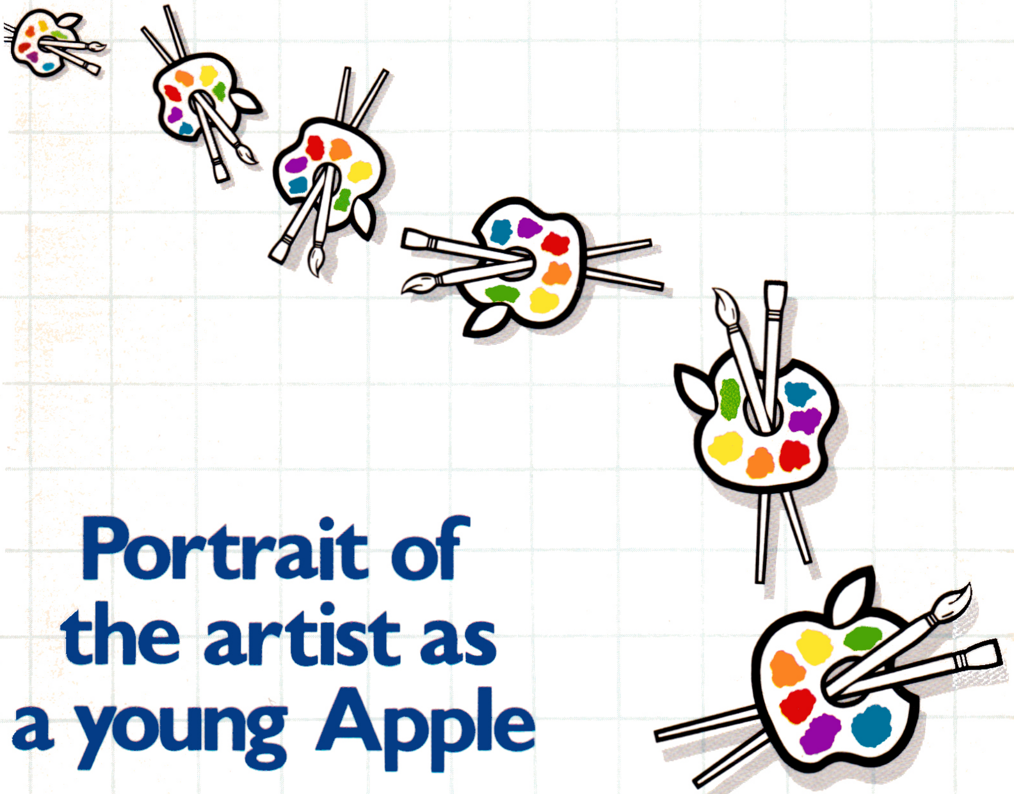
GRAPHICALLY SPEAKING



Portrait of the artist as a young Apple

by Mark Pelczarski

SOFTALK BOOKS

$19.95

GRAPHICALLY SPEAKING

GRAPHICALLY SPEAKING

Portrait of the artist as a young Apple

by Mark Pelczarski

SOFTALK BOOKS

Current Printing (last digit)
10 9 8 7 6 5 4 3 2 1

Softalk Books
P.O. Box 60
North Hollywood, CA 91603

Design by Kurt A. Wahlner

Library of Congress Cataloging in Publication Data

Pelczarski, Mark, 1954-
Graphically speaking.

Includes index.
1. Computer graphics. 2. Apple computer—Programming.
I. Title.
T385.P45 1983 001.64'43 83-20299
ISBN 0-88701-007-5

Program disk is available with this book. It contains all program listings in machine readable format (Apple II, II+, or IIe), plus a nifty logo and an easy-to-use menu.

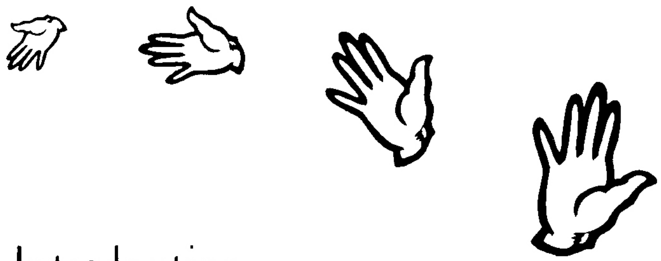
To Cheryl, for going along with all the craziness, and to all the "Penguins" for contributing to it.

And to all those who share as they learn and discover.

GRAPHICALLY SPEAKING

CONTENTS

ONE



An Introduction

This book is about creating computer graphics. Specifically, it's about ***high-resolution*** (hi-res) graphics on Apple computers, but the principles involved are similar to those used in most computer graphics. Parts of the book are about programming in Basic, parts are about machine-language graphics, and some sections contain ready-to-use programs that you can type in and run. We'll look at line drawing, vector shapes, block and character graphics, animation, extra colors and color-filling, and 3-D graphics. We'll also look at how Apple colors are stored on the screen, and at ways that you can store your graphics screens very compactly.

It's recommended that you type in the examples as you read; it makes it much easier to learn about computers if you can see what they do. The examples in Basic you can type right in. For the machine-language examples, although you can enter the numbers directly into the computer, a good *assembler* program is recommended, such as the one in *Applesoft/ DOS Toolkit,* or in *Merlin*. The assembly listings in this book use *Merlin*. Many of the examples are excerpts from the programs *The Complete Graphics System* and *The Graphics Magician* from Penguin Software, and I'd like to thank David Lubar and Dav Holle, co-authors of some of the routines, for their contributions and neat ideas about graphics. If you do use any of the routines from this book in your own programs, we just ask that you acknowledge the authorship of those routines in your program.

Try not to ask "why?" too much when looking at some of the more odd conventions of Apple graphics, it's easy to ask why now, but take a moment instead to appreciate the foresight used in the development of the Apple to include them at all. A few years ago, most state of the art Apple games were done in the 16-color, low-resolution block graphic mode, and we were impressed. Who would have guessed that hi-res graphic games such as *Raster Blaster* were hiding in the Apple, just waiting to get out, that long ago? Today, various graphic games on the Apple rival their big brothers in the arcades, and the capabilities for those were put there several years back. Amazing.

Hi- and Lo-res

There are two types of graphics built into the Apple. There's a *low resolution* mode that has 16 colors, with the smallest unit being a block that's half the height and the same width as a text character. (You, too, can use computer jargon now, by calling this mode *lo-res*). In the lo-res mode, the screen is 40 blocks wide and 48 blocks in height (text mode has 40 characters across and 24 lines of type down the screen). There is also a high resolution mode that uses six colors (the books tell you eight, but they count black and white twice each, for a reason we'll see later). The ***hi-res*** mode lets you access 280 points across the screen and 192 points down (a single text character is made up on a grid seven points across and eight down). It's the hi-res mode that we'll discuss in this book, since it provides much more flexibility with individual access to every point.

There are several commands in Applesoft Basic that give you control over both hi-res and lo-res graphics, but before we get into things you can do with these commands, it may be a good idea to talk about how the graphics work internally and about some of the oddities you will encounter. It may also be a little backwards, talking about bits and bytes before using the handy-dandy, ready-to-go Applesoft graphic commands, but it will also answer a lot of questions before they arise, as well as convince some of the more advanced among you that this isn't just going to be a book about using HPLOT and shape tables.

The hi-res graphics screen, 280 points by 192 points, corresponds directly to a portion of memory in the Apple. Every one of these points corresponds to a certain bit in memory and can be either on or off (lit or black).

A bit of binary

A bit is the smallest unit of memory in a computer, and it is the result of an electrical state that is in one of two positions. Mathematicians gave numbers to those two positions, 0 and 1, and hence computers use binary arithmetic, which is base 2 (as opposed to our usual base 10). Eight bits in a group are a byte, and in a byte you can store the numbers 0 (00000000, each bit off) through 255 (11111111, each bit set). Appendix A elaborates a little more on binary arithmetic. If you are unfamiliar with binary, you should read through it before going on. It does come in handy when you want to do some tricky graphics, although if you sensibly stick to Applesoft graphics commands for a while it's not all necessary.

Memory allocation

Memory consists of about 64,000 bytes (64K bytes) in most Apples. Each byte has a memory address that lets you refer to it, so you can store things in that location or look and see what's there. When using Applesoft Basic, the top 16K of addresses have programs permanently stored there (the programs that make Applesoft and all the "little" functions, like reading keys and putting letters on the screen work). That type of memory is known as ROM, or read-only memory. The rest is RAM, or random access memory, in which you can store things and read things to your heart's content. That's the area where programs and data go when they get read from disk.

After subtracting the 16K of ROM for Applesoft, you have 48K of RAM. The hi-res graphics screen corresponds to 8K of RAM, or 8,192 bytes (although only 7,680 are actually used). There are two areas of memory that can be used for hi-res graphics, called *hi-res page 1* and *hi-res page 2*. The addresses for page 1 are 8192 through 16383, and the addresses for page 2 are 16384 through 24575.

Screen Display

What does this all mean? Not much, except it perhaps convinces you that there's an actual area in the memory of your computer that has a one-to-one relation with what you see on the hi-res screen. To illustrate graphically, here's a program that doesn't do anything particularly useful, but does show you that there's nothing magical about displaying graphics:

```
10   HGR : POKE - 16301,0
20   FOR L = 8192 TO 16383
30   POKE L,1
40   NEXT L
50   TEXT
```

HGR is an Applesoft command that switches the display to hi-res page 1 and turns off every bit in page 1 graphics memory, thus clearing the screen to black. The POKE command makes the bottom four lines of text disappear; more on that later. The loop defined at lines 20 and 40 says to repeat the statements in between with the variable L starting at 8192, increasing by 1 each time, until it reaches 16383 (conveniently, the addresses in graphics page 1 go from 8192 to 16383).

The statement POKE L, 1 says to put the value 1 into the byte with address L. The result should be to turn on one of the eight bits in that byte, and hence turn on one point on the screen. The last statement switches the display back to text, so that you don't think your computer's disappeared into Never-Never Land. Actually, a good hearty Reset after things stop happening on the screen would do the same trick.

Try running the program to see what happens. When you put the number 1 into all those locations, a bunch of points should light up. Try changing the 1 in the POKE statement to 255. That should set every point. How do you get colors? Try using the numbers 42, 85, 170, and 213, which are various concoctions of every other bit being set (00101010, 01010101, 10101010, and 11010101, in that order).

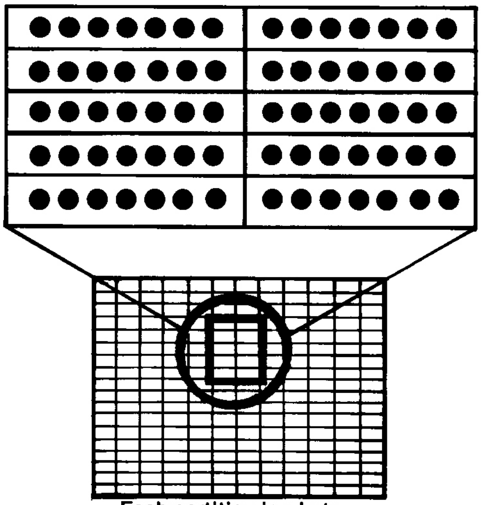


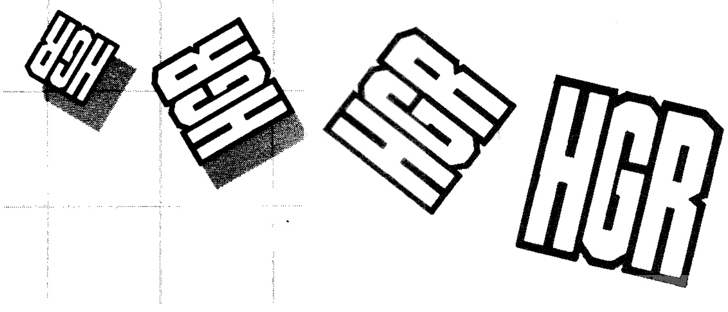
Figure 1.1 A Portion of the Hi-Res Graphics Screen

A bit of color

The trick is that points themselves are either on or off. Color depends on two things: position and the leftmost bit. Only seven of the eight bits in each byte are displayed on the screen. The leftmost bit controls the color of the other seven. Bits in even columns on the screen are violet or blue when they're turned on. Bits in odd columns are green or orange when on. If the leftmost bit (high bit) is off, the colors are violet and green. If the high bit is set (on), the colors are blue and orange. There is no actual white. When blue and orange are next to each other, they appear white; likewise with violet and green. Hence, there are two different whites possible, and similarly, two different blacks (high bit on and high bit off).

Try different numbers to see what happens. For those who want a challenge, try to find a way to poke in values that will make the entire screen orange or blue, and so on. The next chapter will deal with more conventional graphics commands through Applesoft before we eventually find our way back to bit graphics.

TWO



Applesoft Graphics Commands

Applesoft Basic has some nice built-in commands for using hi-res graphics. Although they're described in the Apple II *Applesoft BASIC Programming Reference Manual* (and the IIe Manual)[1], a little repetition doesn't hurt.

Each example has a line number, as it would appear in a Basic program. The line numbers used are arbitrary, though. All the commands can also be used without line numbers as direct commands, so you don't even have to write a program to draw on the screen.

Graphics Commands

HGR

The command HGR sets the display to show whatever is in the hi-res page 1 memory area (addresses 8192 to 16383). It also clears the screen (sets all values in that address range to zero) and sets a pointer that tells all subsequent hi-res commands to draw on page 1. The syntax is:

```
10    HGR
```

1. *Apple II Applesoft BASIC Programming Reference Manual* (Cupertino: Apple Computer, Inc., 1979, 1981)
Kamins, Scot, *Apple IIe Applesoft BASIC Programmer's Reference Manual* (Cupertino: Apple Computer, Inc., 1982)

HGR2

Another command, HGR2, sets the display to show what's in hi-res page 2 (16384 to 24575), clears the screen, and sets the draw pointer to page 2. You'd type it:

```
10   HGR2
```

HCOLOR

HCOLOR sets the color of subsequent draws and plots to the hi-res screen. Colors are:

	High bit off	*High bit on*
No dots set	0 - black	4 - black
Odd dots set	1 - green	5 - orange
Even dots set	2 - violet	6 - blue
All dots set	3 - white	7 - white

Remember that white and black have twice the resolution of the other colors (280 dots across, as opposed to 140) because "white" is every dot set, while any of the other colors have only even or odd dots set. Also, colors from the left column may affect colors in the right column (and vice versa) when positioned close together horizontally (within the same byte), since the high bit controls the colors used for the whole byte. Here's how you set color:

```
20   HCOLOR = 5
```

HPLOT

HPLOT X,Y sets the point X,Y to the current HCOLOR (the last one specified with an HCOLOR command). X can be from 0 to 279, with 0 being the left edge of the screen and 279 the right edge, and Y can be from 0 to 191, with 0 the top and 191 the bottom of the screen. Any arithmetic expression can also be used for X and Y, as long as the resulting values are in the ranges given. If not, you'll get an error in the program.

