN
1010 IF ACODE<96 THEN SCODE = ACODE - 32 : RETURN
1020 SCODE = ACODE : RETURN
```

Here is a short program that will poke all 256 screen codes into video memory, in order, in eight rows of thirty-two characters apiece. Hit RESET before running this program:

```
10 GRAPHICS 0 : POKE 752,1 : ? CHR$(125);
20 VIDMEM = PEEK(88) + 256 * PEEK(89)
30 ADDRESS = 44 : COUNT = 0
40 FOR I = 0 TO 255
50 POKE VIDMEM+ADDRESS, I
60 ADDRESS = ADDRESS + 1
70 COUNT = COUNT + 1
80 IF COUNT = 32 THEN ADDRESS = ADDRESS + 8 : COUNT = 0
90 NEXT I
100 GOTO 100
```

By keeping count of the number of characters poked into each row, and jumping ahead eight memory addresses after each row of thirty-two characters is poked, this program neatly formats the characters on the display. The first row of characters may seem to be slightly indented. This is because the first character in the Atari character set is a blank space and is therefore effectively invisible.

MODES AND MEMORY

The size and organization of video memory varies according to the graphics mode that we are currently using. When we change modes via the GRAPHICS statement, the OS automatically takes care of all housekeeping details concerning video memory, such as deciding where in memory it must reside and how much space it will take up. A mode 0 display requires 960 bytes of video memory, one for every character in twenty-four rows of forty characters apiece. It shouldn't be hard to deduce, then, that a full-screen mode 1 display would require 480 bytes of memory, one for every character in twenty-four rows of twenty characters apiece. And a mode 2 display, naturally enough, requires 240 bytes of memory, one for every character in twelve rows of twenty characters apiece.

In mode 0, the first forty bytes of video memory represent the first row of the display, the second forty bytes of video memory represent the second row of the display, and so forth. Here is a program that fills the first row with A's.

```
10 GRAPHICS 0 : POKE 752,I : ? CHR$(125);
20 VIDMEM = PEEK(88) + 256 * PEEK(89)
30 FOR I = 0 TO 39
40 POKE VIDMEM+I, 33
50 NEXT I
60 GOTO 60
```

In modes 1 and 2, however, the first twenty bytes of video memory represent the first row, the second twenty bytes the second row, etc. Change line 10 of the last program to read

```
10 GRAPHICS 1 + 16
```

and RUN it. This time the first *two* rows should fill up with A's. But now the A's are twice as wide.

Similarly, if we change line 10 to

```
10 GRAPHICS 2 + 16
```

and RUN it, the first two rows again fill up with A's, but the characters are now twice as tall as well.

BITMAP MEMORY

How does video memory work in a bitmap mode? In quite a different manner than in text mode. While in text mode, each byte of video memory describes a character via its screen code. While in bitmap mode, the individual *bits* in video memory describe the colors of the individual pixels. Hence the term *bit map.*

The manner in which these bits describe colors varies according to the mode that we are in. In mode 8, each bit determines which color register will be used for the corresponding pixel. A 0 bit indicates the background color, a 1 bit the foreground color.

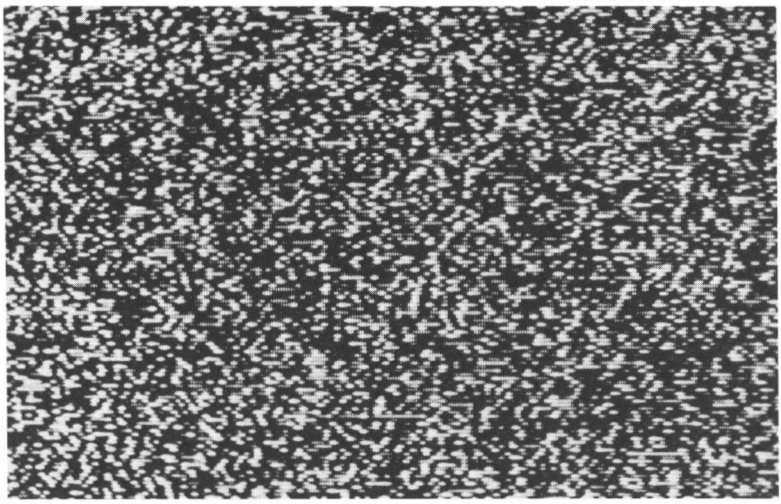
Just as in text mode the first byte of video memory describes the first character in the first row of characters on the display, in mode 8 the first byte of video memory describes the first eight pixels in the first row of pixels on the display. Suppose, for instance, that the first byte of video memory contains the number 186. In binary, the number 186 becomes 10111010. In mode 8, this would mean that the pixel in the upper left-hand corner of the display would have the foreground color, but would be followed by a pixel in the background color, which would in turn be followed by three pixels in the foreground color, one pixel in the background color, one pixel in the foreground color, and one pixel in the background color. The next eight pixels in the first row are determined by the second byte of video memory.

To illustrate this, RUN this program:

```
10 GRAPHICS 8
20 VIDMEM = PEEK(88) + 256 * PEEK(89)
30 POKE VIDMEM,186
40 GOTO 40
```

A series of tiny, nearly indistinguishable pixels will appear in the upper left-hand corner of the display. They are so tiny, in fact, that it is difficult to determine whether they follow the pattern that we just described. Here is a program that fills the mode 8 display with random pixel patterns:

```
10 GRAPHICS 8 + 16
20 VIDMEM = PEEK(88) + 256 * PEEK(89)
30 FOR I = 0 TO 7679
40 POKE VIDMEM+I, INT(RND(1)*256)
50 NEXT I
60 GOTO 60
```



RUNning the author's random-pixel-pattern program will give you a display like this.

This program is slow, but it makes its point. You will be forgiven if you hit BREAK (or RESET) before it finishes filling the display. Here is a program that draws a more distinctive pattern on the mode 8 display:

```
10 GRAPHICS 8 + 16
20 VIDMEM = PEEK(88) + 256 * PEEK(89)
30 FOR I = 0 TO 7679
40 POKE VIDMEM+I, 240
50 NEXT I
60 GOTO 60
```

Bear in mind, as you watch this program execute, that the binary representation of 240 is 11110000. (Notice, also, that this program is considerably faster than the last, because it no longer has to generate a random number on each pass through the loop.)

Video memory in mode 8 is 7,680 bytes long. This may seem a little odd at first, since there are 61,440 pixels on the mode 8 display. However, 1 byte of video memory can describe 8 mode 0 pixels, so the number of bytes required is actually 61,440 divided by 8, or 7,680.

The two-color modes use video memory in the same fashion as mode 8, except that fewer bytes are used to describe a line. Mode 6, for instance, with 160 pixels (20 bytes) on ninety-six lines, requires 1,920 bytes of video memory, with each 0 once

again representing background color and each 1 representing foreground color.

The four-color modes work somewhat differently. In a four-color mode each *pair* of bits represents a color. Because there are four possible combinations of two bits—00, 01, 10, and 11—a bit pair may represent four possible color registers. Thus, each byte in a four-color mode represents only four pixels. Similarly, the sixteen-color modes offered by GTIA require four bits to describe each color register, using the sixteen possible combinations of four bits. Each byte, therefore, can represent only two pixels.

Now that you have read this information, you will be glad to hear that you will not be called upon to create graphics displays by changing the values of individual bits in video memory—at least not while programming in BASIC. (If you ever find yourself working in machine language, this information may come in handy.) It is much easier, and generally faster, to let the OS decide what bits in video memory need to be changed in order to produce colored pixels at the proper coordinates on the graphics display. So we will continue setting pixels using the PLOT statement, and placing characters on the display via the PRINT statement. However, in the next chapter, when we study the display list, it will nonetheless prove useful to understand how video memory is arranged, because we will be playing a few tricks on the OS that may make it a little . . . well . . . confused.

IMPORTANT ADDRESSES

In addition to video memory, there are several other memory locations within the Atari that we can manipulate for important purposes. Some of these are the OS variables. These are simply locations, usually at very low memory addresses, where the OS stores important numbers. By reading these numbers using the PEEK function, we can learn valuable information. By changing them with the POKE command, we can achieve significant results.

One such location is at addresses 88 and 89, where the address of video memory is stored by the OS, as we saw above. Here are some other important addresses:

106 (RAMTOP)—This contains the address, divided by 256, of the highest RAM address available in your machine. Because some of the earlier Atari models contained less than the full 64K of RAM, the value at this address can be used to determine where memory ends, like this:

```
RAMTOP = 106 * 256
```

By changing the value at this address, we can fool the OS into thinking that there is less memory in the computer than there actually is. We generally do this when we need a certain amount of memory for our own purposes, as we shall see in later chapters. To reserve 1K of memory above RAMTOP, we could write

```
POKE 106, PEEK(106) - 4
```

This tells the operating system that there is 1K less memory in the machine than there actually is. (Because the address is divided by 256, subtracting 1 from it actually subtracts 256 from the address.)

82 (LMARGN)—This tells the OS where the left margin of the mode 0 text display begins. The value at this location is usually set at 2, which is why the leftmost two spaces on the mode 0 display are usually not printed on by the OS. To move the margin all the way to the left side of the display, type

```
POKE 82,0
```

Immediately after this instruction is executed, the cursor will move all the way back to the left-hand margin and print all further screen output using all forty columns of the display. Try it and see. To enlarge the margin to ten spaces, type

```
POKE 82,10
```

Hit RESET to cancel the effect.

83 (RMARGN)—This tells the OS where the right margin of the mode 0 display is. The value at this location is normally 39. To create a ten-column display, type

```
POKE 82,20 : POKE 83,30
```

and see what happens. LIST a program (or type one) to see what a ten-column display looks like. To restore the ordinary values, hit RESET, or type