Since HPLOT sets only a single point, though, either of the two whites will appear as another color. White3 will appear green if X is odd, and violet if X is even, and white7 will be orange if X is odd, and blue if X is even. Also, if HCOLOR = 5 (orange), for example, the HPLOT command will only set dots in odd columns (since orange only appears in odd columns). HPLOTing in an even column will leave that dot off. Similar results occur with green, violet, and blue.

```
30    HPLOT 30, 120
40    HPLOT R*2, (T-5)/3
```

HPLOT X1, Y1 TO X2, Y2 draws a line from point X1, Y1 to point X2, Y2 in the current HCOLOR. The same restrictions apply to the range of the X and Y values as in the HPLOT X,Y command. The color restrictions of the Apple show when lines are vertical or near vertical. If both X values are the same and the HCOLOR is white, you'll get the same color results as explained with HPLOTing a single point. If you try to draw a vertical orange or green line with the X value even, nothing will happen (since orange and green only appear in odd columns). Likewise, if you try to draw a vertical blue or violet line in an odd column, it won't work. Lines that are near vertical will often appear broken or in multiple colors for the same reasons.

```
35    HPLOT 5, 10 TO 260, 180
40    HPLOT 2*D, 5+F TO 3 -N, L/2
```

HPLOT TO

HPLOT TO X,Y draws a line from the last point specified in a previous HPLOT command to X,Y. All the above comments about HPLOT commands apply.

```
45    HPLOT TO 45, 50
50    HPLOT TO A + 19, B-8
```

The Applesoft commands just outlined take care of setting the individual bytes in the hi-res screen area appropriately. Considering the examples in Chapter 1, in which we were poking values into various bytes of the hi-res screen, this is a nice convenience. There are several hi-res commands dealing with Applesoft shape tables, too, but we'll talk about those later. In the meantime, here are a few program examples that use the HPLOT commands.

Making Rectangles

The programs in Listing 2.1 all draw a rectangle on hi-res screen 1 in white. If you try them, note that the vertical lines will not appear in white. Try changing white7 to white3 to see the results. Each program uses a slight variation on the HPLOT command to achieve the same result. Note that the usage in the third of the examples is legal and works just fine.

The HPLOT command also lends itself well to use of the READ and DATA statements in Applesoft. Listing 2.2 first shows a program for obtaining the same results in Listing 2.1, then shows a program for a more complex figure. Notice that the variable I is a counter for the number of line segments used, and that the coordinates and the endpoints of the lines are put sequentially in the DATA statements.

If you're into mathematics and want to play around a little with the coordinates, you can even read them into an array and perform some functions on them before plotting. For a few examples, see Listing 2.3.

Listing 2.1A

```
10  HGR
20  HCOLOR= 7
30  HPLOT 10,10 TO 250,10
40  HPLOT 250,10 TO 250,150
50  HPLOT 250,150 TO 10,150
60  HPLOT 10,150 TO 10,10
```

Listing 2.1B

```
10  HGR
20  HCOLOR= 7
30  HPLOT 10,10
40  HPLOT  TO 250,10
50  HPLOT  TO 250,150
60  HPLOT  TO 10,150
70  HPLOT  TO 10,10
```

Listing 2.1C

```
10  HGR
20  HCOLOR= 7
30  HPLOT 10,10 TO 250,10 TO 250,150
    TO 10,150 TO 10,10
```

A GOSUB was used for plotting the figure so that it wouldn't have to be repeated for each example. In lines 100 and 140, you may want to try some other mathematical functions, even things like SIN and COS. The only restrictions are that the results must be in a range 0 to 279 for X, and 0 to 191 for Y.

POKE text on and off

There are other handy-dandy commands you can use from Basic to affect what's happening on the graphics screen. The most common is POKE - 16302,0 (example: 20 POKE - 16302,0), which clears the text from the bottom of the screen after you use HGR. To get the text back, use

Listing 2.2A

```
10  HGR
20  HCOLOR= 7
29  REM  READ AND SET STARTING POINT
30  READ X,Y
40  HPLOT X,Y
50  FOR I = 1 TO 4
60  READ X,Y
70  HPLOT  TO X,Y
80  NEXT I
90  DATA  10,10,250,10,250,150,10,150,10,10
```

Listing 2.2B

```
10  HGR
20  HCOLOR= 7
30  READ X,Y
40  HPLOT X,Y
50  FOR I = 1 TO 37
60  READ X,Y
70  HPLOT  TO X,Y
80  NEXT I
90  DATA  12,5,11,4,8,4,6,6,6,7,7,8,4,
    11,3,13,3,16,4,18,6,20,8,21,7,21,5,22,7,
    23,9, 23,9,21,10,23,12,23,
    13,22,12,21,10,21,12,19,14,16,
    14,13,13,10,8,11,15,7,12,5,8,11,
    7,8,6,12,6,12,6,
    17,8,21,10,21,12,17,12,12,8,11
```

POKE - 16301,0. This is virtually irrelevant when you use HGR2, since the four lines of text at the bottom of the screen are associated with page 1 of hi-res graphics. To try it out, use POKE - 16302,0 as line 15 of any of the sample programs above.

Another command that you may find useful is CALL 62454 (example: 30 CALL 62454). This clears the screen to the most recently HPLOTed HCOLOR. Since the HGR commands clear the screen to black only, this is a way to choose a different background color. To try it, use the following in any of the examples:

```
18   HCOLOR = 6: HPLOT 0,0 : CALL 62454
```

Listing 2.3

```
10  HGR
20  HCOLOR= 7
30  DIM X(5),Y(5)
39  REM  READ THE ENDPOINTS
40  FOR I = 1 TO 5
50  READ X(I),Y(I)
60  NEXT
70  DATA  10,10,25,10,25,15,10,15,10,10
79  REM  DRAW IT NORMAL
80  GOSUB 500
89  REM  PLAY WITH THE ENDPOINTS AND DRAW IT
    AGAIN
90  FOR I = 1 TO 5
100 X(I) = X(I) + 30:Y(I) = Y(I) + 5
110  NEXT
120  GOSUB 500
129  REM  PLAY SOME MORE
130  FOR I = 1 TO 5
140 X(I) = X(I) / 2 + 50:Y(I) = Y(I) * 2
150  NEXT I
160  GOSUB 500
170  END
499  REM  THIS SUBROUTINE DRAWS THE FOUR LINES
     SPECIFIED BY THE CURRENT ENDPOINTS IN THE
     ARRAYS
500  HPLOT X(1),Y(1)
510  FOR I = 2 TO 5
520  HPLOT  TO X(I),Y(I)
530  NEXT I
540  RETURN
```

A couple of other pokes affect the hi-res screen, and may be useful at one time or another. They are listed here only for reference—if they don't sound useful to you now, just ignore them.

You can use pokes to switch between hi-res graphics and text without clearing either screen (the hi-res graphics screens and the text screen are independent and are always updated in memory, even though they may not be displayed at the time). POKE - 16303,0 switches from graphics to text mode, and POKE - 16304,0 switches from text to graphics mode. Neither clears the screen memory. Examples of using these are the graphic adventures that let you switch between viewing the text descriptions of a location and the hi-res picture of a location without erasing either screen.

You can also switch between the two pages of graphics without erasing: POKE - 16299,0 switches from page 1 to page 2, and POKE - 16300,0 switches from page 2 to page 1.

See Appendix B for a full summary of Applesoft and graphics commands. Other good reference material to keep on hand for the various graphics pokes and calls are the Apple II *Applesoft BASIC Programming Reference Manual* (or the Apple IIe Manual)[2] and the Apple II (or IIe) *Reference Manual*[3], all of which have sections on graphics. You'd have to be crazy to memorize all the various numbers to peek and poke, so it's nice to keep some reference within arm's reach. Also, see Appendix C in this book for machine language information on the Applesoft graphics commands that is not in the Apple manuals.

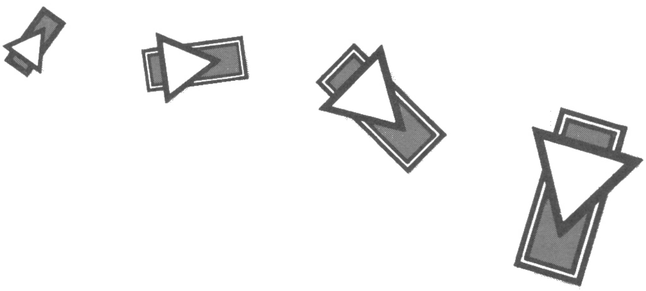
2. *Apple II Applesoft BASIC Programming Reference Manual* (Cupertino: Apple Computer, Inc., 1979, 1981)

Kamins, Scot, *Apple IIe Applesoft BASIC Programmer's Reference Manual* (Cupertino: Apple Computer, Inc., 1982)

3. Kamins, Scot, *Apple II Reference Manual* (Cupertino: Apple Computer, Inc., 1979, 1981)

Watson, Allen, *Apple IIe Reference Manual* (Cupertino: Apple Computer, Inc., 1982)

THREE



Shape Tables and Simple Animation

Hi-res graphics are not an inherent part of the Basic language. When the folks at Apple introduced hi-res graphics to the language, they added new commands that deal with vector shapes. A vector shape is one composed of lines or vectors. The vectors define the construction of the shape. A vector shape definition is something like "plot a line up one unit, plot a line left one unit, plot a line down one unit, plot a line right one unit," which gives a square. In Applesoft a group of vector shapes can be saved in something called a *shape table*.

Defining Shapes by Vectors

The advantage of defining shapes by vectors is the ability to scale and rotate the shapes easily. To scale, you just have to multiply the lengths of the lines by a number. To rotate, you change the directions by an offset. The disadvantage of this type of shape is that it is generally too slow for smooth, fast animation. The shapes are also more limited in color and detail than other types of shapes that we'll discuss later. (Before anyone jumps all over that one, yes, it's possible to create a very detailed and multicolored vector shape, but doing so eliminates the advantages of vector shapes: rotation and scaling. Rotating and scaling destroy any intricacies of color and detail.)

There is one other advantage to vector shapes, though. Because there are commands built into Applesoft for dealing with these shapes, they make a very good learning tool for beginning animation.

Creating a Shape

The first step in working with a shape table is to design a shape. If you look at the section in your Apple II *Applesoft BASIC Programming Reference Manual* (or the IIe Manual)[1] about creating shapes, you might be able to bumble through and define one. Most people, however, see that section, find binary numbers and arrows mixed with hexadecimal digits, and their eyes glaze over. After a few more chapters, yours may too. But for now let's pretend we're all beginners.

The program in Listing 3.1 is a short Basic program that lets you draw a shape and store it in a single-shape table. (Shape tables can store multitudes of shape definitions, all accessible by number, but to keep things simple we'll go with our table of one.) The programming details will be left to those who like to read the incomprehensible; the program basically accomplishes what the Applesoft manual tries to explain.

Type in the program, save it to disk (call it Shape Maker or something like that), then run it. The hi-res screen clears, and several controls are available to you. Your joystick or paddles control the scale and rotation of the shape you draw. (If you don't have either paddles or joystick, change line 30 so that S = 1 and R = 0, instead of the formulas given.) The I, J, K, and M keys are your direction keys. I is up, M is down, J is left, and K is right, just as positioned on the keyboard. Try using them; you should see your shape being drawn. If you keep the rotation set to zero, and the scale to one, you'll see the shape as it will be stored. Scaling and rotations can be used again later. Other controls are the Z and X keys, to turn the plotting on and off. Type F when you finish, name your shape, and it will be saved to disk. It's crude but effective; besides, it's definitely easier than hexadecimal and cheaper than buying a package just for drawing simple shapes.

Using Your Shape In a Program

There's a group of commands designed strictly for drawing shapes from a Basic program. The first is the command that loads the table into memory so the program can use it:

```
10   PRINT CHR$(4); "BLOAD name, A24576"
```

1. *Apple II Applesoft BASIC Programming Reference Manual* (Cupertino: Apple Computer, Inc., 1979, 1981)

Kamins, Scot, *Apple IIe Applesoft BASIC Programmer's Reference Manual* (Cupertino: Apple Computer, Inc., 1982)

Listing 3.1

```
10 L = 24576: POKE L,1: POKE L + 1,0: POKE L +
   2,4: POKE L + 3,0:L = L + 4: POKE 232,0:
   POKE 233,96
20 P = 4: POKE L,0: POKE L + 1,0:SW = 1: HGR :
   HOME : VTAB 21: PRINT "IJKMZXF"
30 S =  INT ( PDL (0) / 256 * 25) + 1: SCALE=
S:R =  INT ( PDL (1) / 4): ROT= R: VTAB 22:
HTAB 1: PRINT "ROT:";R;" SCALE:";S;"    "
40  XDRAW 1 AT 140,80
50  IF  PEEK ( - 16384) > 127 THEN 100
60  IF R <  >  INT ( PDL (1) / 4) THEN 90
70  IF S <  >  INT ( PDL (0) / 256 * 25) + 1
    THEN 90
80  GOTO 50
90  XDRAW 1 AT 140,80: GOTO 30
100  GET A$: IF A$ = "F" THEN 300
110  XDRAW 1 AT 140,80
120  IF A$ = "Z" THEN P = 4: GOTO 30
130  IF A$ = "X" THEN P = 0: GOTO 30
140  IF A$ = "I" THEN M = 0: GOTO 200
150  IF A$ = "M" THEN M = 2: GOTO 200
160  IF A$ = "J" THEN M = 3: GOTO 200
170  IF A$ = "K" THEN M = 1: GOTO 200
180  GOTO 30
200 V = M + P
210  IF SW = 1 THEN SW = 2:V1 = V: POKE L,V:
     POKE L + 1,0: GOTO 30
220  IF V + V1 = 0 THEN  POKE L,88:L = L + 1:
     POKE L,0:V1 = 0: GOTO 30
230  IF V = 0 THEN  POKE L,V1 + 192:L = L + 1:
     POKE L,0:V1 = 1: GOTO 30
240 V = V * 8 + V1: POKE L,V:L = L + 1
250 SW = 1: POKE L,0
260  GOTO 30
300  IF SW = 2 THEN  POKE L,V1:L = L + 1
310  POKE L,0
320  HOME : VTAB 21: INPUT "SHAPE NAME:";A$:
     ONERR  GOTO 320
330  PRINT  CHR$ (4);"BSAVE ";A$;",A24576,L";L
     - 24575
340  TEXT : PRINT "DONE"
```

This is a binary load of your shape table, with whatever name you used, from disk to memory (RAM), starting at RAM address 24576. After that, you have to poke in two pointers that will tell your program at which address you loaded the table. For location 24576, used above, the pokes are:

```
  20    POKE 232,0: POKE 233,96
```

If you want to know where those numbers came from, read the next two paragraphs. If you'd rather come back to it later instead of hitting confusing issues now, skip ahead.

The numbers correspond to the way addresses are usually stored in the computer. You may remember that in any single byte of memory you can store the numbers 0 to 255. To fit a larger address, you need two bytes. One byte holds the number of ones, the other holds the number of 256s. A simple example would be in base 10. Imagine you've got two slots that can only hold two digits each. The number 1587 could be broken up as 15 and 87, with 15 being the number of hundreds and 87 being the number of ones. You get the number for the first slot by dividing by 100 (lopping off the first two digits of a four-digit number) and the number for the second slot by taking the remainder. As long as you remember which is which—so you don't get the number 8715—you're okay.

With the Apple, instead of dealing with hundreds, you divide the number by 256s. The address we used to load our shape table at, 24576, divided by 256 is 96, with a remainder of zero. Therefore, 96 should be put in the high-order byte, and zero in the low-order byte. The only other item that may be confusing is that most of the time the addresses are stored in low/high format. Notice that with the two pokes above, we put zero (the low byte of the address or the remainder after dividing by 256) into location 232, and 96 into location 233. The first location gets the low byte, and the second location gets the high byte. Weird, maybe, but it's pretty consistent.

Two Paragraphs Later

The commands that affect the plotting of shapes are HCOLOR, ROT (rotation), SCALE, DRAW, and XDRAW. DRAW draws a shape at the coordinates you specify. Once a shape table is loaded and the address is poked into the necessary locations (232 and 233), you can use DRAW for any shape in the table. For example:

```
20 DRAW 1 AT 100,150
```

draws shape 1 in the table at the coordinates 100, 150.

HCOLOR sets the color for all subsequent HPLOTs and DRAWs, meaning that you can set the color for your shape to any of the six standard hi-res colors.

ROT controls the rotation that subsequent DRAWs will use. ROT = 0 is normal, ROT = 16 is 90 degrees rotation, ROT = 32 is 180 degrees, ROT = 48 is 270 degrees, and ROT = 64 is a full 360-degree rotation. Intermediate values give varying angles between those listed, depending on the scale used. The higher the scale, the more points of rotation you have available.

SCALE sets the scale of the subsequent draw commands. SCALE = 1 is normal, SCALE = 2 is double size, SCALE = 3 is triple size, and so on.

XDRAW looks like a DRAW command but it doesn't use a color. It reverses everything on the screen where the object is being drawn. If the background was white, the shape is drawn in black, and vice versa. XDRAW is nice because a second XDRAW at the same location erases the shape and restores the background to its original state.

Simple Animation

Moving one object at a time around the screen works all right with shape tables. The lack of speed really starts to show when you try to move more than one object. One, though, gives enough speed to start some simple animation.

Create a shape with the Shape Maker program, then type in the program in Listing 3.2. That program goes through the basics of animation. First you need to initialize all your information. From then on, it's a simple cycle: draw the shape on the screen, update the coordinates, erase the shape at the old coordinates, and repeat. Draw, update, erase, draw, update, erase, and so on.

Notice that lines 10-50 initialize everything. Line 10 lets you input your shape's name, and line 20 BLOADs it at the location you want. Line 30 sets the hi-res graphics screen, and pokes the necessary pointer locations with the address of the table. Line 40 sets the rotation and scale to normal, and line 50 sets the starting X and Y coordinates for our animation, and sets XC and YC (X change and Y change) to two each. Each time we go through the loop in this example, we'll use XC and YC to update the coordinates.

Line 60 begins the animation cycle by DRAWing the shape at location X,Y. Lines 70-120 save the old coordinates in XL,YL and then update them. Line 130 erases the shape by XDRAWing again at the old coordinates, XL,YL. Line 140 causes the sequence to be repeated.

Why draw-update-erase and have to save the old coordinates, instead of draw-erase-update? Because the update part of the cycle is the one that takes the most time, and during that time you want your shape on the screen. By erasing before the update, you'd have more time with your shape off the screen and a lot more flickering would be apparent.

Looking at the update cycle, notice that we use XC and YC to change the X and Y coordinates. That's just an arbitrary formula; try playing around with various ways of modifying X and Y. Notice, though, that in lines 90-120 we check the range of the new X and Y coordinates. If either is less than zero, or if X is greater than 279 or Y greater than 191, trying any DRAW, XDRAW, or HPLOT command will result in an error. Use lines like 90-120 whenever you're not positive that the result of a computation will be within those bounds.

Listing 3.2

```
9  REM  INITIALIZE
10  INPUT "SHAPE NAME :";A$: ONERR  GOTO 10
20  PRINT  CHR$ (4);"BLOAD  ";A$;",A24576"
30  HGR : POKE  - 16302,0: POKE 232,0: POKE
    233,96
40  ROT= 0: SCALE= 1
50 X = 100:Y = 80:XC = 2:YC = 2
59  REM  DRAW SHAPE
60  XDRAW 1 AT X,Y
69  REM  COMPUTE NEW COORDINATES
70 XL = X:YL = Y
80 X = X + XC:Y = Y + YC
90  IF X > 279 THEN X = 279:XC =  - 2
100  IF X < 0 THEN X = 0:XC = 2
110  IF Y > 191 THEN Y = 191:YC =  - 2
120  IF Y < 0 THEN Y = 0:YC = 2
129  REM  ERASE SHAPE
130  XDRAW 1 AT XL,YL
139  REM  REPEAT
140  GOTO 60
```

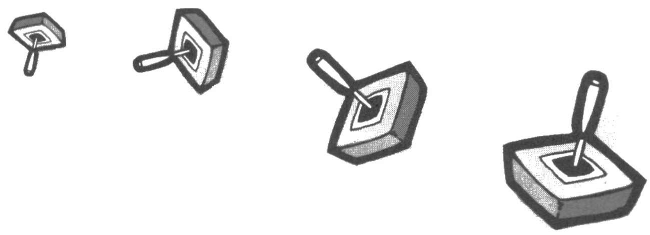
The last example is a simple animation program similar to that in Listing 3.2, spiffed up just a little. Instead of using XC and YC as the X and Y offset, we'll use the joystick (or paddle) values to determine which way to move. We'll also add a couple of optional lines that play with the rotation and scale of the shape. Note that Listing 3.3 varies at lines 50 and 80 from Listing 3.2, and that lines 90-120 are shortened. Lines 132 and 134 are optional and can be added or deleted at any time to demonstrate their effect.

As always, try whatever variations you want. You won't break the computer trying.

Listing 3.3

```
9  REM  INITIALIZE
10  TEXT : HOME : INPUT "SHAPE NAME :";A$:
    ONERR  GOTO 10
20  PRINT  CHR$ (4);"BLOAD ";A$;",A24576"
30  HGR : POKE  - 16302,0: POKE 232,0: POKE
    233,96
40  ROT= 0: SCALE= 1
50 X = 100:Y = 80:R = 0
59  REM  DRAW SHAPE
60  XDRAW 1 AT X,Y
69  REM  COMPUTE NEW COORDINATES
70 XL = X:YL = Y
80 X = X +  INT (( PDL (0) - 128) / 26):Y = Y
   +  INT (( PDL (1) - 128) / 26)
90  IF X > 279 THEN X = 279
100  IF X < 0 THEN X = 0
110  IF Y > 191 THEN Y = 191
120  IF Y < 0 THEN Y = 0
129  REM  ERASE SHAPE
130  XDRAW 1 AT XL,YL
132 R = R + 8: ROT= R: IF R > 64 THEN R = 0
134  SCALE= 6 - ( ABS (X - 140) +  ABS (Y -
     96)) / 50
139  REM  REPEAT
140  GOTO 60
```

F O U R



Controlling Animation

Now we have a program that will create Applesoft shapes and have some clues to how animation works. By plotting a shape, updating the coordinates, erasing it, and then replotting, all in a loop, we can create the illusion (if not the actuality) of movement. Now, can that illusion be created using a joystick, or keyboard, or some predetermined function or path? Probably so.

Listing 4.1 shows the beginning of a program that will load and initialize a shape made with last chapter's Shape Maker. Alternately, you can use Listing 4.2, which has the beginning of a program that pokes in a shape in the form of a little tiny circle and does the same initialization. Either one of these two listings must be used as the beginning of each of the other listings in this chapter.

Controlling the Shape

Listing 4.3 shows how a shape can be controlled with a joystick. You may notice some similarities to Listing 3.3. Notice that line 50 simply

Listing 4.1

```
1  REM  EXHIBIT A
9  REM  INITIALIZE
10  INPUT "SHAPE NAME:";A$: ONERR  GOTO 10
20  PRINT  CHR$ (4);"BLOAD ";A$;",A24576"
30  HGR : POKE  - 16302,0: POKE 232,0: POKE
    233,96
40  ROT= 0: SCALE= 1
```

sets the starting point for the animation. The animation loop runs from lines 60 through 100. Line 60 does the draw, line 70 saves the old coordinates, line 80 does the update, and line 90 erases. The key is line 80, where the update occurs.

The variable S in lines 55 and 80 controls the sensitivity of the joystick, and its value can range from 1 to 127. The lower the value, the less sensitive the joystick will be to your movements. The function in line 80 takes the joystick values, which range from 0 to 255, and maps them to a range of -1 to 1 (when the sensitivity is increased, the range of values will eventually increase also). This way, based on the setting of the joystick, X and Y will either increase or decrease by 1.

Also in line 80 is a GOSUB (go to subroutine) to line 200, which is the beginning of a sequence of IF-THEN statements that test the values of X and Y. If either is out of the proper range (0-279 for X; 0-191 for Y), the subroutine assigns the value for the edge of the screen. Each of the programs in this chapter contains a similar subroutine that prevents an out of range error.

Can You Draw With It?

Listing 4.4 contains essentially the same program, with a few lines added that allow your animation to leave a trail (answering the trick question, "Can you draw with it?"). Lines 45, 85, 86, and 95 have been added and allow you to turn HPLOT on and off with the two button inputs. Line 95 does the HPLOT (when P = 1, the routine does the plot; when

Listing 4.2

```
1  REM  EXHIBIT A
9  REM  INITIALIZE
10  POKE 24576,1: POKE 24577,0: POKE 24578,4:
    POKE 24579,0: POKE 24580,18: POKE
    24581,63:
    POKE 24582,32: POKE 24583,100: POKE
    24584,45:
    POKE 24585,21
20  POKE 24586,54: POKE 24587,30: POKE
    24588,7: POKE 24589,0
30  HGR : POKE  - 16302,0: POKE 232,0: POKE
    233,96
40  ROT= 0: SCALE= 1
```

P = 0, it skips). Lines 85 and 86 read the buttons, and if either is pushed, change the value of P accordingly. You may want to try changing the value of S (sensitivity, remember?) in line 55 to a value of 100 or so, just to see the difference in the handling of the joystick.

But suppose you don't have a joystick. Never fear, much the same can be done through the keyboard! Enter Listing 4.5. The changes from the joystick example have all been made between lines 70 and 90. Instead of updating the position using joystick reads, line 72 checks to see if a key has been pressed. If one hasn't, the program skips down to line 84 and uses whatever increments were in effect before (XI and YI). If a key has been pressed, however, line 74 gets the value of the key, and a

Listing 4.3

```
1  REM  EXHIBIT A
9  REM  INITIALIZE
10  POKE 24576,1: POKE 24577,0: POKE 24578,4:
    POKE 24579,0: POKE 24580,18: POKE
    24581,63:
    POKE 24582,32: POKE 24583,100: POKE
    24584,45:
    POKE 24585,21
20  POKE 24586,54: POKE 24587,30: POKE
    24588,7: POKE 24589,0
30  HGR : POKE  - 16302,0: POKE 232,0: POKE
    233,96
40  ROT= 0: SCALE= 1
50 X = 140:Y = 96
55 S = 50:D = 255 - 2 * S
59  REM  BEGINNING OF ANIMATION LOOP
60  XDRAW 1 AT X,Y
70 XL = X:YL = Y
80 X = X +  INT (( PDL (0) - S) / D):Y = Y +
   INT (( PDL (1) - S) / D): GOSUB 200
90  XDRAW 1 AT XL,YL
100  GOTO 60
199  REM  SUBROUTINE TO CHECK RANGE OF X AND Y
200  IF X < 0 THEN X = 0
210  IF X > 279 THEN X = 279
220  IF Y < 0 THEN Y = 0
230  IF Y > 191 THEN Y = 191
240  RETURN
```

sequence of IF statements determines what is to be done based on the key that's been pressed. If the letter I has been pressed, for example, the X increment is set to zero and the Y increment to negative one, causing the next movement to be upward. M causes a downward move, and J and K move left and right respectively. A space sets both increments to zero, which stops movement, and, to keep your imagination going, W moves several units at a time down and right. Also, the Z and

---

Listing 4.4

```
1  REM  EXHIBIT A
9  REM  INITIALIZE
10  POKE 24576,1: POKE 24577,0: POKE 24578,4:
    POKE 24579,0: POKE 24580,18: POKE
    24581,63:
    POKE 24582,32: POKE 24583,100: POKE
    24584,45:
    POKE 24585,21
20  POKE 24586,54: POKE 24587,30: POKE
    24588,7: POKE 24589,0
30  HGR : POKE  - 16302,0: POKE 232,0: POKE
    233,96
40  ROT= 0: SCALE= 1
45  HCOLOR= 7
50 X = 140:Y = 96
55 S = 50:D = 255 - 2 * S
59  REM  BEGINNING OF ANIMATION LOOP
60  XDRAW 1 AT X,Y
70 XL = X:YL = Y
80 X = X +  INT (( PDL (0) - S) / D):Y = Y +
   INT (( PDL (1) - S) / D): GOSUB 200
85  IF  PEEK ( - 16287) > 127 THEN P = 1
86  IF  PEEK ( - 16286) > 127 THEN P = 0
90  XDRAW 1 AT XL,YL
95  IF P THEN  HPLOT XL,YL
100  GOTO 60
199  REM  SUBROUTINE TO CHECK RANGE OF X AND Y
200  IF X < 0 THEN X = 0
210  IF X > 279 THEN X = 279
220  IF Y < 0 THEN Y = 0
230  IF Y > 191 THEN Y = 191
240  RETURN
```

X keys are set to turn plotting on and off, using the variable P as in the last example.