```
POKE 82,2 : POKE 83,39
```

87 (DINDEX)—Contains the current display mode number. The default value, after hitting RESET, is 0, indicating that the display is in mode 0. Though this location can be used to read the current mode (if, for instance, your program does not know what

mode it is running in and must find out), it should not be used to change the mode.

The preceding are only a sampling of the OS variables that we can PEEK and POKE in the Atari's memory. Entire books have been written on this subject; these are only the locations that are relevant to our purposes in this book.

In addition to the OS variables, we can also use the Atari's memory to control graphics through the display list, and the character set. These are just some of the exciting features of Atari graphics to be discussed in the next two chapters.

Suggested Projects

1. Write statements in Atari BASIC that will accomplish the following:

   a. Set the mode 0 margins to 9 (left) and 35 (right).
   b. Lower the top of the available memory by 2K (2,048 bytes).
   c. Determine the current display mode and save the mode number in variable MODE.

2. Write a program that POKEs the entire Atari character set onto the mode 1 display. How does the character set produced by this program differ from a mode 1 character set produced by the CHR$ function?

3. Write a program that uses the POKE command to create patterns on the mode 3 bitmap display. Experiment to see what different patterns can be created by different numbers.

6 THE DISPLAY LIST

In the introduction, we referred to the Antic and GTIA chips as small computers in their own right. This was no exaggeration. In fact, Antic even has its own program, which resides in the Atari's memory like any other program. Using the PEEK and POKE commands, we can examine and alter this program, gaining considerable power over the Antic chip.

The program for Antic is called the *display list*; it tells Antic in which graphics mode it should output each line of the display. By manipulating this program we can create custom graphics modes on the Atari video display. No longer need we be bound by the OS graphics modes. We can create new combinations of the modes that we have studied in earlier chapters and even create new modes not provided by the OS.

Before we can understand the display list, however, we must first understand the way your video monitor generates a picture.

MODE LINES

Turn on your Atari, if you haven't done so already, and look carefully at the display. If you study it very closely, you will notice that the picture is made up of thin horizontal lines. In fact, you may already have noticed these lines while watching television programs. On a standard television there are 256 of these

lines. (Technically, there are 512 lines, though if you count the lines you see on the display you will at most count 256. This discrepancy is caused by a phenomenon called *interlace*, which makes 512 lines look like 256. However, we will not be discussing interlace; instead, we will simply treat the video display image as though it were made of 256 lines, which for all practical purposes it is.)

The lines that make up the display image are called *raster scan lines*, though we will refer to them here simply as *scan lines*. They are produced by a beam of electrons shooting out of an electron gun inside the display. The inside of the display screen is coated with color phosphors; as the electron beam strikes the phosphors, it causes them to glow. A powerful magnetic field pulls the beam from side to side and up and down on the display, so that it strikes all of the phosphors inside the screen, painting a complete image. Because it moves primarily from side to side, moving further down the display for each horizontal stroke, the image seems to be made up of horizontal lines.

It may seem difficult to believe that the entire image on the display is created by a single beam of electrons, but this is in fact the case. Bear in mind that the beam moves very quickly; before the phosphors have time to lose their glow, the beam has begun painting a new image. The beam creates a fresh picture sixty times a second.

The electron beam starts its journey at the upper left-hand corner of the display. It then travels from the left side of the display to the right, drawing a complete scan line as it travels. Once at the right-hand side of the display, the beam is turned off, so that it can be repositioned at the left-hand side to draw the next line. The period during which this repositioning takes place is called the *horizontal blank*, because there is no picture being painted on the display during this interval.

Once repositioned, the beam draws another scan line from left to right, directly underneath the previous scan line. This process continues until the entire display has been covered with scan lines and the beam is at the lower right-hand corner of the display. Once again, the beam is turned off, this time to be repositioned back at the upper left-hand corner, so that it can draw the next screen. The interval during which this repositioning takes place is called the *vertical blank*; it is considerably longer than the horizontal blank, though neither takes more than the tiniest fraction of a second.

The scan lines created by the electron beam are actually wider than the video display itself; they run off the edges on both sides. This phenomenon, called *overscan*, is more pronounced on older televisions, where there has been a certain loss of picture tube quality, but it is present to some degree on all video displays.

This is why the portion of the Atari display allotted to text and graphics does not fill the entire screen; if it did, information would be lost where the scan lines disappear.

Assuming that your Atari is in mode 0 (hit RESET if it is not), type a few characters on the video display. Study the characters in relation to what you just learned about how the display image is created. Note that each mode 0 character is eight scan lines tall. (Some of these lines may contain only background color, but they are nonetheless part of the display space allotted to that character. The cursor is a good example of a character in which all eight lines are shown in foreground color.) Roughly speaking, then, we can say that a line of mode 0 characters is eight scan lines in height.

Now RUN the following program:

```
10 GRAPHICS 2
20 PRINT #6; "THESE ARE MODE 2       CHARACTERS."
```

The mode 2 characters that you are now looking at are twice as tall as the mode 0 characters; a line of mode 2 characters is therefore sixteen scan lines in height. (Mode 1 characters, because they are the same height as mode 0 characters, are only eight scan lines in height.)

Similarly, the pixels in mode 8 are each a single scan line in height. The pixels in modes 6 and 7 are each two scan lines in height. The pixels in modes 4 and 5 are each four scan lines in height. And the pixels in mode 3 are eight scan lines in height (same as the characters in mode 0).

What is the point of all this comparison of scan lines? That will become apparent in time. For now, let's simply start thinking of the Atari display as being divided into two different types of lines: scan lines, which are created by the electron gun, and *mode lines*, which are made up of a series of characters (or pixels) that are each a certain number of scan lines in height, depending on which mode we are in. In mode 0, the mode line is eight scan lines in height; in mode 2 the mode line is sixteen scan lines in height; in mode 7 the mode line is two scan lines in height; and so forth.

And that brings us back to the display list.

Display lists are created in the Atari's memory by the operating system. Every time we execute a GRAPHICS statement to create a new mode, the OS generates a display list to describe that mode to Antic. To find the display list in the Atari's memory, we must PEEK at locations 560 and 561. We can obtain the address of the display list by typing:

```
DISPLIST = PEEK(560) + 256 * PEEK(561)
```

The display list is a list of numbers. The Antic chip, however, sees these numbers as instructions.

To see what a display list looks like, type and RUN this program:

```
10 GRAPHICS 0
20 DISPLIST = PEEK(560) + 256 * PEEK(561)
30 FOR I = 1 TO 31
40 PRINT PEEK(DISPLIST + I) ; " ";
50 NEXT I
```

You should see a series of numbers that looks like this:

```
112 112 112 66 64 156 2 2 2 2 2 2 2 2 2 2
2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 65 32 156
```

This is the display list. What it does may not be immediately apparent. (If you have less than 48K memory in your Atari, the display list that you see may look a little different than this.)

What it does, primarily, is to tell Antic what sort of mode lines it is expected to put on the display. Rather than specifying a graphics mode for the entire display, as we do with the GRAPHICS command, the display list specifies the mode for every individual mode line.

You will notice that there are a lot of 2's in the display list. From this evidence, you might surmise that the number 2 is very important to Antic. In fact, the number 2 is the display list's way of representing mode 0. Unfortunately, the display list uses different numbers for the Atari graphics modes than the operating system does, which can lead to some confusion. (From now on, we will refer to the display list method of numbering the modes as the *Antic modes* and the operating system method of numbering the modes as the *OS modes.*) Essentially, this display list tells Antic that every mode line on the display is to be in OS mode 0. And, if you count the number of 2's in the display list, you will see that there are twenty-three of them, which is just one less than the number of mode lines in OS mode 0. What happened to the twenty-fourth line? We'll see in a moment.

THE LANGUAGE OF ANTIC

Let's look at the display list, instruction by instruction. The first part of the display list consists of the number 112 repeated three times. Each 112 instruction tells Antic to output eight blank scan lines to the video display, for a total of twenty-four blank scan lines. These are the blank lines at the very top of the display,

above the area where text and graphics are displayed. Some of these lines may be off the top of your display.

The next entry in the display list is 66. Although this may look like a single instruction, Antic sees it as two instructions, added together. One of these instructions is a 2, which (as we saw earlier) tells Antic to output a mode line to the display in OS mode 0 (Antic mode 2). And, sure enough, the first mode line of the display is in OS mode 0. The other instruction is a 64. (Note that 64 plus 2 equals 66.) This instruction tells Antic that the two bytes of memory that follow contain the address of video memory, so that Antic will know where to go to fetch the video image. These two bytes contain the numbers 64 and 156. (On machines with less than 48K memory, these numbers will be different.)

Although our program prints these bytes as two separate decimal numbers, they are actually one long binary number, spread across two memory addresses. To see these binary numbers as a single decimal number, we must perform a simple arithmetic operation, which we have already used in several of our PEEK routines: multiply the second number by 256 and add the two numbers together. This gives us an address of 40000. This is the same number that appears at addresses 88 and 89, where we ordinarily find the address of video memory.

To verify that this actually is the address of video memory, type

```
POKE 40000,1
```

and press RETURN. Sure enough, an exclamation point will appear in the upper left-hand corner of the display. (Once again, if you have less than 48K memory in your computer, you will arrive at different figures, but the principle will be the same.)

The next twenty-three numbers in the display list are also 2's, telling Antic to output twenty-three more mode lines in OS mode 0, for a grand total of twenty-four lines in this mode. You can verify that this is the case by looking at the display, which should contain twenty-four lines in what you have by now come to recognize as OS mode 0.

After the mode lines have been dispensed with, the next instruction is a 65. This is a kind of GOTO instruction: it tells Antic to wait for the next vertical blank (the interval during which the electron beam is repositioned at the top of the display) and go to the top of the display list, where it will execute the instructions in the list all over again. The two bytes following this instruction contain the address of the top of the display list. Using the method described a moment ago, we can calculate this address as 32 + 256 * 156, or 39968. This should be the same number contained at addresses 560 and 561, where we ordinarily

PEEK the address of the display list. By placing a different address here, we could actually have two display lists in memory at once, and alternate rapidly between them. This is rarely done, however.

A SPLIT-SCREEN LIST

Let's look at a second display list. Type and RUN this program:

```
10 GRAPHICS 1
20 DISPLIST = PEEK(560) + 256 * PEEK(561)
30 FOR I = 0 TO 33
40 PRINT PEEK(DISPLIST + I); " ";
50 NEXT I
60 GOTO 60
```

The display list revealed by this program should look like this on a 48K or larger machine:

```
112 112 112 70 128 157 6 6 6 6 6 6 6 6 6 6
6 6 6 6 6 6 6 6 6 66 96 159 2 2 2 65 94 157
```

Once again, we start out with twenty-four blank scan lines. This is followed by a 70, which is actually 64 plus 6. The 64 tells Antic that video memory is at 128 + 256 * 157, or 40320. The 6 tells it to output a mode line in Antic mode 6, which is OS mode 1. The following nineteen lines are also in Antic mode 6. This corresponds to the twenty lines of your display that should now be in OS mode 1. At this point in the display list, however, something odd happens. We see a 66 instruction, which is a 64 plus a 2. This tells Antic to move video memory to 96 + 256 * 159, or 40800. This new video memory is used for the remaining four lines of the display. The 2 tells Antic to output a line in Antic mode 2, which (you'll recall) is OS mode 0. The next three instructions output three more lines in OS mode 0. These four lines, altogether, make up the mode 0 text window at the bottom of the display.

This is how the split-screen effect of the text window is created. The OS creates a display list that tells Antic to output twenty mode lines in Antic mode 6 (OS mode 1) and four mode lines in Antic mode 2 (OS mode 0). As far as Antic is concerned, there is no such thing as a full-screen graphics mode. Every display is created a mode line at a time, and each mode line is in the mode that the display list specifies. If we wish, we can create our own custom display lists, where the mode changes on every line, generating a display that is part high-resolution text mode (OS

mode 0), part large text mode (OS mode 2), and part bitmap mode. We can mix text and graphics together freely on the same screen, even placing the text *above* the graphics (instead of using the OS-generated text window, which is always below the graphics).

BUILDING A DISPLAY

However, certain rules must be followed in creating a custom display list. If we do not assemble and use our display list in precisely the right manner, we can crash the entire graphics system, sending Antic (or the OS) off to never-never land, from which it may not return until we hit the RESET key. At the very least, we will produce some very odd displays.

To begin with, here is a list of the instructions that can be used in a display list:

0—Output 1 blank scan line
16—Output 2 blank scan lines
32—Output 3 blank scan lines
48—Output 4 blank scan lines
64—Output 5 blank scan lines
80—Output 6 blank scan lines
96—Output 7 blank scan lines
112—Output 8 blank scan lines
2—Output mode line in Antic mode 2 (GRAPHICS mode 0)
3—Output mode line in Antic mode 3
4—Output mode line in Antic mode 4
5—Output mode line in Antic mode 5
6—Output mode line in Antic mode 6 (1)
7—Output mode line in Antic mode 7 (2)
8—Output mode line in Antic mode 8 (3)
9—Output mode line in Antic mode 9 (4)
10—Output mode line in Antic mode 10 (5)
11—Output mode line in Antic mode 11 (6)
12—Output mode line in Antic mode 12 (14)
13—Output mode line in Antic mode 13 (7)
14—Output mode line in Antic mode 14 (15)
15—Output mode line in Antic mode 15 (8)
0—Jump to instruction at address in next 2 bytes
65—Wait for vertical blank and jump to instruction at address in next 2 bytes
64—Address of video memory is in next 2 bytes (must precede mode line output)

The Antic unconditional jump instruction is used to link two portions of a display list, located at different portions in memory,

together into a single display list. The only time that you might want to do this is if the display crosses a 1K boundary, that is, if it stretches across an address evenly divisible by 1,024. This is an Antic no-no; the chip cannot continue reading past such an address. You must direct its attention specifically to the next address in the display list with a jump instruction followed by the address in the next two bytes.

Here is a list of the Antic modes and their equivalent GRAPHICS statement modes:

ANTIC MODE	GRAPHICS MODE
Text Modes:	
2	0
3	-
4	-
5	-
6	1
7	2
Bitmap Modes:	
8	3
9	4
10	5
11	6
12	14
13	7
14	15
15	8

Notice that three of the text modes have no GRAPHICS statement equivalents; neither do two of the bitmap modes (12 and 14) if you are using a non-XL Atari. Descriptions of all of these modes, with the exception of Antic modes 3 to 5, can be found in the chapter on color. Mode 3 is a two-color text mode, similar to mode 0, except that each character (and therefore each mode line) is ten pixels tall, with forty characters per line. Mode 4 is also similar to mode 0 (with forty characters per line and eight pixels per mode line), except that you can have four colors on the display at one time. Mode 5 is a four-color, double-height text mode: each mode line is sixteen pixels tall, but (unlike mode 2) the characters are the same width as characters in mode 0; in brief, a tall, skinny text mode. You can still place forty characters on a line in this mode.

GRAPHICS modes 9, 10, and 11 have no equivalent Antic modes. We can establish these modes by creating a display list for

Antic mode 15 and POKEing an appropriate value into memory address 623. Here is a list of the necessary POKEs:

```
POKE 623,64 : REM GRAPHICS MODE 9
POKE 623,128 : REM GRAPHICS MODE 10
POKE 623,192 : REM GRAPHICS MODE 11
```

With this information, we can build a display list of our own. Suppose, for instance, that we wish to create a title screen for a program with characters displayed in three different sizes. The title would be in large mode 2 letters, the name of the programmer in medium-sized mode 1 letters, and anything else in small mode 0 letters. None of the OS modes would allow us to create such a display, because it requires the mixing of three different text modes on the same display. However, we can create such a combination of modes easily with a custom display list.

The first three instructions in the list should be 112, 112, and 112, so that twenty-four blank lines will be output to the top of the display. This should be followed by the instruction 64 and the address of video memory. Then we must describe each mode line to Antic, one at a time. To simplify matters, we'll put all mode lines in Antic mode 2 (GRAPHICS mode 0), except for one line in Antic mode 6 (GRAPHICS mode 1) and one line in Antic mode 7 (GRAPHICS mode 2).

How many mode lines should we include? The rule of thumb is that the total number of *scan* lines (not mode lines) on the display should not exceed 192. If we have more than 192 scan lines on the display, Antic will continue outputting mode lines during the vertical blank, and the picture will begin to roll very rapidly. On the other hand, we may safely put fewer than 192 scan lines on the display, but this reduces the area in which we may PRINT and PLOT information. It is best to try for exactly 192 scan lines, or as close under that number as we can get.

When we create a display list, we must carefully count the number of scan lines per mode line. Antic modes 2 and 6 both use 8 scan lines per mode line; Antic mode 7 uses 16. Thus, if we include one line apiece in modes 6 and 7, they will use 24 scan lines between them, leaving room for 168 scan lines. Dividing 168 by 8 tells us that we have room left over for 21 lines in Antic mode 2. The first of these can be indicated simply by adding 2 to the 64 of the previous instruction, producing an instruction of 66. The remainder could be apportioned like this: 2, 2, 2, 2, 2, 2, 2, 2, 7, 2, 2, 2, 2, 2, 6, 2, 2, 2, 2, 2, 2, 2, 2.

Finally, we must include a 65 instruction, followed by the address of the start of the display list. This, as you will recall, tells Antic to wait for the vertical blank and start executing the display list all over again.

For inclusion with a program, we must place this display list in a DATA statement, like this:

```
DATA 112, 112, 64, -1, 2, 2, 2, 2, 2, 2, 2, 2, 2, 7, 2, 2, 2, 2, 2, 2, 6, 2, 2, 2,
2, 2, 2, 2, 2, 65
```

The addresses of video memory and the display list are not included in the DATA statement, because these must be calculated when the program is actually run.

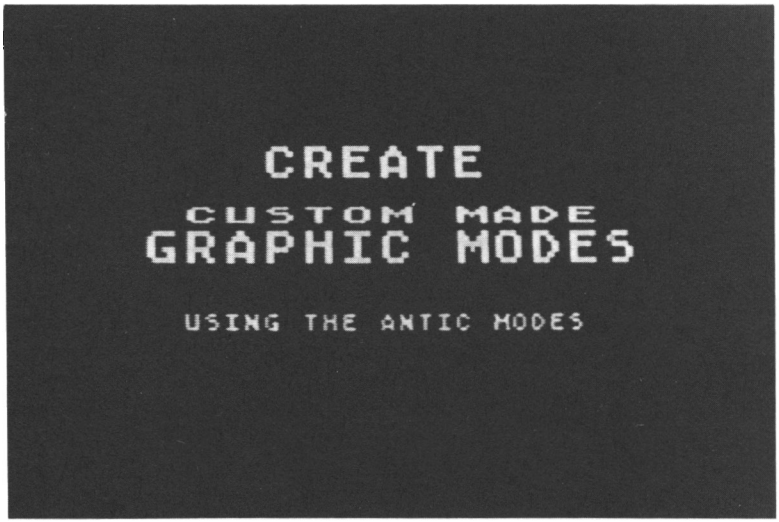
INSTALLING THE LIST

How do we go about putting the display list in place? First, we must use the GRAPHICS statement to force the OS to generate a display list of its own. Because the OS will perform most of the related housekeeping functions for us, we need not worry about such petty details as finding a safe place in memory to put the list, or telling Antic where the list is located. What GRAPHICS mode should we tell the OS that we want? Well, as a rule we should always ask Antic to create a display list that will be as long as or longer than the one we are creating. Ideally, it should be exactly the same length as the one we are creating, but this is not always possible.

If we plan to display text information, it is also helpful to have Antic generate a text mode display list. The text mode with the largest display lists are modes 0 and 1; hence, we will use mode 0.

We then need a sequence of instructions that will take the display list in our DATA statements and place it in memory. Here is one possible sequence:

```
10 GRAPHICS 0
20 DISPLIST = PEEK(560) + 256 * PEEK(561) : REM GET ADDRESS
OF DISPLAY LIST
30 ADDRESS = DISPLIST : REM POINTER TO START OF LIST
40 FOR I = 0 TO 27 : REM 28 ITEMS IN DATA STATEMENT
50 READ NUMBER : REM READ NEXT ITEM
60 IF NUMBER = -1 THEN POKE ADDRESS, PEEK(88) : POKE
ADDRESS+1,PEEK(89) : ADDRESS = ADDRESS+2 : GOTO 90
70 POKE ADDRESS, NUMBER : REM ADD ENTRY TO DISPLAY LIST
80 ADDRESS = ADDRESS + 1 : REM POINT TO NEXT ADDRESS
90 NEXT I
100 POKE ADDRESS, PEEK(560) : POKE ADDRESS+1, PEEK(561) :
REM ADD ADDRESS OF DISPLAY LIST TO LAST INSTRUCTION
1000 DATA 112, 112, 112, 66, -1, 2, 2, 2, 2, 2, 2, 2, 2, 7, 2, 2, 2, 2, 2,
6, 2, 2, 2, 2, 2, 2, 2, 2, 65
```



The program for this display incorporates techniques described by the author.

This routine will POKE the new display list into place over the old display list. Type the program and RUN it.

Immediately, the display will start to look odd. Two bands of black will appear near the center. These are the Antic mode 6 and 7 mode lines. Because they draw their background colors from a different register than Antic mode 2, the background is a different color in these mode lines than in the surrounding lines.

Now type LIST and press RETURN. You might receive something of a shock as the text of the program is printed on the display. The text that prints in the Antic mode 6 and 7 lines is a different size and color from the text in the other lines. Type LIST again at the bottom of the display to watch text scroll upward through the new mode lines. The effect is odd, to say the least.

An important thing to notice, however, is that text displayed after the Antic mode 7 line (the first black stripe) is out of place, as though the left-hand margin had somehow been moved to the middle of the display. After the mode 6 line (the second black stripe), the text seems to straighten out again.

Remember that the OS, which is printing these characters on the display, thinks that the entire screen is in mode 0, because we have not told it otherwise, In fact, there is no way that we *can* tell it otherwise. Thus, it positions the text in video memory on the assumption that each line contains forty characters of text. How-

ever, two of the lines on the display contain only twenty characters. The first of these throws the operating system out of phase. It now believes that the middle of the display is the left-hand margin. The second twenty-character line compensates for this and puts the OS back in sync again.

POSITIONING TEXT

As you can imagine, this makes printing characters on a modified display a tricky task. We must try to outwit the OS, positioning our text according to where the OS thinks the text should be, rather than where it actually is. To this end, the POSITION command will be useful.

First of all, however, we must move the left-hand margin of the display all the way to the left, with this instruction:

```
5 POKE 82,0
```

(Hit RESET before adding this line.) Address 82, you may recall, contains the position of the left-hand text margin, as used by the OS.

Now, suppose that we wish to display the following title screen:

DISPLAY LIST DEMO

by

CHRISTOPHER LAMPTON

with the first line in the Antic mode 7 portion of the screen, the second line in the Antic mode 2 portion of the screen, and the third line in the Antic mode 6 portion of the screen. Positioning the first line isn't difficult, as long as we remember that there are only twenty characters on the mode 7 line. The line has a total of seventeen characters. Thus, we will print it starting in position 1 of line 9. This can be accomplished with the following POSITION statement:

```
110 POSITION 1,9
```

Add this line to your program. Then, add this line:

```
120 PRINT "DISPLAY LIST DEMO"
```

Positioning the word "by" in the mode 2 portion of the screen is trickier. Ordinarily, to center a two-letter word on a forty-column line, we would start the word in horizontal position

19, and line 12 would be a good, central location between the mode 7 and mode 9 lines. It would make sense, if the entire display were in Antic mode 2, to use the statement POSITION 19,12 to place this text where it belongs. However, bear in mind that the OS has been thrown out of sync by the mode 7 line and we must now position characters twenty spaces earlier than we normally would. Thus, we must place this word in position 39 of the previous line, with the statements

```
130 POSITION 39,11
140 PRINT "by"
```

Add these lines to the program.

The mode 6 line, which is line 15 on the display, is also twenty characters long. Positions on this line are still twenty characters out of sync, as far as the OS is concerned. The name CHRISTOPHER LAMPTON is nineteen characters long, so we might as well start it at the margin. However, the OS sees the margin as twenty spaces before its actual position. Thus, we can position the name at the margin (position 0) on line 15, with the statements

```
150 POSITION 20,14
160 PRINT "CHRISTOPHER LAMPTON"
```

Add these lines to the program, as well as the following line:

```
170 GOTO 170
```

and RUN the program. The three lines of text should be neatly formatted in their respective mode lines.

Two more details would neaten up this display nicely. One is to dispose of the cursor with this line:

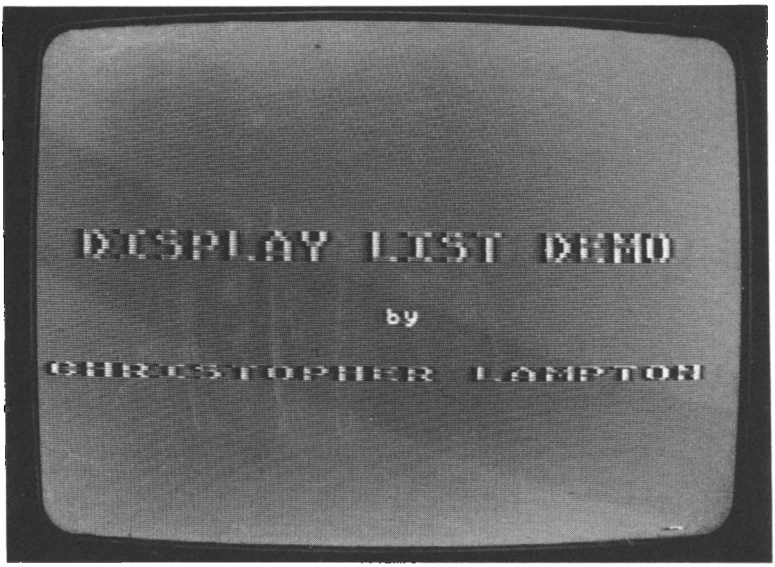
```
15 POKE 752,1
```

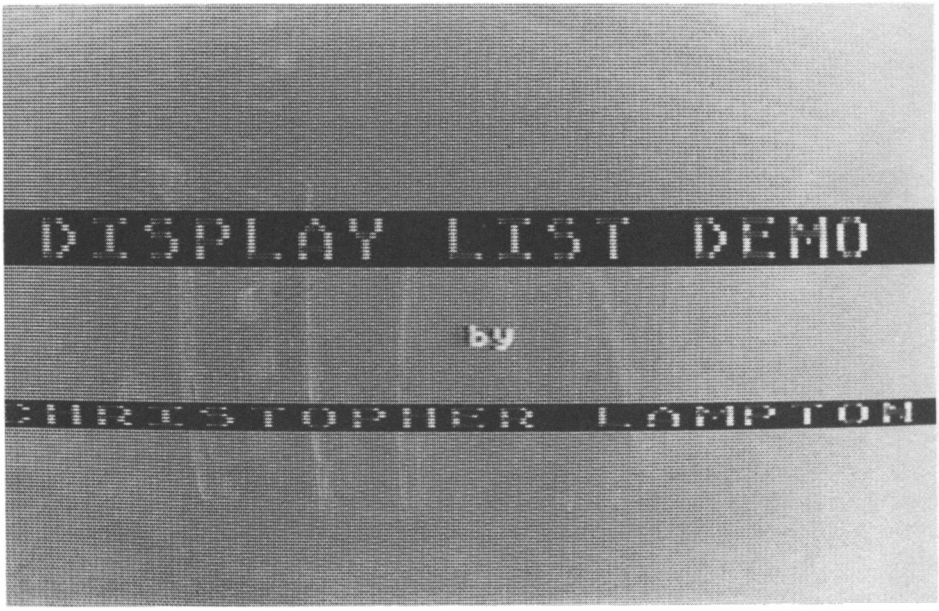
Another is to change the background color of the modes 6 and 7 lines to the same background color as the mode 2 lines. This can be done with the addition of this line:

```
17 SETCOLOR 4,9,4
```

Now that you have the entire program typed, RUN it.

This should give you some idea of what can be done with custom display lists. Although it is certainly easier to have the OS create a display list for you than to build one yourself, the custom display list greatly increases the versatility of the machine. And





These pictures were created using the display list demo program in the text. On the screen you would see yellow letters, a blue background, and two black bands behind the letters until you added line 17: SETCOLOR 4,9,4.

you need not restrict yourself to mixing text modes on the display. You can also mix bitmap modes with text modes, if that should be useful.

However, some of the best graphic effects possible on an Atari can be achieved using text graphics alone, that is, graphics constructed from sequences of graphics characters. In the next chapter we will see how character graphics can be taken a quantum leap beyond the simple graphics we developed in Chapter Two using the built-in Atari character set. We'll take a look at the mysteries and possibilities of redefinable character sets.

Suggested Projects

1. Is it possible to write a display list that will put all sixteen of the Antic display modes on the screen at one time? If so, write a display list for such a mixed mode display. Write a program to put this display list in memory and activate it.

2. Write a program that will reverse the normal mode 8 display and put the text window on the top of the display and the high-resolution display underneath it.

3. Write a program that displays text in alternating lines of OS mode 0 text and OS mode 1 text.

7 CHARACTER SETS

There are 128 printable characters in the Atari character set. Because some of these double as control characters, it is occasionally necessary to press ESC (or to PRINT CHR$(27)) before printing a character on the display, to inform the OS that we wish to see the printable character rather than the control action. (See Chapter One for a discussion of how this works.)

Maybe you've wondered, as you watched the Atari print the alphabet and the various numerals, punctuation marks, and graphics characters on the display, just how the OS knows what these characters *look like.* How does it know that the letter A comprises two opposing diagonal lines connected by a horizontal one, or that the letter Y looks like a slingshot without the sling?

The answer is that the OS neither knows nor cares what any of these characters look like. All it cares about is what their ATASCII code numbers are. Once it has determined that it is supposed to print ATASCII character 65 (the letter A) on the video display, it converts that character code into the equivalent screen code (see Chapter Five) and passes the buck to the graphics chips, telling them to print the character represented by that screen code at the requested position on the display.

CHARACTER MAPS

How do the graphics chips know what the character looks like? The memory of the Atari contains *pictures* of those characters.

Pictures? Stored in the computer's memory? How can that be?

Well, we've already seen one method by which pictures can be stored, pixel by pixel, in a series of memory locations: the bitmap. And this is exactly how the pictures of the characters are stored. A special portion of the Atari's memory called the *character ROM* contains bitmapped pictures of every character that the Atari can print on its display. The character ROM is located between addresses 57344 and 58367. Every eight bytes in this range, starting with the eight bytes from 57344 to 57351, contain a bitmap for a single character.

You'll recall that the simplest form of bitmap is one in which the binary digit 1 represents a pixel in the foreground color and the binary 0 digit represents a pixel in the background color. And this is precisely how the character bitmaps work. Every byte (eight bits) in the eight-byte bitmap represents a single line of pixels in the character. Here, for instance, is the bitmap for the letter A:

DECIMAL	BINARY
0	00000000
24	00011000
60	00111100
102	01100110
102	01100110
126	01111110
102	01100110
0	00000000

Look closely at the binary representation of the bitmap. Within these 0's and 1's you should be able to discern the structure of the letter A. Remember that the 1's represent pixels in the foreground color (i.e., the pixels in the body of the letter) and the 0's represent pixels in the background color. The same bitmaps, incidentally, are used in all of the Antic text modes; only the size and shape of the pixels is changed from one mode to the next.

The character bitmaps are stored in ROM, so they cannot be changed. However, it is possible to create our own set of character bitmaps in RAM and tell the graphics chips to use these new bitmaps instead of the ones in ROM. In this way, we can redefine the character set, that is, design an entirely new set of characters that look the way we want them to look.

REDEFINING CHARACTERS

Why would we do this? Well, the simplest reason is that we might not like the way the current character set looks. Suppose, for

instance, that we have written a program that is more suited to an old English-style character set than to the characters normally displayed by the Atari. We could design a set of appropriate characters to replace the standard ones.

However, the most common reason for redefining the character set is to create custom graphic characters. We can use these custom characters to construct images that combine the appearance of high resolution with the simplicity of character graphics. Like high-resolution graphics, custom character graphics give us control over individual pixels—the pixels within the bitmaps of the characters. But once we have defined the individual pixels within a character, we can use those characters in the same way that we use ordinary character graphics, by printing them on the display or poking them into video memory.

Here are the steps we must take to create a custom character set:

1. Design the bitmaps for the new characters.
2. Place the bitmaps in the Atari's RAM.
3. Tell the graphics chips where the new character set is located.

The first of these steps is the most difficult. Designing character bitmaps can be a pretty tedious job. In general, we will not want to redesign the entire character set. We might, for instance, wish to leave the letters of the alphabet looking like they normally do, and only redesign the graphics characters. Thus, we will want to retain the normal bitmaps for part of the character set and alter only a selected portion.

INSTALLING THE NEW SET

There are several methods we can use to place the new bitmaps in memory. If we wish to retain portions of the old character set, we should first copy the existing character set from ROM into RAM. This can be done with a simple FOR-NEXT loop, like this:

```
10 FOR I = 0 TO 1023
20 POKE CHARSET+I, PEEK(57344+I)
30 NEXT I
```

where CHARSET equals the address at which we wish to place the new character set. Once the old set has been copied into place, we can modify those sections of it that need modification. In general, we will store the bitmaps for the new characters inside our BASIC program in DATA statements, and poke the bitmaps into place with another FOR-NEXT loop.

Finally, we tell the graphics chips where the new character set is located by dividing the address of the new character set by 256 and poking the result into location 756, like this:

```
POKE 756, CHARSET/256
```

Obviously, we must choose a location for the character set that is evenly divisible by 256.

Before we can put the new character set in place, we must protect a section of the Atari's memory, so that the portion of memory where we store the character set will not be used by either the BASIC interpreter or the OS. As indicated in Chapter Five, we do this by POKEing a new value into location 106, where the OS stores the highest address that it can safely use:

```
POKE 106, PEEK(106)-4 : GRAPHICS 0
```

This takes the value already in 106 and reduces it by 4. This will reserve 1,024 bytes at the top of your computer's memory, whether your computer contains 16K or 48K or 64K or whatever. (Subtracting 1 from the value at 106, you will recall, lowers the top of memory by 256 addresses.) The GRAPHICS 0 instruction forces the OS to move video memory and the display list out of the way of our reserved memory area. We can now determine a safe address for CHARSET by multiplying the new value at 106 by 256, like this:

```
CHARSET = PEEK(106) * 256
```

This tells us where the top of memory is currently located. We can build our character set upward from there.

Here, then, is a complete routine for moving the character set from ROM to RAM:

```
10 POKE 106, PEEK(106)-4 : GRAPHICS 0
20 CHARSET = PEEK(106) * 256
30 FOR I = 0 TO 1023
40 POKE CHARSET+I, PEEK(57344+I)
50 NEXT I
60 POKE 756, CHARSET/256
```

RUN this program. There will be a momentary pause as the FOR-NEXT loop moves the character set, and then the BASIC "READY" message will appear. The first thing you should notice happening is that, well, nothing happens. Everything should look the same after this program executes as it did before, assuming that the program was correctly typed. (If it was not typed correct-

ly, some *very* strange things may happen; if so, hit RESET and try again.) Though you cannot tell it simply by looking at the display, we are now using a character set based in RAM rather than in ROM.

PLAYING WITH THE CHARACTERS

The advantage of the RAM character set, as stated, is that we can make changes in it. In the spirit of experimentation, then, let's make a change. Type this statement in the immediate mode:

```
POKE CHARSET, 255
```

and press RETURN. Whoa! What just happened? Suddenly the video display filled with stripes!

To explain this effect, we need to have some idea of how the character bitmaps are arranged. Because the graphics chips see the character set as a sequence of screen codes, the bitmaps for the characters are arranged in the order of their screen codes. If you use the simple method we showed you in the last chapter for deriving screen codes from ATASCII codes, you'll see that the first bitmap in the character set belongs to the character with ATASCII code 32, which has a screen code of 0. This happens to be the code for the blank space. Yes, there is actually a bitmap for the blank space. It is made up of the binary numbers 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, 00000000, and 00000000, indicating that all of the pixels in the blank space are set to the background color. By POKEing a value of 255 into the first position in the bitmaps, we changed the first of these binary numbers to 11111111, indicating that the first line of pixels in the blank space character should be set entirely to foreground color. Thus, all of the blank spaces on the display—which is to say all of the character positions without a character in them—were changed by this single POKE. The effect is quite dramatic.

For a slightly different effect, type

```
POKE CHARSET,204
```

this POKES the binary number 11001100 into the first position of the blank space bitmap, producing a sequence of dotted lines across the display.

For an even more dramatic effect, type the following in the immediate mode:

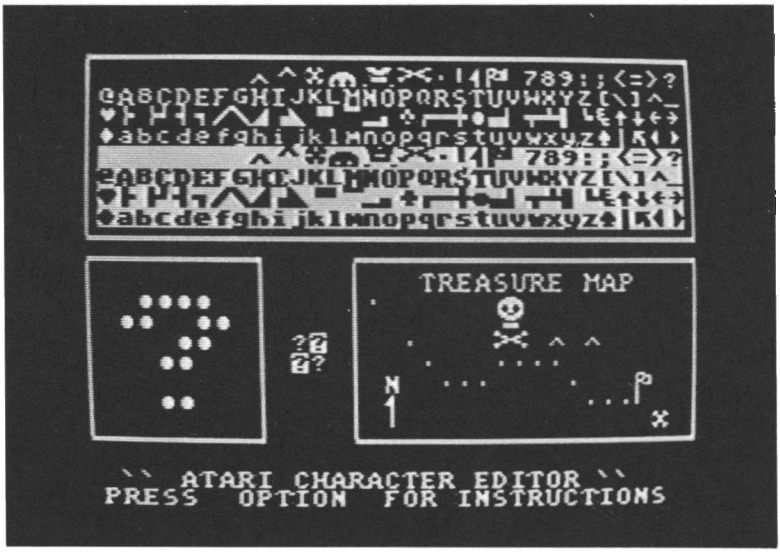
```
FOR I = 0 TO 7 : POKE CHARSET+I, 204 : NEXT I
```

and press RETURN. Now the stripes run vertically instead of horizontally. If you change the 204 to a 255 and enter this line again, the display will be almost entirely whited out. To cancel the effect and return the display to normal, change the 204 to a 0 or simply hit RESET, which will restore the normal character set. The latter method is recommended, at least while you are working in the immediate mode.

THE CHARACTER EDITOR

Rather than learning the steps necessary to design your own custom character sets, you will now be presented with a program that will automate the task. This character set editor is written in Atari BASIC and features full-screen bitmap editing, as well as an option to save your new character sets (along with the remaining portions of the old set) to disk or tape. Because this is a complex program and features some advanced programming techniques not discussed in this book, most of its internal workings will not be explained. Here is the listing:

```
1 REM ****************************************
2 REM *                                      *
3 REM *  CHARACTER SET EDITOR                *
4 REM *                                      *
5 REM ****************************************
6 REM
10 GOSUB 5000 : REM INITIALIZE EDITOR
20 GOSUB 2900
30 ? CHR$(125) ;
35 GOSUB 7000
39 REM **MAIN PROGRAM LOOP
40 A = USR(PRTCHARS) : REM PRINT CHARSET
50 POSITION 5,21 : ? "** ATARI CHARACTER EDITOR **";
60 POSITION 4,22 : ? "PRESS 'OPTION' FOR INSTRUCTIONS";
70 IF SHOWCHAR = 125 THEN COLOR 32 : PLOT 14,14 : PLOT 15,15 :
GOTO 80
75 COLOR SHOWCHAR : PLOT 14,14 : PLOT 15,15
80 COLOR SHOWCHAR+128 : PLOT 14,15 : PLOT 15,14
90 A = USR(BIGCHAR, CURCHAR) : REM DRAW CHARACTER
100 LOCATE COL,ROW,OLDCHAR
105 POSITION COL,ROW
109 REM WAIT FOR KEY PRESS
110 KEY = PEEK(764) : IF KEY = 255 THEN 117
115 ? CHR$(OLDCHAR) ; :COUNT = 19 : POKE 764,255 : POSITION
COL,ROW : GOTO 150
117 KEY = PEEK(53279) : IF KEY = 3 THEN 9700
```



The character editor program listed in this chapter will produce a screen that looks something like this—without the treasure map and the question mark. These are the inventions of the three young artists who created the other original designs in this book.