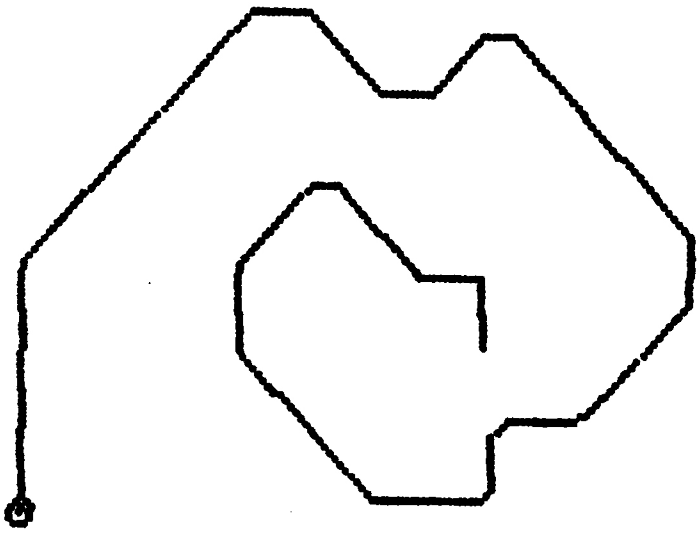
More Control

Suppose you want to get the computer to control your animated shape in some fashion. You have several ways to do this. One is by using a formula to determine the new coordinates. Another is by predefining a path for the shape. A third is to let the computer generate a random path.

Using a formula

The first method, using a formula, is probably the most difficult because it requires some use of (ugh) mathematics. Listing 4.6 shows how it's used, but this is not the place to get into heavy discussion of the sociological values of sines, cosines, and absolute values. This particular part will be short and sweet.

A new subroutine has been added at line 150. It evaluates a function—being very creative, we tried using SIN (X)—and returns an X,Y coordinate scaled to fit the screen. How? Okay, first, the initial values



Example 4.4 Joystick Draw

Listing 4.5

```
1  REM  EXHIBIT A
9  REM  INITIALIZE
10  POKE 24576,1: POKE 24577,0: POKE 24578,4:
    POKE 24579,0: POKE 24580,18: POKE
    24581,63:
    POKE 24582,32: POKE 24583,100: POKE
    24584,45:
    POKE 24585,21
20  POKE 24586,54: POKE 24587,30: POKE
    24588,7: POKE 24589,0
30  HGR : POKE  - 16302,0: POKE 232,0: POKE
    233,96
40  ROT= 0: SCALE= 1
45  HCOLOR= 7
50 X = 140:Y = 96
59  REM  BEGINNING OF ANIMATION LOOP
60  XDRAW 1 AT X,Y
70 XL = X:YL = Y
72  IF  PEEK ( - 16384) < 128 THEN 84
74  GET A$: IF A$ = "I" THEN XI = 0:YI =  - 1
75  IF A$ = "M" THEN XI = 0:YI = 1
76  IF A$ = "J" THEN XI =  - 1:YI = 0
77  IF A$ = "K" THEN XI = 1:YI = 0
78  IF A$ = " " THEN XI = 0:YI = 0
79  IF A$ = "W" THEN XI = 5:YI = 3
80  IF A$ = "Z" THEN P = 1
81  IF A$ = "X" THEN P = 0
84 X = X + XI:Y = Y + YI: GOSUB 200
90  XDRAW 1 AT XL,YL
95  IF P THEN  HPLOT XL,YL
100  GOTO 60
199  REM  SUBROUTINE TO CHECK RANGE OF X AND Y
200  IF X < 0 THEN X = 0
210  IF X > 279 THEN X = 279
220  IF Y < 0 THEN Y = 0
230  IF Y > 191 THEN Y = 191
240  RETURN
```

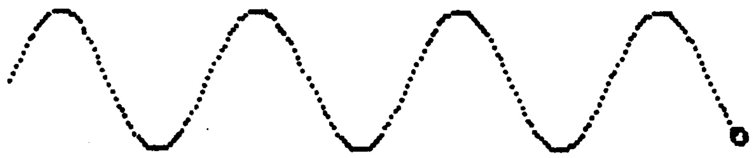
Listing 4.6

```
1  REM  EXHIBIT A
9  REM  INITIALIZE
10  POKE 24576,1: POKE 24577,0: POKE 24578,4:
    POKE 24579,0: POKE 24580,18: POKE
    24581,63:
    POKE 24582,32: POKE 24583,100: POKE
    24584,45:
    POKE 24585,21
20  POKE 24586,54: POKE 24587,30: POKE
    24588,7: POKE 24589,0
30  HGR : POKE  - 16302,0: POKE 232,0: POKE
    233,96
40  ROT= 0: SCALE= 1
45  HCOLOR= 7
50 XC = 0: GOSUB 150
55 P = 1
59  REM  BEGINNING OF ANIMATION LOOP
60  FOR XC = .1 TO 28 STEP .1
65  XDRAW 1 AT X,Y
70 XL = X:YL = Y
80  GOSUB 150
90  XDRAW 1 AT XL,YL
95  IF P THEN  HPLOT XL,YL
100  NEXT XC
110  END
150 YC =  SIN (XC)
160 X = 10 * XC
170 Y = 191 - (YC * 20 + 95)
180  GOSUB 200: RETURN
199  REM  SUBROUTINE TO CHECK RANGE OF X AND Y
200  IF X < 0 THEN X = 0
210  IF X > 279 THEN X = X -  INT (X / 279) *
     279
220  IF Y < 0 THEN Y = 0
230  IF Y > 191 THEN Y = 191
240  RETURN
```

of X and Y are computed and plotted at line 50, using the subroutine, of course. XC and YC will be the actual coordinates used in the functions. X and Y will be the values that are fitted to the screen. The animation loop has a FOR-NEXT counter that changes XC from .1 to 28 in increments of .1. Those were chosen because SIN was used for a function (it cycles through four circles, using radian measure), and it was easy to adjust to the screen (280 dots—the exact screen width). You can actually use whatever values you want. Within the loop, the updating is done by calling the subroutine at line 150.

In the subroutine, YC is first calculated as a function of XC (YC = SIN XC). The next two lines scale X and Y appropriately. Since XC goes from 0 to 28 and we want X to go from 0 to 279 (the screen coordinates), we can multiply whatever XC is by 10 so it fits the screen exactly. Y is a little more tricky; see line 170. Since SIN gives results from -1 to 1, we can multiply that by something to get more than a two dot vertical movement. If we were graphing the exact function, 10 would be appropriate, since that's what we multiplied X by. To make it a little more dramatic, though, we used 20. That makes the low end -1*20 or -20, and the high end 1*20 or 20.

Since the actual screen values for Y are all positive (from 0 to 191), we then added 95 to center the results vertically on the screen (-20+95=75, 20+95=115). The last step, if we are worrying about an



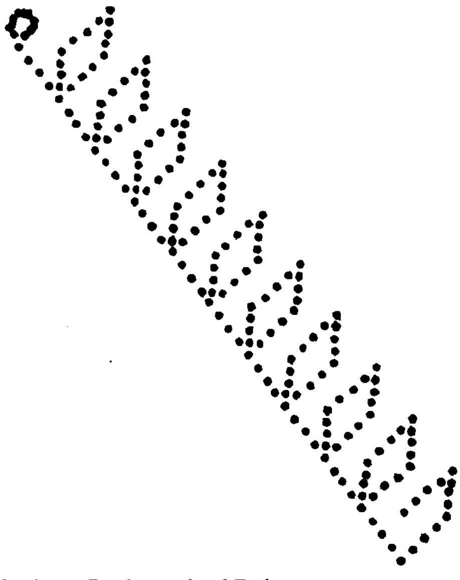
Example 4.5 Plotting a Formula

accurate result, is to subtract all of the above from 191. That's because on the hi-res screen the Y values all appear reversed from the way your experience with normal graphing would lead you to expect. Of course, if we are just using a function for effect and not for accurate plotting, this last step can be left out. Note in the program that line 55 sets P so that the trail will be plotted.

Enough Math!

If none of that made sense, or if it wasn't very interesting, here's another approach that has some more immediate results. See Listing 4.7.

Similar in a way to what we did with keystrokes, this program sets a path in advance that determines how the shape will move. First line 55 optionally sets P so that the trail will be plotted. It then sets N, the number of moves in the path, to 20; but you can choose whatever you want. Lines 60-100 compose the animation loop again. This time a FOR-NEXT loop counts from 1 to N, and each time through the loop the next move, M, is read from a DATA statement (line 150). Line 74 is called a computed GOTO statement. Program control goes to the line number in the list that corresponds to the value of M. If M = 3, for example, it will use the third line number, 77. Each of the eight lines sets the X and



Example 4.7 Plotting a Predetermined Path

Y increments so that moves corresponding to 1 through 8 give the directions shown in Figure 4.1.

Note that the DATA statement in line 150 contains numbers that, when read by the program, will correspond to these directions. After the animation loop completes N moves (finishing the list of numbers in the DATA statement), the RESTORE in line 110 starts the data at the beginning of the list again and causes the path to be repeated.

Note also that in the subroutine that begins at line 200, the values have been changed slightly. Now if a shape gets to the edge of the screen, instead of stopping it there, the program puts it at the opposite edge so it can continue. Something similar was done in line 210 of Listing 4.6, where we put a formula that, instead of using X, uses the remainder after dividing by 279. In Listing 4.6 try changing the number 28 in line 60 to something like 56.

Listing 4.7

```
1  REM  EXHIBIT A
9  REM  INITIALIZE
10  POKE 24576,1: POKE 24577,0: POKE 24578,4:
    POKE 24579,0: POKE 24580,18: POKE
    24581,63:
    POKE 24582,32: POKE 24583,100: POKE
    24584,45:
    POKE 24585,21
20  POKE 24586,54: POKE 24587,30: POKE
    24588,7: POKE 24589,0
30  HGR : POKE  - 16302,0: POKE 232,0: POKE
    233,96
40  ROT= 0: SCALE= 1
45  HCOLOR= 7
50 X = 140:Y = 96
55 P = 1:N = 20
59  REM  BEGINNING OF ANIMATION LOOP
60  FOR I = 1 TO N
65  XDRAW 1 AT X,Y
70 XL = X:YL = Y
72  READ M
74  ON M GOTO 75,76,77,78,79,80,81,82
75 XI =  - 2:YI =  - 2: GOTO 84
76 XI = 0:YI =  - 2: GOTO 84
77 XI = 2:YI =  - 2: GOTO 84
```

Listing 4.7 (continued)

```
78 XI = 2:YI = 0: GOTO 84
79 XI = 2:YI = 2: GOTO 84
80 XI = 0:YI = 2: GOTO 84
81 XI =  - 2:YI = 2: GOTO 84
82 XI =  - 2:YI = 0
84 X = X + XI:Y = Y + YI: GOSUB 200
90  XDRAW 1 AT XL,YL
95  IF P THEN  HPLOT XL,YL
100  NEXT I
110  RESTORE : GOTO 60
150  DATA  3,3,3,3,6,6,6,6,7,7,7,7,
     1,1,1,1,2,2,2,2
199  REM  SUBROUTINE TO CHECK RANGE OF X AND Y
200  IF X < 0 THEN X = 279
210  IF X > 279 THEN X = 0
220  IF Y < 0 THEN Y = 191
230  IF Y > 191 THEN Y = 0
240  RETURN
```

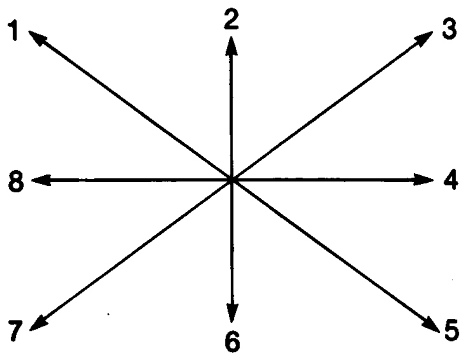


Figure 4.1 Directions of the Shape Paths

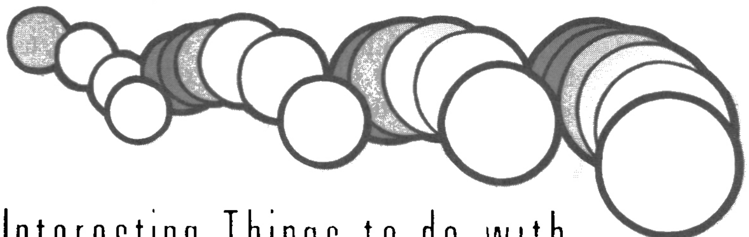
Listing 4.8

```
1  REM  EXHIBIT A
9  REM  INITIALIZE
10  POKE 24576,1: POKE 24577,0: POKE 24578,4:
    POKE 24579,0: POKE 24580,18: POKE
    24581,63:
    POKE 24582,32: POKE 24583,100: POKE
    24584,45:
    POKE 24585,21
20  POKE 24586,54: POKE 24587,30: POKE
    24588,7: POKE 24589,0
30  HGR : POKE  - 16302,0: POKE 232,0: POKE 23
    POKE 232,0: POKE 233,96
40  ROT = 0: SCALE = 1
    45  HCOLOR = 7
    50  X = 140:Y = 96
    55  P .1
    60  XDRAW 1 AT X,Y
70  XL .X:YL .Y72 M = X:YL = Y
72 M =  INT ( RND (1) * 8) + 1
74  ON M GOTO 75,76,77,78,79,80,81,82
75 XI =  - 2:YI =  - 2: GOTO 84
76 XI = 0:YI =  - 2: GOTO 84
77 XI = 2:YI =  - 2: GOTO 84
78 XI = 2:YI = 0: GOTO 84
79 XI = 2:YI = 2: GOTO 84
80 XI = 0:YI = 2: GOTO 84
81 XI =  - 2:YI = 2: GOTO 84
82 XI =  - 2:YI = 0
84 X = X + XI:Y = Y + YI: GOSUB 200
90  XDRAW 1 AT XL,YL
95  IF P THEN  HPLOT XL,YL
100  GOTO 60
199  REM  SUBROUTINE TO CHECK RANGE OF X AND Y
200  IF X < 0 THEN X = 279
210  IF X > 279 THEN X = 0
220  IF Y < 0 THEN Y = 191
230  IF Y > 191 THEN Y = 0
240  RETURN
```

Draw at random

In Listing 4.8 we let the computer do whatever it wants. It's very similar to Listing 4.7, except instead of putting the moves in DATA statements, we use the computer's random number generator to pick random moves. For that we don't need N, the FOR-NEXT loop, or the READ, DATA, and RESTORE statements. Just replace the READ with an instruction that chooses a random number from 1 to 8. Call it random computer scribbling, if you wish.