```
119 REM BLINK CURSOR
120 COUNT = COUNT + 1 : IF COUNT < 20 THEN 110
130 COUNT = 0 : IF CURSOR = 0 THEN CURSOR = 1 : ?
CHR$(OLDCHAR + 128) ; : GOTO 105
140 CURSOR = 0 : ? CHR$(OLDCHAR) ; : GOTO 105
150 IF KEY = 14 AND ROW > 11 THEN ROW = ROW - 1 : GOTO 100
160 IF KEY = 15 AND ROW < 18 THEN ROW = ROW + 1 : GOTO 100
180 IF KEY = 6 AND COL > 4 THEN COL = COL-1 : GOTO 100
190 IF KEY = 7 AND COL < 11 THEN COL = COL+1 : GOTO 100
200 IF KEY <> 33 THEN 220
203 GOSUB 2000 : POKE BYTE, USR(BITAND,MAP,255-MASK) : ? " ";
205 COL = COL+1 : IF COL > 11 THEN COL = 4 : ROW = ROW+1 :
IF ROW > 18 THEN ROW = 11
210 GOTO 100
220 IF KEY <> 34 THEN 230
222 GOSUB 2000 : POKE BYTE, USR(BITOR,MAP,MASK) : ? CHR$(20) ;
: GOTO 205
230 IF KEY <> 18 THEN 240
232 COLOUR = COLOUR + 1 : IF COLOUR > 15 THEN COLOUR = 0
235 GOSUB 2900 : GOTO 110
240 IF KEY <> 0 THEN 250
```

```
242 LUMINANCE = LUMINANCE + 2 : IF LUMINANCE > 14 THEN
LUMINANCE = 0
245 GOSUB 2900 : GOTO 110
250 IF KEY = 54 THEN GOSUB 2100 : GOTO 70
260 IF KEY = 55 THEN GOSUB 2200 : GOTO 70
270 IF KEY <> 142 THEN 280
272 GOSUB 2600 : FOR I = TARGET TO TARGET + 6
274 POKE I,PEEK(I+1) : NEXT I
276 POKE TARGET+7,0 : GOTO 90
280 IF KEY <> 143 THEN 290
282 GOSUB 2600 : FOR I = TARGET + 7 TO TARGET + 1 STEP -1
284 POKE I,PEEK(I-1) : NEXT I
286 POKE TARGET,0 : GOTO 90
290 IF KEY <> 134 THEN 300
292 GOSUB 2600 : FOR I = TARGET TO TARGET + 7
294 POKE I, USR(BITAND,PEEK(I)*2,255) : NEXT I
296 GOTO 90
300 IF KEY <> 135 THEN 310
302 GOSUB 2600 : FOR I = TARGET TO TARGET + 7
304 POKE I,INT(PEEK(I)/2) : NEXT I
306 GOTO 90
310 IF KEY <> 40 THEN 320
312 GOSUB 2600 : TARGET 2 = TARGET + OFFSET : FOR I = 0 TO 7
314 POKE TARGET + I, PEEK(TARGET2+I) : NEXT I : GOTO 90
320 IF KEY <> 46 THEN 330
322 GOSUB 2700 : PRINT "DO YOU WANT TO RESTORE THE
ENTIRE"
324 PRINT "CHARACTER SET (Y/N)"; : INPUT QUERY$
326 IF QUERY$ = "Y" THEN A = USR(MOVESET,OLDSET,NOOSET)
328 GOTO 2820
330 IF KEY <> 58 THEN 340
332 MODE = 0 : OPEN #1,8,0,"S:" : GOSUB 2900 : POKE 752,1
333 PRINT #1; "CURRENT COLOR: "; COLOUR:PRINT #1; "CURRENT
LUMINANCE: "; LUMINANCE
334 PRINT #1; "ATASCII CODE: "; SHOWCHAR
335 PRINT #1; "SCREEN CODE: "; CURCHAR : PRINT #1;
"CHARACTER DATA: "
336 GOSUB 2600 : FOR I = TARGET TO TARGET+7 : PRINT #1;
PEEK(I); ", "; : NEXT I
337 PRINT #1; CHR$(155); : CLOSE #1 : IF MODE = 0 THEN GOTO
2800
338 TRAP 65535 : GOTO 110
340 IF KEY = 10 THEN MODE = 1 : TRAP 9500 : OPEN #1,8,0, "P:" :
GOTO 333
350 IF KEY <> 62 THEN 370
352 TRAP 9500 : GOSUB 2700 : ? "SAVE TO <C>ASSETTE OR
<D>ISK"; : INPUT QUERY$
```

```
354 IF QUERY$ = "C" THEN OPEN #1,8,0,"C:" : GOTO 360
356 IF QUERY$ <> "D" THEN 358
357 ? "FILENAME"; : INPUT TEMP$ : NAME$(3,14) = TEMP$ : OPEN
#1,8,0,NAME$ : GOTO 360
358 TRAP 65535 : GOTO 2820
360 FOR I = 0 TO 1023 : PUT #1,PEEK(NOOSET+I) : NEXT I : CLOSE
#1 : GOTO 358
370 IF KEY <> 61 THEN 390
372 TRAP 9500 : GOSUB 2700 : ? "LOAD FROM <C>ASSETTE OR
<D>ISK"; : INPUT QUERY$
374 IF QUERY$ = "C" THEN OPEN #1,4,0,"C:" : GOTO 380
376 IF QUERY$ <> "D" THEN 378
377 ? "FILENAME"; : INPUT TEMP$ : NAME$(3,14) = TEMP$ : OPEN
#1,4,0,NAME$ : GOTO 380
378 TRAP 65535 : GOTO 2820
380 FOR I = 0 TO 1023 : GET #1,IN : POKE NOOSET+I,IN : NEXT I :
CLOSE #1 : GOTO 378
390 IF KEY <> 118, THEN 400
392 GOSUB 2600 : FOR I = TARGET TO TARGET + 7 : POKE I,0 :
NEXT I : GOTO 70
400 IF KEY = 5 THEN GOTO 1000
999 GOTO 105
1000 ROW = 11 : COL = 18 : POKE 752,0
1005 POSITION COL-1,ROW : ? CHR$(31);
1010 GET #2,KEY
1020 IF KEY = 28 AND ROW > 11 THEN ROW = ROW-1 : ?
CHR$(KEY);
1030 IF KEY = 29 AND ROW < 18 THEN ROW = ROW+1 : ?
CHR$(KEY);
1040 IF KEY = 30 AND COL > 18 THEN COL = COL-1 : ?
CHR$(KEY);
1050 IF KEY = 31 AND COL < 35 THEN COL = COL+1 : ?
CHR$(KEY);
1060 IF KEY <> 155 THEN 1070
1065 LOCATE COL,ROW,OLDCHAR : POKE 752,1 : IF OLDCHAR>127
THEN ? CHR$(OLDCHAR-128) : GOTO 1068
1067 PRINT CHR$(OLDCHAR+128)
1068 COL = 4 : ROW = 11 : GOTO 100
1070 IF KEY < 27 OR (KEY > 31 AND KEY < 125) OR (KEY > 127
AND KEY < 155) OR (KEY > 159 AND KEY < 253) THEN GOTO 1090
1080 GOTO 1005
1090 PRINT CHR$(KEY); : COL = COL+1 : IF COL > 35 THEN
COL = 18 : ROW = ROW+1 : IF ROW > 18 THEN ROW = 11
1100 GOTO 1005
2000 BYTE = NOOSET + CURCHAR * 8 + ROW - 11 : BIT =
COL - 4 : MASK = 2^(7-BIT) : MAP = PEEK(BYTE) : RETURN
```

```
2100 CURCHAR = CURCHAR - 1 : IF CURCHAR < 0 THEN
CURCHAR = 127
2110 GOSUB 2500 : RETURN
2200 CURCHAR = CURCHAR + 1 : IF CURCHAR > 127 THEN
CURCHAR = 0
2210 GOSUB 2500 : RETURN
2300 CURCHAR = CURCHAR - 32 : IF CURCHAR < 0 THEN
CURCHAR = CURCHAR + 128
2310 GOSUB 2500 : RETURN
2400 CURCHAR = CURCHAR + 32 : IF CURCHAR > 127 THEN
CURCHAR = CURCHAR - 128
2410 GOSUB 2500 : RETURN
2500 IF (CURCHAR > = 0) AND (CURCHAR < 64) THEN SHOWCHAR
= CURCHAR + 32 : RETURN
2510 IF (CURCHAR > 63) AND (CURCHAR < 96) THEN SHOWCHAR
= CURCHAR - 64 : RETURN
2520 SHOWCHAR = CURCHAR : RETURN
2600 TARGET = NOOSET + CURCHAR * 8 : RETURN
2700 ? CHR$(125); : POKE 756,INT(OLDSET/256) : RETURN
2800 ? :? "PRESS RETURN TO CONTINUE"
2810 K = PEEK(764) : IF K <> 12 THEN 2810
2820 ? CHR$(125) ; : POKE 756,INT(NOOSET/256) : GOTO 35
2900 SETCOLOR 2,COLOUR,LUMINANCE : SETCOLOR 4,
COLOUR,LUMINANCE : RETURN
5000 GRAPHICS 0 : POKE 752,1 : OPEN #2,4,0,"K:"
5010 POSITION 9,10 : ? "ATARI CHARACTER EDITOR"
5020 POSITION 12,11 : ? "NOW INITIALIZING"
5030 DIM HEX$(2), HEX1$(1), HEX2$(1), QUERY$(1),
NAME$(14),TEMP$(12)
5035 NAME$(1,2) = "D:"
5040 PRTCHARS = 1536 : BIGCHAR = 1586 : MOVESET = 1698 :
BITAND = 1730 : BITOR = 1746
5050 CURCHAR = 1 : SHOWCHAR = 33 : COLOUR = 10 :
LUMINANCE = 0
5060 OLDSET = PEEK(756)*256 : NOOSET = (PEEK(106)-4)*256 :
OFFSET = OLDSET - NOOSET
5070 COL = 4 : ROW = 11
5100 GOSUB 9000 : REM POKE ML ROUTINES
5110 POKE 106,PEEK(106)-4 : GRAPHICS 0 : POKE 752,1 : REM SET
TOP OF MEMORY
5120 A=USR(MOVESET,OLDSET,NOOSET) : POKE
756,INT(NOOSET/256)
5999 RETURN
6999 REM ** DRAW BORDERS ON SCREEN
7000 COLOR 17 : PLOT 3,0
7010 COLOR 18 : PLOT 4,0 : DRAWTO 35,0
```