FIVE



Interesting Things to do with Shape Tables and Simple Animation

Although most graphics are done in machine language for speed, there are some interesting, off-the-wall effects that can be created easily and quickly using shape tables and Applesoft graphics commands.

Explosions

Using Applesoft shape tables, you can do fast, decent-looking explosions. First, use the Shape Maker from Chapter 3 to create a somewhat random shape like Figure 5.1. Make it small, as we will be scaling it larger. Also, be sure that your starting point is somewhere in the middle of the shape so that it stays centered when we increase the scale. Name this file SHAPE (original, huh?).

Now use the routine in Listing 5.1 to create an explosion at location X,Y. The routine first draws the shape you created in scale 1, erases it,

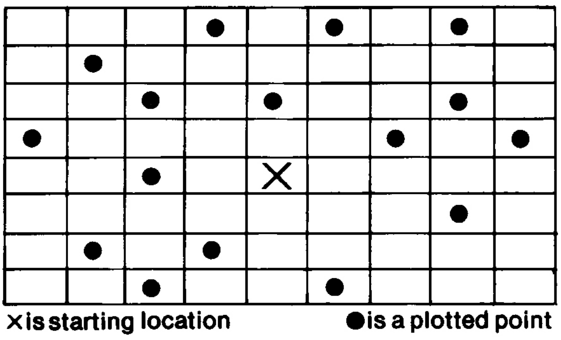


Figure 5.1 Random Explosion Shape

draws it in scale 2, erases it, and so on until it reaches scale 7. Along the way, it varies the HCOLOR from one to seven, adding color to the effect. If you use this routine on a background you want to keep, use XDRAWs instead of DRAWs, and skip the HCOLOR commands—DRAW will clobber whatever was there. Variations of this can be done by waiting until all seven scales are drawn, then erasing, or by erasing in reverse order.

Lasers

The HPLOT command can be used to create a quick laser effect between any two points. Usually you have a set source for the beam. The two bottom corners are favorites in several games because they give the impression that the user is in a spaceship (or something like that). That is what we use in Listing 5.2. Unfortunately, there is no HPLOT command in lines with the features of XDRAW for shapes, so any fancy background will be hurt by this effect.

Listing 5.1

```
10  PRINT  CHR$ (4);"BLOAD SHAPE,A16384"
20  POKE 232,0: POKE 233,64
30  HGR : POKE  - 16302,0
40  REM  X,Y IS THE LOCATION OF THE EXPLOSION
50 X = 140:Y = 96
320  FOR I = 1 TO 7
330  SCALE= I: HCOLOR= I
340  DRAW 1 AT X,Y
350  HCOLOR= 0
360  DRAW 1 AT X,Y
370  NEXT I
```

Listing 5.2

```
10  HGR : POKE  - 16302,0
20 X = 140:Y = 96: REM  TARGET LOCATION
300  HCOLOR= 5: HPLOT 278,190 TO X,Y: HPLOT
     1,190 TO X,Y: REM  DRAW
310  HCOLOR= 0: HPLOT 278,190 TO X,Y: HPLOT
     1,190 TO X,Y: REM  ERASE
```

Now, of course, these two effects are just dying to be used together, which is why we used the odd line numbers. Listing 5.3 is a routine that incorporates both into a laser blast followed by an explosion sequence.

Bouncing Ball

Last chapter we provided some samples of animation using random paths, present paths, and joystick paddles and keyboard control. As an extension of those, Listing 5.4 is a program by friendly Softalk editor, David Durkee, that simulates the natural movement of a common object: a bouncing ball. You'll need a new shape for this program, as illustrated in Figure 5.2. Name your new shape file BALL.

Listing 5.3

```
10  PRINT  CHR$ (4);"BLOAD SHAPE,A16384"
20  POKE 232,0: POKE 233,64
30  HGR : POKE  - 16302,0
40 X = 140:Y = 96: REM  WHERE IT ALL HAPPENS
300  HCOLOR= 5: HPLOT 278,190 TO X,Y: HPLOT
     1,190 TO X,Y
310  HCOLOR= 0: HPLOT 278,190 TO X,Y: HPLOT
     1,190 TO X,Y
320  FOR I = 1 TO 7
330  SCALE= I: HCOLOR= I
340  DRAW 1 AT X,Y
350  HCOLOR= 0
360  DRAW 1 AT X,Y
370  NEXT I
```

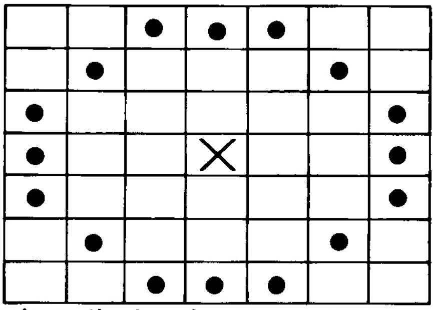


Figure 5.2 Bouncing Ball Shape

Listing 5.4

```
10  PRINT  CHR$ (4);"BLOAD BALL,A16384"
20  POKE 232,0: POKE 233,64
30  HGR : POKE  - 16302,0
40  SCALE= 1: ROT= 0: HCOLOR= 3
50  HPLOT 0,0 TO 279,0 TO 279,191 TO 0,191 TO
    0,0
60 X = 30:Y = 30:XM = 5:YM =  - 1
70 XF = 12:YF = 20:GF = .4
100  REM  MAIN LOOP
110  XDRAW 1 AT X,Y:XO = X:YO = Y
120 X = X + XM:Y = Y + YM
130  IF X < 3 THEN X = 3: GOTO 200
140  IF X > 276 THEN X = 276: GOTO 200
150  IF Y < 3 THEN Y = 3: GOTO 250
160  IF Y > 188 THEN Y = 188: GOTO 250
170 YM = YM + GF
180  XDRAW 1 AT XO,YO
190  GOTO 110
200  REM  LEFT OR RIGHT
210 XM =  - XM
220 XM =  SGN (XM) * ( ABS (XM) -  ABS (XM) /
    XF)
230  GOTO 150
250  REM  TOP OR BOTTOM
260 YM =  - YM
270 YM =  SGN (YM) * ( ABS (YM) -  ABS (YM) /
    YF)
280  GOTO 170
300  HCOLOR= 5: HPLOT 278,190 TO X,Y: HPLOT
     1,190 TO X,Y
310  HCOLOR= 0: HPLOT 278,190 TO X,Y: HPLOT
     1,190 TO X,Y
320  FOR I = 1 TO 7
330  SCALE= I: HCOLOR= I
340  DRAW 1 AT X,Y
350  HCOLOR= 0
360  DRAW 1 AT X,Y
370  NEXT I
380  GOTO 300
```

Lines 10 through 70 set things up. The main loop, lines 100 through 190, follows this sequence:

Line 110 draws the shape at X,Y.

Line 120 updates the shape's location by adding the momentum factors, XM and YM. These are preset in line 60 and updated in various places throughout to account for gravity and bouncing.

Lines 130 through 160 check to see if the ball has hit a wall. If it has, the routines from lines 200 through 280 reverse the direction of movement and simulate the loss of energy that occurs when a ball bounces.

Line 170 is the gravity factor.

Line 180 erases the shape.

Line 190 goes to the beginning of the loop.

The effects of entropy

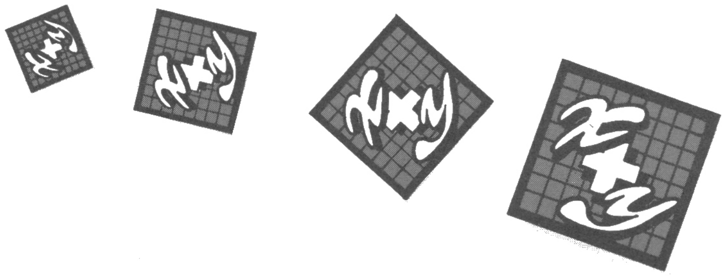
Take a look at line 220. This is where the object loses some of its momentum in bouncing. ABS (XM) / XF is the fraction of the energy that is lost. If you make XF smaller, the object will lose more energy with each collison. However, if you make it a negative number, it will gain energy!

The preset values for this variable and others are located in lines 60 and 70. Try changing them and seeing how each affects the physical laws in the little world on the hi-res screen. The remarks in line 1 through 4 explain what each controls.

Now, the only trouble with this is that it goes on forever unless you use Control-C to get out of it. This is the reason that we numbered the important parts of the laser and explosion routine starting at 300. If you just add those lines 300 through 370 into the Listing 5.4 program, and then add the following two lines, you get a neat way out by stopping the program when you press any key. It demonstrates aptly how to use such a subroutine in a larger program.

```
185   IF = PEEK( - 16384) < 127 THEN POKE -
16368,0 : GO TO 300
380   FOR DL = 1 TO 500: NEXT DL: HOME : TEXT
: END
```

S I X



Lookup Tables and Bit-Mapped Graphics

To create really fancy graphics as a programmer you have to acquaint yourself with the actual mapping of the Apple hi-res screen and with handling graphics on a byte and bit level. It's not very difficult once you learn a few tricks of the trade. The first is coping with the strange memory map of the screen.

Figure 6.1 shows a small part of the upper left corner of hi-res page 1. The actual screen is 192 bytes tall and 40 bytes wide. The bytes are displayed horizontally, with 7 bits (on/offs) in each displayed, so the 40 bytes width gives 280 dots. The actual address (location in memory) of any byte on the screen can be computed by taking the Y address down the left side of Figure 6.1 and adding the X value, from 0 to 39. The X offset is nice and simple, with 0 on the left, 39 on the right, and all the numbers running normally in between. The locations of the start of each line, given by the Y value, are quite a different story. At one point there must have been a good reason for the strange sequencing of addresses, but it remains an unusual puzzle for most beginning graphics programmers. There is a formula for computing the starting address of each line,

Listing 6.1A

```
5  HOME
10  INPUT "Y VALUE : ";Y
20 Y1 =  INT (Y / 8):YR = Y - Y1 * 8
30 Y2 =  INT (Y1 / 8):YS = Y1 - Y2 * 8
40 L = 8192 + Y2 * 40 + YS * 128 + YR * 1024
50  PRINT "STARTING ADDRESS IS ";L
60  GOTO 10
```

given the Y value of that line. Listing 6.1 is a short program that contains that computation. It allows you to input a number for Y (0-191), and prints the address of the start of that line. There's no need to go into the actual formula in this context; we'll just say it's there. (Note: to use hi-res page 2, instead of page 1, change the 8192 in line 40 to 16384.)

Listing 6.2 contains a little more excitement. Using the same computation for the Y location, it allows you to enter values for X and Y.

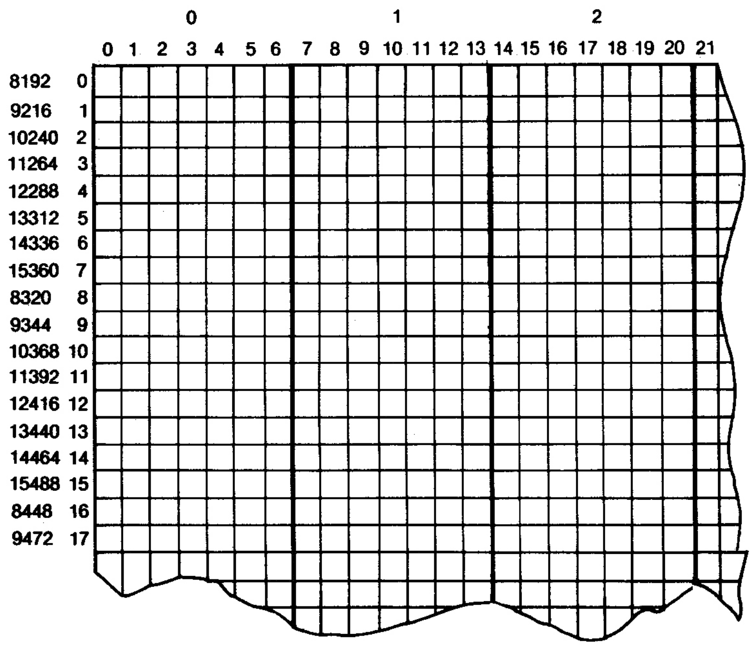


Figure 6.1 Cutaway of Upper Left Corner of Hi-Res Screen

Listing 6.2

```
10  HGR : VTAB 23
15  INPUT "X : ";X: IF X < 0 OR X > 39 THEN 15
16  INPUT "Y : ";Y: IF Y < 0 OR Y > 191 THEN
    16
20 Y1 =  INT (Y / 8):YR = Y - Y1 * 8
30 Y2 =  INT (Y1 / 8):YS = Y1 - Y2 * 8
40 YL = 8192 + Y2 * 40 + YS * 128 + YR * 1024
50 L = YL + X
60  POKE L,255
70  GOTO 15
```

It computes the Y location (lines 20-40) then adds the X value (line 50) to give a specific screen address. Then it pokes into that address the number 255, which sets all seven points in that byte (the number 127 would have the same result).

Tricks of the Trade #1

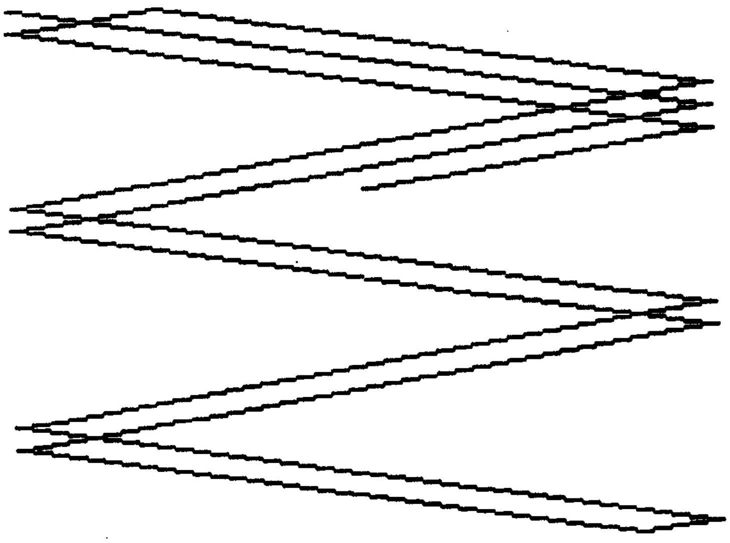
Doing a three line computation each time you want to find a screen location takes a lot of time. Even in machine language, the time used to compute a Y address can make a difference when speed is an important factor (it usually is). To avoid using computations all the time, a trick that is often used is pre-computing all the Y locations and storing them in a table. Then, when an address is needed, instead of computing, you just look for it in the table. This is usually done in machine language programs, but for explanation we'll do it in Basic. Next chapter, we'll convert.

Listing 6.3 has a program similar to that in Listing 6.2, except it repeats in a pattern around the screen. It uses a trick we've used before, with initial values of X and Y set in line 15, along with increment values. In lines 70 and 80, X and Y are updated, and if they reach the edge of the screen the increments are reversed, giving the illusion of bouncing. Try it, and take notice of the speed. The Y value in this example is computed before each plot.

Listing 6.4 has the same program with a Y-lookup table for the addresses. The subroutine starting at line 150 creates the table, computing each location for Y, from 0 to 191, and storing them in an array,

Listing 6.3

```
10  HGR
15 X = 0:Y = 0:XC = 1:YC = 1
20 Y1 =  INT (Y / 8):YR = Y - Y1 * 8
30 Y2 =  INT (Y1 / 8):YS = Y1 - Y2 * 8
40 YL = 8192 + Y2 * 40 + YS * 128 + YR * 1024
50 L = YL + X
60  POKE L,255
70 X = X + XC: IF X < 1 OR X > 38 THEN XC =
    - XC
80 Y = Y + YC: IF Y < 1 OR Y > 190 THEN YC =
    - YC
90  GOTO 20
```



Example 6.3 Plotting Using a Y-Lookup Table

Listing 6.4

```
10  HGR
12  GOSUB 150
15 X = 0:Y = 0:XC = 1:YC = 1
20 L = YT(Y) + X
60  POKE L,255
70 X = X + XC: IF X < 1 OR X > 38 THEN XC =
   - XC
80 Y = Y + YC: IF Y < 1 OR Y > 190 THEN YC =
   - YC
90  GOTO 20
140  REM THIS SUBROUTINE CREATES A Y-LOOKUP
     TABLE, YT.
150  DIM YT(191)
160  FOR Y = 0 TO 191
200 Y1 =  INT (Y / 8):YR = Y - Y1 * 8
210 Y2 =  INT (Y1 / 8):YS = Y1 - Y2 * 8
220 YL = 8192 + Y2 * 40 + YS * 128 + YR * 1024
230 YT(Y) = YL
240  NEXT Y
250  RETURN
```

YT. The first thing that is done in the program is the subroutine call (line 12), which means that for the rest of the program we'll have these pre-computed addresses sitting in the array YT. The rest of the program is exactly as before, except the computation from lines 20 to 50 has been reduced to one short line 20, which takes the computed Y address in the array YT for whichever Y value you need, then adds the X offset. Note that it takes several seconds in Basic to create the lookup table when you run the program, but after the table is computed the graphics move much faster.

A Basic Character Generator

Once you have an idea of what bit/byte graphics are (poking values into screen memory), you are ready to start using them to put things on the hi-res page. What we've used before, shape tables, are known as *vector graphics*. Objects are defined by taking a starting point and moving in lines and plotting from point to point. The move commands can be translated anywhere on the screen, so your shape can be plotted anywhere. The other type of graphics is called *bit-mapped graphics,* where you define a set of dots that are to be on or off, store them as a sequence of bytes, then put those bytes wherever you want them in the screen memory area. That's what the programs in Listings 6.2-6.4 do, with the bit map being the number 255 that we were poking into screen memory.

Listing 6.5 is similar again. It uses the lookup table to find the Y addresses. But instead of a single POKE command, we've put in lines 20 to 70. Lines 20 to 60 form a loop that repeats eight times, incrementing the Y value and poking a number into the screen area each time. The result is that instead of one byte, it puts eight bytes of information on the screen or a block of dots seven wide and eight tall. It reads the values for the eight bytes from the DATA statement in line 70. As it turns out, those numbers result in the letter H being put on the screen.

To see how those numbers were discovered, see Figure 6.2. Each row of that grid represents one byte: eight rows, eight bytes. Each column represents bits within the bytes. Bits 0-6 are used as dots on the screen. Remember that bit 7 is used as a color flag. The base 10 numbers associated with each bit are listed above each column. Note that bit 0 is on the left; those of you who've worked with binary or machine language would expect the opposite. (Many of us have, and kept getting mirror images of everything on the screen until the solution was discovered.)

To compute a number for each of the eight bytes, after marking each dot that you want set, go across each row and add the values in each column marked. In Figure 6.2 those numbers are at the right of each row. Notice that they match the numbers used in the DATA statement in Listing 6.5. You've now got a hi-res character generator for one character. In other words, a bit-mapped graphics generator that will put text and any other graphic designs on the hi-res screen. Try using different values in the DATA statement; you can create all kinds of interesting little graphics characters.

Listing 6.5

```
10  HGR
12  GOSUB 150
15  INPUT "X : ";X: IF X < 0 OR X > 39 THEN 15
16  INPUT "Y : ";Y: IF Y < 0 OR Y > 184 THEN
    16
20  FOR I = Y TO Y + 7
30  READ B
40  POKE YT(I) + X,B
50  NEXT I
60  RESTORE : GOTO 15
70  DATA  34,34,34,62,34,34,34,0
140  REM THIS SUBROUTINE CREATES A Y-LOOKUP
     TABLE, YT.
150  DIM YT(191)
160  FOR Y = 0 TO 191
200 Y1 =  INT (Y / 8):YR = Y - Y1 * 8
210 Y2 =  INT (Y1 / 8):YS = Y1 - Y2 * 8
220 YL = 8192 + Y2 * 40 + YS * 128 + YR * 1024
230 YT(Y) = YL
240  NEXT Y
250  RETURN
```

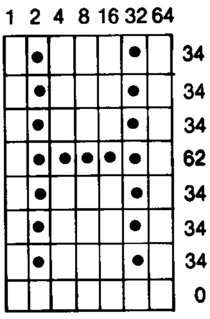


Figure 6.2 Hi-Res Character Grid

A Longer Basic Character Generator

Listing 6.6 has a more complete character generator, capable of more than one character. It doesn't have a complete alphabet defined, but it's got enough of a start where you can put in the data to do it.

Listing 6.6

```
10  HGR
12  GOSUB 150: GOSUB 300
15  INPUT "X : ";X: IF X < 0 OR X > 39 THEN 15
16  INPUT "Y : ";Y: IF Y < 0 OR Y > 184 THEN
    16
18  GET A$:A =  ASC (A$) - 65: IF A < 0 OR A >
    2 THEN 18
20  FOR I = 0 TO 7
40  POKE YT(Y + I) + X,CT(A,I)
50  NEXT I
60 X = X + 1: IF X > 39 THEN X = 0:Y = Y + 8:
   IF Y > 184 THEN Y = 0
70  GOTO 18
140  REM THIS SUBROUTINE CREATES A Y-LOOKUP
     TABLE, YT.
150  DIM YT(191)
160  FOR Y = 0 TO 191
200 Y1 =  INT (Y / 8):YR = Y - Y1 * 8
210 Y2 =  INT (Y1 / 8):YS = Y1 - Y2 * 8
220 YL = 8192 + Y2 * 40 + YS * 128 + YR * 1024
230 YT(Y) = YL
240  NEXT Y
250  RETURN
290  REM THIS SUBROUTINE CREATES A CHARACTER
     DEFINITION TABLE, CT
300  DIM CT(2,7)
310  FOR I = 0 TO 2
320  FOR J = 0 TO 7
330  READ CT(I,J)
340  NEXT J: NEXT I: RETURN
350  DATA  8,20,34,34,62,34,34,0
360  DATA  30,34,34,30,34,34,30,0
370  DATA  28,34,2,2,2,34,28,0
```

Another subroutine has been added at line 300, which defines a character table. Another array is used, CT, and is dimensioned 2,7 for three characters (0-2) of eight bytes each (0-7). The nested FOR-NEXT loops read the table from DATA statements, with I counting through the characters, and J counting through eight bytes for each character. The characters used in the DATA statements are A, B, and C from Figure 6.3.

Lines 18 and 60 are the only others of significant change. Line 18 gets a character from the keyboard. As soon as you press a key, that key's value is put in A$. The ASC function is then used to find the ASCII value of that character (there is an ASCII table in your Apple II *Applesoft BASIC Programming Reference Manual,* and in the IIe Manual, as well)[1]. This particular program will only recognize the keys A, B, and C, which have ASCII codes 65, 66, and 67, so after subtracting 65 we have a number that corresponds to our character table. You can do a lot of fiddling with that and the length of the table. Line 60 increments X after each character (so the next one is printed one space over to the right), and if the right edge of the screen is reached, Y is incremented and X set back to zero. When Y reaches the bottom of the screen, it is set back to the top.

1. *Apple II Applesoft BASIC Programming Reference Manual* (Cupertino: Apple Computer, Inc., 1979, 1981)
Kamins, Scot, *Apple IIe Applesoft BASIC Programmer's Reference Manual* (Cupertino: Apple Computer, Inc., 1982)

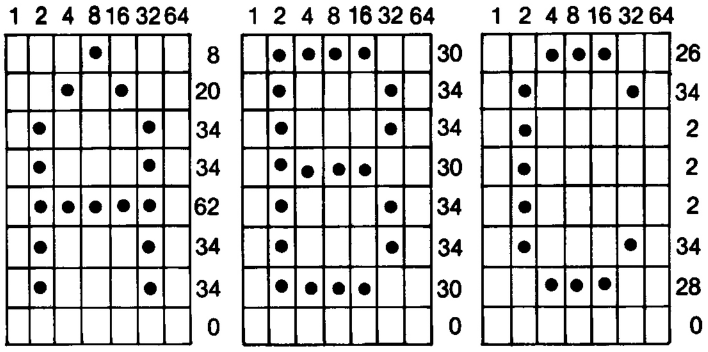


Figure 6.3 Hi-Res Character Grids

SEVEN

Machine Language Graphics Routines

The most difficult part about machine language is learning to think in hexadecimal, or base 16. Base 16 is shorthand for binary, the number system that your computer actually "understands." The best way to handle it is to *not* try converting to base 10 and back unless it is really necessary. Most of the time it isn't. The only time you'll have to convert is when you are trying to reference your machine language routines or addresses from Basic.

In base 10 (decimal), you count 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 ,11,.... In base 16 (hexadecimal), you count 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11,.... A is really what we call ten, B is 11, C is 12, and so on. F is 15, after which comes 10, or 16 (it's base 16, remember?).

To keep things straight, when referencing hexadecimal numbers, we'll precede the number with a dollar sign (why? because everyone else does!). So 2000 is the good old number two thousand you all know, while $2000 does not mean two thousand dollars (drat); it means hexadecimal two thousand, which is really 8192. "Slow down!", you say. Okay. Look at Figure 7.1. It shows numbers in binary, hexadecimal, and decimal. Notice how conveniently the numbers $0-$F correspond to all combinations of binary numbers that can fit in four bits. Remember also how conveniently a byte, the basic storage unit, has eight bits. That means that two hexadecimal digits can give you all the possible values for one byte ($00-$FF). That's why base 16 is such a convenient shorthand.

The largest hexadecimal number you'll be dealing with is four digits long, or $FFFF. That's because the Apple and other similar computers use two bytes to store addresses ($FF $FF, in this case). The number $FFFF in decimal is 65535, which gives the 64K that you hear about as the Apple's maximum memory size. One K is approximately 1000; it's actually 1024, or two to the tenth power. To convert hexadecimal numbers to decimal, use Figure 7.2 as an example. The rightmost of the four places is the one's column, the next is the 16's column, the next is for 16 to the second power, or 256, and the leftmost of the four places is for 16 to the third, or 4096.

Machine Language Coding

The first thing we'll do with machine language is to take Listing 6.4 from the last chapter and convert the graphics part. Remember that the program first creates a Y-lookup table (lines 150-250), then cycles through a loop, putting bytes with all the dots set onto the screen.