```
7020 COLOR 5 : PLOT 36,0
7030 COLOR 124 : PLOT 3,1 : DRAWTO 3,8 : PLOT 36,1 :
DRAWTO 36,8
7040 COLOR 26 : PLOT 3,9
7050 COLOR 18 : PLOT 4,9 : DRAWTO 35,9
7060 COLOR 3 : PLOT 36,9
7070 COLOR 17 : PLOT 3,10
7080 COLOR 18 : PLOT 4,10 : DRAWTO 11,10
7090 COLOR 5 : PLOT 12,10
7100 COLOR 124 : PLOT 3,11 : DRAWTO 3,18 : PLOT 12,11 :
DRAWTO 12,18
7110 COLOR 26 : PLOT 3,19
7120 COLOR 18 : PLOT 4,19 : DRAWTO 11,19
7130 COLOR 3 : PLOT 12,19
7140 COLOR 17 : PLOT 17,10
7150 COLOR 18 : PLOT 18,10 : DRAWTO 35,10
7160 COLOR 5 : PLOT 36,10
7170 COLOR 124 : PLOT 17,11 : DRAWTO 17,18 : PLOT 36,11 :
DRAWTO 36,18
7180 COLOR 26 : PLOT 17,19
7190 COLOR 18 : PLOT 18,19 : DRAWTO 35,19
7200 COLOR 3 : PLOT 36,19
7999 RETURN
8800 HEX1$ = HEX$(1,1) : HEX2$ = HEX$(2)
8810 DEC1 = ASC(HEX1$)-48 : IF DEC1 > 9 THEN DEC1 =
DEC1 - 7
8820 DEC 2 = ASC(HEX2$)-48 : IF DEC2 > 9 THEN DEC2 =
DEC2 - 7
8830 DEC = DEC1 * 16 + DEC2 : RETURN
9000 FOR I = 1536 TO 1761
9010 READ HEX$ : GOSUB 8800 : POKE I,DEC
9020 NEXT I
9030 RETURN
9500 CLOSE #1 : GOSUB 2700 : ERROR = PEEK(195)
9510 IF ERROR = 138 OR ERROR = 130 THEN ? "DEVICE DOES NOT
RESPOND" : GOTO 2800
9530 IF ERROR = 136 THEN ? "BAD FILE" : GOTO 2800
9540 IF ERROR = 165 THEN ? "BAD FILE NAME" : GOTO 2800
9550 IF ERROR = 162 THEN ? "DISK FULL" : GOTO 2800
9560 IF ERROR = 167 THEN ? "FILE LOCKED" : GOTO 2800
9570 IF ERROR = 170 THEN ? "FILE NOT FOUND" : GOTO 2800
9580 IF ERROR = 180 THEN ? "DIRECTORY FULL" : GOTO 2800
9590 ? "ERROR NUMBER ";ERROR;" HAS OCCURRED" : GOTO 2800
9700 GOSUB 2700 : POSITION 13,0 : ? "INSTRUCTIONS" : ?
9710 ? "ARROW KEYS (UNSHIFTED) : MOVE CURSOR"
9720 ? "CONTROL/ARROW KEYS: MOVE CHARCTER"
9730 ? "CLEAR: ERASE CHARACTER"
```

```
9740 ? ".: ADD DOT                    SPACE: ERASE DOT"
9750 ? "<: PREV CHARACTER             >: NEXT CHARACTER"
9760 ? "D: DATA TO SCREEN             P: DATA TO PRINTER"
9770 ? "C: CHANGE COLOR               L: CHANGE LUMINANCE"
9780 ? "S: SAVE CHARSET               G: GET OLD CHARSET"
9790 ? "R: RESTORE CHAR               W: RESTORE CHAR SET"
9800 ? "K: SKETCH MODE"
9850 GOTO 2800
10000 REM ** MACHINE LANGUAGE ROUTINE
10010 REM ** TO DISPLAY CHARACTER SET
10020 DATA 18, A5, 58, 69, 2C, 85, CB, A5
10030 DATA 59, 69, 00, 85, CC, A9, 00, 85
10040 DATA CD, A2, 08, A0, 00, A5, CD, 91
10050 DATA CB, C8, E6, CD, C0, 20, D0, F5
10060 DATA A9, 28, 18, 65, CB, 85, CB, A9
10070 DATA 00, 65, CC, 85, CC, CA, D0, E3, 68, 60
10075 REM **MACHINE LANGUAGE ROUTINE
10077 REM **TO DRAW ENLARGED CHAR
10080 DATA 68, 68, AD, F4, 02, 85, CC, A9, 00, 85
10090 DATA CB, 85, CD, 68, 0A, 26, CD, 0A
10100 DATA 26, CD, 0A, 26, CD, 18, 65, CB
10110 DATA 85, CB, A5, CD, 65, CC, 85, CC
10120 DATA A5, 58, 85, CD, A5, 59, 85
10130 DATA CE, 18, A9, BC, 65, CD, 85, CD
10140 DATA A9, 01, 65, CE, 85, CE, A0, 00
10150 DATA A9, 80, 85, CF, B1, CB, 84, D1
10160 DATA A0, 00, 25, CF, F0, 05, A9, 54, 38, B0, 02, A9, 00
10170 DATA 91, CD, A4, D1, E6, CD, D0, 02
10180 DATA E6, CE, 46, CF, 90, E1, A9, 20
10190 DATA 18, 65, CD, 85, CD, A9, 00, 65
10200 DATA CE, 85, CE, C8, 98, C9, 08, D0
10210 DATA CA, 60
10220 REM **MACHINE LANGUAGE ROUTINE
10230 REM **TO MOVE CHARACTER SET
10240 REM **FROM ROM TO RAM
10250 DATA 68, 68, 85, CC, 68, 85, CB, 68
10260 DATA 85, CE, 68, 85, CD, A2, 04, A0
10270 DATA 00, B1, CB, 91, CD, C8, D0, F9
10280 DATA E6, CE, E6, CC, CA, D0, F2, 60
10290 REM **MACHINE LANGUAGE ROUTINE
10300 REM **TO PERFORM BITWISE AND
10310 DATA 68, 68, 68, 85, CB, 68, 68, 25
10320 DATA CB, 85, D4, A9, 00, 85, D5, 60
10330 REM **MACHINE LANGUAGE ROUTINE
10340 REM **TO PERFORM BITWISE OR
10350 DATA 68, 68, 68, 85, CB, 68, 68, 05
10360 DATA CB, 85, D4, A9, 00, 85, D5, 60
```

When you RUN this program, you will be presented (after a fairly lengthy initialization period) with a display that looks like the one on page 108. At the top of the screen, inside a large rectangle, is the complete Atari character set, in order according to screen code. Below this are two more rectangles, the one to the left containing an expanded image of the exclamation point character (screen code 1) and the one to the right containing a considerable amount of blank space. Between these two rectangles, four normal-sized exclamation points have been printed in a small square, two of them reversed and two of them normal. In the upper left-hand corner of the left-hand rectangle, a cursor is blinking.

The expanded exclamation point in the left-hand rectangle is based on the bitmap of the character as it is stored in memory. The circles that make up the body of the exclamation point are the 1's in the bitmap and the blank spaces surrounding the exclamation point are the 0's.

At the bottom of the display, you should see the message "PRESS 'OPTION' FOR INSTRUCTIONS". This refers to the OPTION key on the right-hand side of the keyboard. Press this key and you will be presented with this menu of commands:

```
                    INSTRUCTIONS

ARROW KEYS (UNSHIFTED): MOVE CURSOR
CONTROL/ARROW KEYS: MOVE CHARACTER
CLEAR: ERASE CHARACTER
.: ADD DOT                  SPACE: ERASE DOT
<: PREV CHARACTER           > : NEXT CHARACTER
D: DATA TO SCREEN           P: DATA TO PRINTER
C: CHANGE COLOR             L: CHANGE LUMINANCE
S: SAVE CHARSET             G: GET OLD CHARSET
R: RESTORE CHAR             W: RESTORE CHAR SET
K: SKETCH MODE
```

Press the RETURN key and you will be returned to the character set display. Using the commands shown on the instruction menu, you can edit the enlarged character in the left-hand rectangle. For instance, pressing the period (".") key, you can add new pixels to the character, at the position in the bitmap directly underneath the flashing cursor. By pressing the spacebar, you can erase the pixels underneath the cursor.

Pressing one of the four cursor arrow keys while *not* holding down any other keys will move the cursor around the bitmap. Pressing the cursor arrow keys while holding down the CONTROL key will move the character itself. Pressing the CON-

TROL-up arrow, for instance, will shift every line of the arrow character up one line, erasing the topmost line and adding a blank line at the bottom.

If you do not wish to edit the exclamation point character, pressing the > key (on the top row of the keyboard) will display an enlarged image of the next character in the character set. Hold this key down continuously to move rapidly through the entire character set. To move through the character set in the opposite direction, hold down the < key. As the enlarged image of each character appears, four normal-sized copies of the character will be automatically displayed between the two lower rectangles.

To see a copy of the eight numbers that constitute the bitmap for the current character, press the D key. This displays the bitmap information, plus the current color and luminance of the display, and the ATASCII and screen codes of the current character. If you have a printer attached to your computer, you can print this information by pressing the P key instead of the D key.

To change the color of the display, press the C key. By holding down this key continuously, you can cycle through all sixteen colors. Similarly, the L key will cycle through all eight luminances.

The S key will save the entire custom character set to either tape or disk, directly from the Atari's memory. When you choose this option, you will be asked whether you wish to save to <C>ASSETTE or <D>ISK. Press C for cassette or D for disk, and press RETURN. (Press RETURN by itself to return to the character set display.) If you choose disk, you will be further prompted for a filename. Use any valid Atari filename, as defined in your Atari disk manual.

To load a previously saved character set from tape or disk, use the L key. Once again, you will be prompted for a C or a D and a filename if you choose the latter.

The R key will restore the current character to its normal form. The W key will restore the entire character *set* to its normal form. If you request the latter option, you will be asked to verify the request, since the accidental restoration of the character set could destroy hours (or at least minutes) of work.

Pressing the K key will place you in sketchpad mode. The cursor will disappear from the left-hand rectangle and reappear in the right-hand rectangle. This is the "sketch pad." You may now type characters in a manner that is quite similar to the normal BASIC editing mode. Pressing CONTROL and a cursor arrow key will move the cursor. Pressing a key that normally produces a character will produce that character. Pressing RETURN will return you to the normal character editing mode. Use the sketch-

pad mode to try out combinations of the characters that you have created in the character editing mode.

USING THE DATA

To use the character data printed by the D command, you will need to write a loop that will read that data from a DATA statement and POKE it into the proper eight bytes of the character set memory. Here is a routine that will do exactly that, if SCRNCODE is equal to the screen code of the character to be modified, CHARSET is equal to the location of the new character set, and DATALINE is equal to the number of the line with the DATA statement containing the character DATA:

```
100 RESTORE DATALINE : FOR I = 0 TO 7
110 READ BYTE : POKE CHARSET+SCRNCODE*8, BYTE
120 NEXT I
```

This routine assumes that the character data is the first eight numbers in the DATA statement on line DATALINE.

With the aid of the character set editor, however, it may not be necessary to place character data in DATA statements within your program. Instead, you can simply save an entire character set to disk or tape and write a routine in your program that will load the set back into memory. Bear in mind that the character set is stored on disk or tape as a sequence of 1,024 bytes, which can be loaded back into memory with a simple input loop that pokes the bytes into place.

Here is an example of such a routine for a character set on tape, where CHARSET is, once again, the address where you wish to place the modified character set:

```
1000 OPEN #1, 4, 0, "C:"
1010 FOR I = 0 TO 1023
1020 GET #1, IN : POKE CHARSET+I, IN : NEXT I
1030 CLOSE #1
```

This routine expects to find a tape already in the cassette recorder, with the character set file recorded on it.

Here is a version of the routine modified to work with a disk drive:

```
1000 OPEN #1,4,0,"D:FILENAME"
1010 FOR I = 0 TO 1023
1020 GET#1,IN : POKE CHARSET+I, IN : NEXT I
1030 CLOSE #1
```

The word FILENAME in line 1000 should be replaced by the name of the file in which you have saved your character set. The disk containing this file must be in drive 1 at the time this program is executed. If the disk will be in drive 2, the letter D in front of the file name should be changed to D2.

CUSTOM GRAPHICS CHARACTERS

Creating graphics with a redefined character set is not difficult. In fact, it is exactly like working with the character graphics described earlier, only now you can create your own characters, rather than relying on Atari's. As soon as you POKE the new character set address, divided by 256, into address 756 and lower the top of memory, you can use the new set exactly as you would the built-in set, with the text mode commands that work with standard characters. Here is the procedure for creating custom graphics:

1. Design the new set using the character editor (easy) or graph paper (harder).

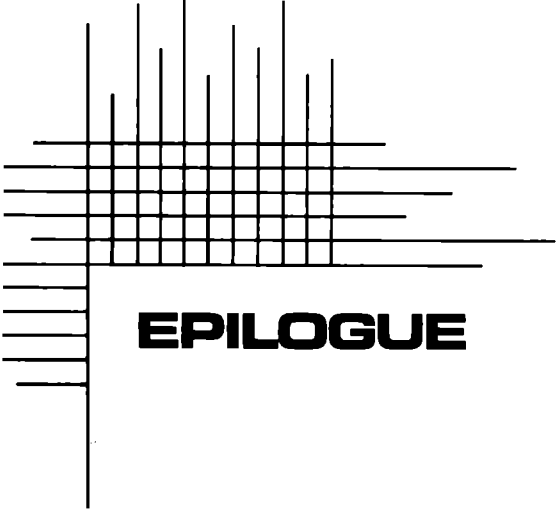
2. Create the bitmaps for that character set. If you are using graph paper, you will need to create the bitmaps by hand. Every row of eight pixels in the character must be translated into a single binary number, with foreground color pixels represented by 1s and background color pixels represented by 0s. You must then translate the binary numbers into decimal numbers—not an easy task, and one that we will not explain further in this book. If you use the character editor, this translation will be performed for you automatically. By using the D command in the editor, you can call up a decimal translation of the bitmap of the character currently displayed for editing. The S command enables you to save bitmaps for *all* characters in the set in a file on a disk or tape.

3. Put the bitmaps back into the Atari's memory. When you are ready to run a program that uses the custom character set, you will need to put the bitmaps for that character set back into the Atari's RAM. There are two common ways of doing this. You can move the existing bitmaps into RAM, then alter selected characters with bitmaps that you have placed in DATA statements in your program, using the READ and POKE commands to move the bitmaps into RAM starting at any available address divisible by 256, as described above. If you have used the editor to save entire bitmaps on disk or tape, you can read those bitmaps back off the disk and into RAM using the subroutines described above.

Suggested Projects

1. Books and newspapers have their own "character sets," much as the Atari does; that is, they use a distinctive style of type in which each letter of the alphabet or other symbol has its own unique design. Examine some of these character sets, and use the character set editor to create your own Atari character set in imitation of these designs. Look through a number of books, magazines and other sources to find particularly striking character designs and adapt the best features of these designs to your original character set.

2. Create an original graphics character set for the Atari, from which you can piece together detailed drawings of houses, animals, people, and other pictorial designs.



EPILOGUE

Atari graphics is a big subject. We haven't yet said everything there is to say about it. For instance, we lack the space here to discuss Player-Missile graphics, an elaborate animation system that allows the Atari programmer to create movable graphic images that are completely separate from the images in video memory and that can be moved around the display through a series of PEEK and POKE instructions. Be warned, however, that Player-Missile graphics cannot be fully utilized from an Atari BASIC program and requires machine language, or a faster high-level language, for full effect.

We also haven't mentioned display list interrupts, a programming trick that can be used to squeeze extra colors onto an Atari bitmap display. In fact, display list interrupts can be used to display all 128 (or even 256) Atari colors at one time. This technique also requires a knowledge of machine language.

In this book, you have gotten a glimpse of what Atari graphics can do, and a working knowledge of the techniques required to create a graphics program. It is now up to you to decide what you want to do with this knowledge. Any program can benefit from graphics, if only in the creation of an impressive title screen. For instance, while a program loads from the Atari disk drive it is sometimes useful to draw a picture on the display to keep the user company, or prevent him or her from getting bored.

Have fun!

BIBLIOGRAPHY

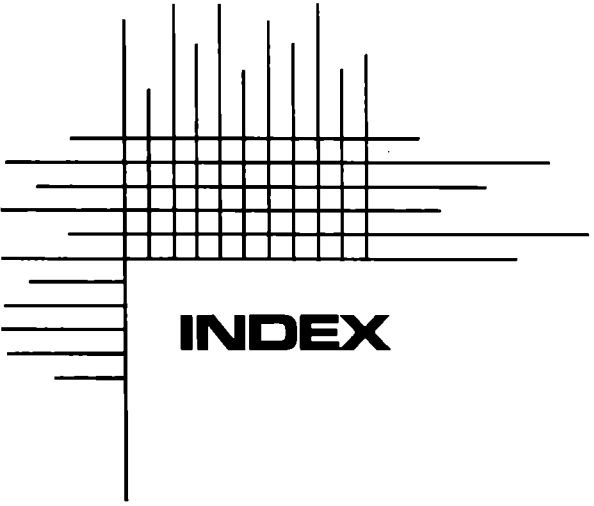
Your Atari Computer by Lon Poole with Martin McNiff & Steven Cook. OSBORNE/McGraw-Hill, Berkeley, CA, 1982. An excellent reference book on general programming in Atari BASIC. Includes two chapters on graphics programming, including discussions of both OS and Antic/GTIA graphics.

Compute!'s First/Second/Third Book of Atari. Compute! Publications, Inc., Greensboro, NC. Three collections of articles, many concerned with graphics programming.

Compute!'s First/Second Book of Atari Graphics. Compute! Publications, Inc., Greensboro, NC. Two collections of articles specifically relating to graphics programming techniques on Atari computers.

Atari Color Graphics: A Beginner's Workbook by Joseph W. Collins. Arrays, Inc., Los Angeles, CA, 1984. A general introduction to OS supported graphics on the Atari, notable for its detailed examinations of each of the OS graphics modes.

De Re Atari. Atari Inc., 1981. The definitive collection of essays on advanced Atari programming techniques. Available directly from Atari.



INDEX

ABOUT THE AUTHOR

Christopher Lampton is the author of more than twenty books for Franklin Watts, including a number of popular First Book and Impact titles. He has written all the books on computer languages and graphics in Watts' Computer Literacy Skills series.

Chris first became a computer enthusiast when he purchased a Radio Shack computer to use for word processing. He now owns eight computers.

Chris has a degree in radio and television broadcasting and lives right outside Washington, D.C. In addition to his books in the area of science and technology, he has written four science-fiction novels.

FRANKLIN WATTS
387 PARK AVENUE SOUTH
NEW YORK, NEW YORK 10016

JACKET DESIGN BY GINGER GILES

PRINTED IN THE UNITED STATES OF AMERICA BY MOFFA PRESS, INC.

FRANKLIN WATTS' COMPUTER LITERACY SKILLS SERIES

ADVANCED BASIC

BASIC FOR BEGINNERS

COBOL FOR BEGINNERS

FORTH FOR BEGINNERS

FORTRAN FOR BEGINNERS

GRAPHICS AND ANIMATION ON THE APPLE

GRAPHICS AND ANIMATION ON THE ATARI

GRAPHICS AND ANIMATION ON THE COMMODORE 64

GRAPHICS AND ANIMATION ON THE TRS-80

PASCAL FOR BEGINNERS

PILOT FOR BEGINNERS

6502 ASSEMBLY-LANGUAGE PROGRAMMING

Z80 ASSEMBLY-LANGUAGE PROGRAMMING

531-10144-4
A07