Binary	*Hexadecimal*	*Decimal*
0000	$0	0
0001	$1	1
0010	$2	2
0011	$3	3
0100	$4	4
0101	$5	5
0110	$6	6
0111	$7	7
1000	$8	8
1001	$9	9
1010	$A	10
1011	$B	11
1100	$C	12
1101	$D	13
1110	$E	14
1111	$F	15
10000	$10	16

Figure 7.1 Binary Hexadecimal Decimal Equivalents

```
$25C7 = $2*16    + $5*16  + $C*16 + $7*16
      = $2*4096 + $5*256 + $C*16 + $7*1
      = 2*4096  + 5*256  + 12*16 + 7*1
      = 8192    + 1280   + 192   + 7
      = 9671
```

Figure 7.2 Converting Hexadecimal to Decimal

Our first task will be to create a permanent Y-lookup table for our machine language routines. There's no need to recompute it all the time; once we have it we can BLOAD it into any program that needs it. To make it easy, we'll use a Basic program to do the computations and poke the values we want into memory. Then we'll save that portion of memory to disk, and we've got it whenever it's needed.

We'll put the lookup table just above the hi-res page 1 screen memory. Page 1 is located from addresses 8192 ($2000) to 16383 ($3FFF). That means our table should start at 16384, or $4000. For reasons that will become apparent later, we'll store the table in two sections: first the high bytes of all the addresses, then the low bytes. By ***high*** and *low* bytes, we mean that for an address such as $4F8A, the high byte is the left half, or $4F, and the low byte is $8A, the right half. Since there are 192 Y-values on the screen, the first part of the table, the high bytes, will take 192 bytes, from 16384 to 16575 ($4000-$40BF). The second half, the low bytes of the addresses, will be put in 16576 to 16767 ($40C0-$417F).

The high byte of a value is computed from decimal by dividing by 256 and chopping off the remainder. The low byte is the remainder of that same division. Listing 7.1 does all the same computations for the lookup table that we did last chapter, but then splits the address into high and low bytes and pokes them into memory. Type it in, then run it.

After you run the program, your lookup table is in memory. Now you want to save it to disk. Type:

```
BSAVE LOOKUP,A16384,L384
```

or, alternately,

```
BSAVE LOOKUP,A$4000,L$180
```

Listing 7.1

```
160  FOR Y = 0 TO 191
200 Y1 =  INT (Y / 8):YR = Y - Y1 * 8
210 Y2 =  INT (Y1 / 8):YS = Y1 - Y2 * 8
220 YL = 8192 + Y2 * 40 + YS * 128 + YR * 1024
230  POKE 16384 + Y, INT (YL / 256)
235  POKE 16576 + Y,YL -  INT (YL / 256) * 256
236  REM  THE RIGHT HALF OF LINE 235 HAS THE
     FORMULA FOR FINDING THE REMAINDER OF THE
     DIVISION YL/256
240  NEXT Y
```

The A means starting at which address, and the L means length. BSAVE means Binary SAVE.

You may want to BSAVE the file the same way on a few disks. We'll be using this lookup table a lot, and if you wind up doing much programming in machine language, you will find it frequently invaluable.

Assembly Language

Now for a short introduction to *assembly language*. Did I say machine language before? Well, assembly language is almost the same as machine language, except machine language is just numbers (hexadecimal, at that). Assembly language corresponds one-to-one with those machine language numbers, but its commands are mnemonics instead. In other words, you use letters that mean something to you instead of the numbers that the machine understands. To convert your assembly language mnemonics to machine language numbers, you need an assembler. An assembler is a program that interprets the assembly language instructions you write and pokes in the corresponding numbers for the machine. A couple of assemblers on the market that come recommended are those included with the *Applesoft/DOS Toolkit* and with *Merlin*. Of course you could just enter the finished numbers in the examples in these articles, but that's dull and boring and you really wouldn't learn much about *writing* in assembly/machine language.

This first little machine language routine takes an X and Y value that you give it, finds the Y lookup value in the lookup table, then puts the value $FF (255) at the corresponding byte on the screen. Not much, but it does illustrate a few of the instructions and addressing methods used in machine language. Let's go through Listing 7.2 line-by-line:

The first line says that the machine language routine will ORiGinate at address $6000 (24576). The assembler will start putting our instructions at that address in memory.

The next four lines are EQUates, or label definitions. The first says that whenever we use TEMPLO, we'll really mean the number $06. We could just as easily use the number $06 throughout, but it's not as easy to remember or to change later. In all cases in this example, the equates refer to addresses in memory that we'll be using in our program for storing things. We use them like variables in Basic, except we tell the computer exactly where in memory these storage locations should be.

TEMPLO and TEMPHI refer to addresses $6 and $7 in your computer. The first 256 bytes (addresses $0000 to $00FF) are referred to as zero-page, and can be accessed much faster and do some special things that other memory addresses cannot do. Most of the zero page is used by Applesoft and DOS (both of which are machine language programs, in reality). Addresses $6-$9 are free, however, so we'll use two of those.

LOOKHI and LOOKLO are also EQUated; they are the starting addresses of the two parts of our lookup table that we created and saved. Again, we could just use the addresses in our assembly language program when needed, but the labels give them a little more meaning.

Next, we define two more bytes of storage (DFB means DeFine Byte). With these two we don't really care where exactly they go in memory, we just want them in there. As it is, they are the first two bytes actually set aside by our assembly language program, so they'll be put at $6000 and $6001 (the ORG told the assembler to start at $6000, and the EQUates don't take any storage themselves, since they just tell the assembler that a label means a particular number). We defined the bytes as having values of zero, although that doesn't matter, because it's in these locations that we'll poke our X and Y values.

Now the program starts. The first instruction is to LoaD Y with the number in YVALUE. The Apple has three main *registers*, which are single bytes set aside inside the actual 6502 microprocessor. These are accessed *very* quickly, and most instructions center around use of these bytes (and one other). The registers are labeled A, X, and Y. A is the *accumulator*, where most everything happens, including all mathematics and logic operations. X and Y are used mostly as pointers, offsets, and counters. LDY YVALUE means to load the Y register with the number

Listing 7.2

```
        ORG  $6000
TEMPLO  EQU  $06
TEMPHI  EQU  $07
LOOKHI  EQU  $4000
LOOKLO  EQU  $40C0
XVALUE  DFB  0
YVALUE  DFB  0
START   LDY  YVALUE
        LDA  LOOKLO,Y
        STA  TEMPLO
        LDA  LOOKHI,Y
        STA  TEMPHI
        LDA  #$FF
        LDY  XVALUE
        STA  (TEMPLO),Y
        RTS
```

in YVALUE in memory. We'll be using this number as an offset, much as you would in using an array in Basic. In fact, we'll use it as an offset in our lookup table, just as we did in the Basic examples with arrays.

The next line says to LoaD A with the value in the address LOOKLO, offset by Y. Given our EQUates, that means that it will load the accumulator with the value in address $40C0 + Y. It's exactly like using an array in Basic!

That is followed by STore A in TEMPLO. This instruction takes the value that we just loaded into A from the lookup table, and puts it in address TEMPLO ($06). Note that in assembly/machine language, you can't really say "take something from here in memory and put it over there." You have to load it into one of the registers from a memory location, then store the contents of that register in another memory location.

The next two instructions duplicate the load and store commands for the high byte from the table. That was easy!

Next, we load the accumulator (LDA) with the number $FF (255). Note that the # sign means to use the number following it. If the # were left off, the instruction would mean LoaD A with whatever is in address $FF. A frequent error is leaving off the # and staying up late at night wondering why programs do such strange things. About the time you convince yourself that your computer is broken, you usually discover the missing # that changed the entire meaning of the program. Grrrr... You have to tell it everything!

Now for some tricky maneuvering. We mentioned that page zero addresses had some special functions that allows them to do things other addresses can't. One is called *indirect addressing*. The next two instructions, LDY XVALUE (put XVALUE in the Y register), and STA (TEMPLO),Y put the value $FF from the accumulator onto the screen at the address we want. The previous loads and stores had put the base address of the screen line from our lookup table into TEMPLO and TEMPHI. (Note that the low byte of the address went in the first address.) STA (TEMPLO),Y says to store the contents of the accumulator in the address contained in TEMPLO and its following byte (TEMPHI), offset by Y. In other words, take the address stored in TEMPLO and TEMPHI, add the value in Y, and store the contents of the accumulator in the resulting location. In this program, it takes the base address of the line that we stored in TEMPLO and TEMPHI, adds the X value, and stores the number $FF in the resulting location.

Confusing, perhaps, but that's also the most tricky that machine language addressing gets. If you can handle that, the rest of machine language will be relatively easy.

The last line is a ReTurn from Subroutine, which is the equivalent of a Basic RETURN statement. It means to go back to the instruction from whence it was called.

Listing 7.3 shows the same program after it has gone through the assembly process. After assembling, you are shown the addresses of each assembled instruction and the actual hexadecimal values to which those instructions were converted. You can hand enter this short routine with the commands in Listing 7.4.

Listing 7.3

```
                           ORG   $6000
                  TEMPLO   EQU   $06
                  TEMPHI   EQU   $07
                  LOOKHI   EQU   $4000
                  LOOKLO   EQU   $40C0
6000: 00          XVALUE   DFB   0
6001: 00          YVALUE   DFB   0
6002: AC 01 60    START    LDY   YVALUE
6005: B9 C0 40             LDA   LOOKLO,Y
6008: 85 06                STA   TEMPLO
600A: B9 00 40             LDA   LOOKHI,Y
600D: 85 07                STA   TEMPHI
600F: A9 FF                LDA   *$FF
6011: AC 00 60             LDY   XVALUE
6014: 91 06                STA   (TEMPLO),Y
6016: 60                   RTS
```

Listing 7.4

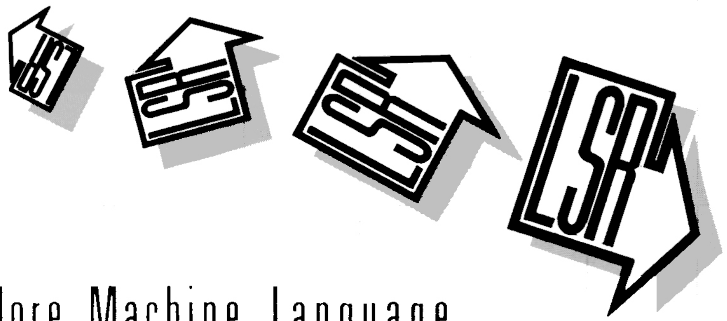
```
Entering the Machine Language Program Directly

Note: The bracket (]) and asterisk (*) charac-
      ters at the beginning of each line are
      prompts. You type the rest. After each
      line, press Return.

]CALL -151
*6000:00 00 AC 01 60 B9 C0 40
*6008:85 06 B9 00 40 85 07 A9
*6010:FF AC 00 60 91 06 60
*3D0G
]BSAVE PLOT,A24576,L23
```

Finally, Listing 7.5 is a Basic program that BLOADs the lookup table and machine language program, then loops through, pokes the X and Y values into locations 24576 ($6000) and 24577 ($6001), and then calls the subroutine at 24578 ($6002). Notice the similarities to and differences from Listing 6.4. The new version may not seem much faster than the old, since most of the work is still being done in Basic, and the machine language routine itself is very short with no repetition. But as the tasks become slightly more complex, the speed differences in machine language are incredible, as you'll soon find for yourself.

Listing 7.5

```
5   PRINT  CHR$ (4);"BLOAD LOOKUP"
6   PRINT  CHR$ (4);"BLOAD PLOT"
10   HGR
15 X = 0:Y = 0:XC = 1:YC = 1
20   POKE 24576,X: POKE 24577,Y: CALL 24578
70 X = X + XC: IF X < 1 OR X > 38 THEN XC =
    - XC
80 Y = Y + YC: IF Y < 1 OR Y > 190 THEN YC =
    - YC
90   GOTO 20
```

E I G H T



More Machine Language

Now we'll continue experimenting with assembly language graphics by taking the Basic character generator from Chapter 6 and the start of a machine language routine from Chapter 7, and finish writing a machine language character graphics generator. This chapter is actually more about assembly language than graphics itself, but to do fast graphics you have to know some of the tricks to doing them quickly. Playing with fast graphics routines also happens to be a nice, visual way to learn some assembly language.

The routine from the last chapter, when given X and Y screen values, would find the address of the Y line in memory from a lookup table we created, then, using what's called *indirect indexed addressing*, put the value $FF in the screen location we'd specified. The indirect addressing happened when we stored the address of the beginning of the screen line in a pair of locations, TEMPLO and TEMPHI, then loaded the X-offset on the screen into the Y-register, and used the command:

```
STA (TEMPLO),Y
```

The parentheses around TEMPLO meant that the computer should find the address stored in TEMPLO and the following location (which we named TEMPHI), and add the contents of the Y-register to that to get the final address in which to store something.

In the process, we also used absolute indexed addressing to get numbers out of our lookup table with commands like:

```
LDA LOOKLO,Y
```

Without the parentheses, it means to take the address LOOKLO and add the contents of the Y-register to get the final address.

Note the differences between the two types of indexed addressing we used. Both give something like an array in Basic, but the absolute indexed (the one without parentheses) uses the exact value of the address you specify plus Y. Indirect indexed finds the address *in* the location you give, then adds Y.

Why these two types? Because the Y-register is only one byte, and can contain only the values 0 to 255. So with either addressing mode, your "array" can contain only 256 numbers. But with the indirect indexed we can change the ***base address***. Going back to the short routine from last chapter, we couldn't index 8,192 bytes of a hi-res screen with only the Y-register. But we could index a single line on the screen, because it's only 40 bytes long. We wouldn't want 192 labels and 192 different plot routines, one for each line on the screen. So we compute the base address of each line (actually using the lookup table) and put that in a location that we use for indirect indexing from only one routine. The only restrictions of indirect indexed are that the location where you put the address must be on page zero ($00-$FF), must start on an even address (we used $06), and must be in LO,HI format (in a 4-digit address, such as $12CD, the right digits, $CD, go in the first byte, and the left digits, $12, go in the second byte).

If it seems terribly confusing to you at first, you're not alone. Addressing is the most difficult concept to grasp when first learning assembly language.

Time For Another Routine

Instead of putting only one byte on the screen at a time, we'll take our Basic character generator and convert it to machine language so that we can put any of 128 characters on the screen. Each character is eight bytes; one byte wide (7 dots) and eight bytes tall. Before, we had the character definitions stored in a Basic array. Now, we'll store them in a binary table, and load each byte using indirect indexed addressing (128 characters times 8 bytes gives 1024 bytes; too much for using absolute indexed).

To start, we'll create a character table using the program in Listing 8.1. You may notice that most of it is taken directly from the last program in Chapter 6, except instead of storing in an array, we're poking into memory. We'll put the table at $7000, or 28672 decimal. The letters A, B, and C, which are the three we defined before, have ASCII values 65, 66, 67. (ASCII is the standard character-to-number translation used by most computers. There's a table of ASCII values for each character in the Apple II *Applesoft BASIC Programming Reference Manual,* and in the IIe Manual)[1]. Anyway, each character takes 8 bytes, so to find the first byte of the Nth character, you would look in 28672+8*N. That's where the POKE address in Listing 8.1 comes from.

This program only has three character definitions. Others are left to you, using the technique from Chapter 6. There are also shortcuts. If you have the *Applesoft/DOS Toolkit, Higher Text,* or *The Complete Graphics System,* the small character sets from all of those use the same format. In Listing 8.3, where we actually use our finished routine from Basic, you can substitute a BLOAD of any small character set at location $7000 (because some of them do not use the first 32 characters, try BLOADing at $7100 if the letters don't match up).

After storing a character table on your disk, either with Listing 8.1 or through other means, we're ready to do the assembly language routine. We'll go through Listing 8.2 step by step. Note that this is an assembled listing so the numbers corresponding to the commands have already been generated in the left columns by the assembler program.

Note also that this routine was written with the *Merlin* assembler. Some of the conventions differ with other assemblers, and we'll try to point out some differences. Also, if you are really interested in pursuing even a small amount of assembly language programming, invest in an assembler and a 6502 assembly language reference book that has a chart of the 6502 assembly language commands. Otherwise it's going to seem like I'm pulling all these commands out of a hat, making them up as I go. There are 55 commands, but many are seldom used.

1. *Apple II Applesoft BASIC Programming Reference Manual* (Cupertino: Apple Computer, Inc., 1979, 1981)
Kamins, Scot, *Apple IIe Applesoft BASIC Programmer's Reference Manual* (Cupertino: Apple Computer, Inc., 1982)

Listing 8.1

```
290  REM  THIS CREATES PART OF A CHARACTER
     DEFINITION TABLE, FROM ASCII 65 TO 67
310  FOR I = 65 TO 67
320  FOR J = 0 TO 7
330  READ V: POKE 28672 + 8 * I + J,V
340  NEXT J: NEXT I
345  PRINT  CHR$ (4);"BSAVE CHARTABLE,A$7000,
     L$100": END
350  DATA  8,20,34,34,62,34,34,0
360  DATA  30,34,34,30,34,34,30,0
370  DATA  28,34,2,2,2,34,28,0
```

Listing 8.2

```
                          ORG   $6000
                 TEMPLO   EQU   $06
                 TEMPHI   EQU   $07
                 CTABLO   EQU   $8
                 CTABHI   EQU   $9
                 LOOKHI   EQU   $4000
                 LOOKLO   EQU   $40C0
                 CHRTAB   EQU   $7000
6000: 00         CHAR     DFB   0
6001: 00         XVALUE   DFB   0
6002: 00         YVALUE   DFB   0
6003: A9 00      START    LDA   #<CHRTAB
      ;PUT CTABLE ADDRESS
6005: 85 08               STA   CTABLO
      ;IN CTABLO, CTABHI
6007: A9 70               LDA   #>CHRTAB
6009: 85 09               STA   CTABHI
600B: AD 00 60            LDA   CHAR
      ;GET CHARACTER NUMBER
600E: 4A                  LSR
600F: 4A                  LSR
6010: 4A                  LSR
6011: 4A                  LSR
6012: 4A                  LSR
      ;DIVIDE BY 32
6013: 18                  CLC
6014: 65 09               ADC   CTABHI
      ;ADD TO HIGH BYTE OF ADDRESS
6016: 85 09               STA   CTABHI
6018: AD 00 60            LDA   CHAR
601B: 29 1F               AND   #$1F
      ;FIND REMAINDER AFTER DIVIDING BY 32
601D: 0A                  ASL
601E: 0A                  ASL
```

Listing 8.2 (continued)

```
601F: 0A                     ASL
      ;MULTIPLY BY EIGHT
6020: 18                     CLC
6021: 65 08                  ADC   CTABLO
      ;ADD TO LOW BYTE OF ADDRESS
6023: 85 08                  STA   CTABLO
6025: A2 00                  LDX   #0
      ;X REG WILL GO FROM 0 TO 7
6027: AC 02 60      LOOP     LDY   YVALUE
      ;Y LOCATION INTO Y REGISTER
602A: B9 C0 40               LDA   LOOKLO,Y
      ;GET ADDRESS OF YTH LINE FROM
602D: 85 06                  STA   TEMPLO
      ;LOOK-UP TABLE AND PUT IN
602F: B9 00 40               LDA   LOOKHI,Y
      ;TEMPLO,TEMPHI
6032: 85 07                  STA   TEMPHI
6034: 8A                     TXA
      ;GET THE NEXT BYTE FROM
6035: A8                     TAY
6036: B1 08                  LDA   (CTABLO),Y
      ;THE CHARACTER TABLE
6038: AC 01 60               LDY   XVALUE
      ;STORE THE BYTE ON THE
603B: 91 06                  STA   (TEMPLO),Y
      ;SCREEN
603D: EE 02 60               INC   YVALUE
      ;NEXT LINE
6040: E8                     INX
      ;HAVE WE DONE 8 LINES?
6041: E0 08                  CPX   #8
6043: D0 E2                  BNE   LOOP
      ;IF NOT, DO IT AGAIN
6045: 60                     RTS
```

The first 11 lines should look familiar from the last chapter. We added a few extra EQUates and a new DeFine Byte, but the rest is the same. The routine will ORiGinate at $6000. We'll use two pairs of page zero addresses for indirect indexing: TEMPLO and TEMPHI for the screen line addresses taken from the lookup table, and CTABLO and CTABHI for addressing the character table. LOOKHI and LOOKLO, the pointers to the lookup table, are the same as before. CHRTAB is the pointer to the beginning of the character table.

CHAR is where we will put the ASCII value of the character we want printed. XVALUE and YVALUE are the X,Y location to print on the screen, as before.

Okay. The beginning of the program loads the address CHRTAB into the accumulator, half at a time, and stores it in CTABLO and CTABHI. The symbol #<CHRTAB means the number that is the low byte of the address CHRTAB. #>CHRTAB is the number which is the high byte of the address. At the far left, you see the hex numbers that are generated when the code is assembled. Note that at $6004, you get the values A9 00. A9 is the LDA code, and 00 is the value given #<CHRTAB. Two lines down, you get A9 70; again the LDA code, and then #>CHRTAB. In the beginning you EQUated CHRTAB with $7000, which is where 00 and 70 come from. The nice thing is you don't worry much about the numbers generated; the assembler does that for you.

(Note: The #> and #< format varies with assemblers. If all else fails, do two EQUates for the address; something like:

```
CHRTAH EQU $70
CHRTAL EQU $00
```

and use those in place of #>CHRTAB and #<CHRTAB)

A little multiplication and division by powers of 2

The next part may be tricky. What we're trying to do is get CTABLO and CTABHI to point to the beginning of the letter we want. Now it points to the beginning of a 1,024 ($400) byte table. A page of memory (not related to hi-res pages) is 256 ($100) bytes. The high byte of any address gives its page number in memory. The character table takes 4 pages ($400 bytes). Each page holds 32 characters (256/8 bytes = 32 characters), so CTABHI should be $70 if the character number is 0 to 31, $71 if the code is 32 to 63, $72 if the code is 64 to 95, or $73 if it's 96 to 127. It's $70 now; what we have to add is the character code divided by 32.

Similarly, the low byte should point to where in that 256-byte page the 8 bytes for the letter start. To do that, we need the remainder after dividing the character code by 32 (i.e., is it character 0-31 on that page?), multiplied by 8.

For example, say we want character 36. Character 36 would be on the second page of the character table, since from $7000 to $70FF are the characters 0-31. 36 divided by 32 is 1, with a remainder. The page number that character 36 is on is $70+1, or $71. Furthermore, it is character 4 on that page (characters 32, 33, 34, 35, 36...., are characters 0 to 4, and so on, on page $71.) Since each character is 8 bytes long, the definition for character 36 starts at byte 32 of that page (8 times 4).

Now come the three tricky commands. LSR stands for *Logical Shift Right*. It takes all the bits in a byte and moves them to the right one place. The rightmost bit gets thrown away (although you can still find it, if you need it). The effect of this is dividing by two and throwing away the remainder! Examples (with 4 bits):

Before LSR		*After LSR*	
0010	(2)	0001	(1)
1000	(8)	0100	(4)
1010	(6)	0101	(3)
1101	(13)	0110	(6)

By doing two LSRs in a row, it's like dividing by 4. With 3, it's dividing by 8. And with 5, it's dividing by 32! That's why the next six lines of the listing load the character number, then do 5 LSRs.

Similarly, ASL is *Arithmetic Shift Left*. It does the same thing, but in the other direction. It's equivalent to multiplying by two. More than one in sequence allows you to multiply by 2, 4, 8, 16, and so on. A few lines down, when we need to multiply by 8, you see 3 ASLs.

AND is a logical and operation. If you remember some logic operations from high school, with AND you get true as a result only if the two things you are ANDing are true. In binary, trues are ones. To show you how it works:

```
    10110
AND 00111
---------
    00110
```

Notice that only when both corresponding bits are set will the same bit in the result be set.

There is also an OR command and an Exclusive OR command, and all three of these logic operators are extremely useful with graphics. Here, though, we're using AND as a short way to find the remainder of a division by 32. Line 26 in the listing has an AND #$1F. $1F is 31 decimal, or 00011111 in binary. ANDing it with any number gives the remainder of a division by 32. The left 3 bits in the byte would give the result of the division. Try it with a few numbers.

Back to the program. The five LSRs all affect the accumulator. Most assemblers accept LSR by itself as meaning that the accumulator is used. The *Applesoft/DOS Toolkit* assembler requires the operand A to specify Accumulator. Depending on the assembler you use, you may have to insert or omit the A.

After the LSRs is a CLear Carry. There is one bit called the carry bit that holds the carry for any addition. That allows numbers larger than 255 to be used in arithmetic. It is also used in other operations, such as subtraction, and it is where the chopped off bit in LSRs and ASLs get "thrown away." Since the next step is an addition, we don't want any junk messing up the addition operation, so CLear Carry sets the carry bit to zero.

ADC is ADd with Carry. It adds the number pointed to, in this case CTABHI, plus the carry bit, to the accumulator. Then we store the result (STA), which is in the accumulator, back in CTABHI. CTABHI now points to the correct page for the character we want.

The next two lines load the character number again, and then use AND to find the remainder of a division by 32. Then the 3 ASLs multiply by 8, and we have the offset on the page for the character we want. So we use CLC again, and add this offset to CTABLO, storing the result back in CTABLO. (Note: again, for ASL, the A operator isn't used by all assemblers. Also, in our example, we started the table on a page boundary, $7000. We could have omitted this last addition and a couple other steps because we are just adding to zero. The routine is more generalized this way, so at some later time you could put your table anywhere you want in memory, if need be. The only change you'd have to make is in the EQUate at the beginning.)

Now for a loop. We have 8 bytes to put on the screen, and CTABLO/CTABHI points to the first of these. For fun, we'll use the X-register as a counter, from 0 to 7 (stopping when it reaches 8). LDX #0 does what you probably think: loads the X-register with the number zero.

The first five lines in the loop you should recognize. They are still here from the short routine last chapter. We take the Y-value, find the address of the start of that line in our lookup table, and put that address in TEMPLO/TEMPHI. Next we do something similar to get the first (or next) byte of our character. The X-register is our counter. TXA Transfers X to A. TAY Transfers A to Y. (There is no TXY; too bad.) We transferred our counter to the Y-register because it's the Y-register that must be used for indirect indexed addressing. So now we use it to get the byte for our character in line 41. The next two lines are also from our first routine last month; they store the accumulator value on the screen at the proper X-offset.

That being done, we INCrement YVALUE (add one to it) so that the next byte will go one line down. Simple. Then we INcrement the X-register with the INX command. Okay. Then ComPare X to the number 8 (CPX #8) and Branch if the comparison was Not Equal (BNE is Branch if Not Equal) to LOOP and repeat. If the comparison was equal, continue on the next line, which is a ReTurn from Subroutine.

Done!

Last, there's Listing 8.3, which is a Basic program that BLOADs the three binary files we need, gets a starting X,Y location, and loops through getting a key and poking the ASCII value of the character and the X and Y values into the memory locations of the plot routine and calling the routine. Compare Listing 8.3 to Listing 6.6. It's the same program, with the graphics changed to machine language. Remember that unless you added your own characters, the only letters that will make sense on the screen are A, B, and C. Adding the others is left to you.

Listing 8.3

```
10  HGR
12  PRINT  CHR$ (4);"BLOAD LOOKUP,A$4000"
13  PRINT  CHR$ (4);"BLOAD LISTING 8.2,A$6000"
14  PRINT  CHR$ (4);"BLOAD CHARTABLE,A$7000"
15  INPUT "X : ";X: IF X < 0 OR X > 39 THEN 15
16  INPUT "Y : ";Y: IF Y < 0 OR Y > 184 THEN
    16
18  GET A$
20  POKE 24576, ASC (A$): POKE 24577,X: POKE
    24578,Y: CALL 24579
60 X = X + 1: IF X > 39 THEN X = 0:Y = Y + 8:
   IF Y > 184 THEN Y = 0
70  GOTO 18
```

NINE

The Wonderful World of Color

Now we'll diverge from teaching assembly language and take a look at hi-res colors on the Apple. If you're interested in learning more assembly language, now's the time to look into other books and references on the subject, such as Roger Wagner's *Assembly Lines*.[1]

How Many Colors?

So how does it work? Some of these places advertise that they can give you anywhere from 20 to over a hundred colors. Well, what they (we) do is combine the existing six colors in various patterns to make it look like there are more. It's effective. There are hi-res pictures done in several shades of a single color alone that look really nice; not the same as a computer that has a few hundred pure colors built in, but definitely stretching your normal Apple to its limits. To see how it's done, we can do a couple of experiments. Then, we'll list a machine language fill routine that uses 108 color combinations (it's the standard fill and color set used in the Penguin Software graphics products).

Black

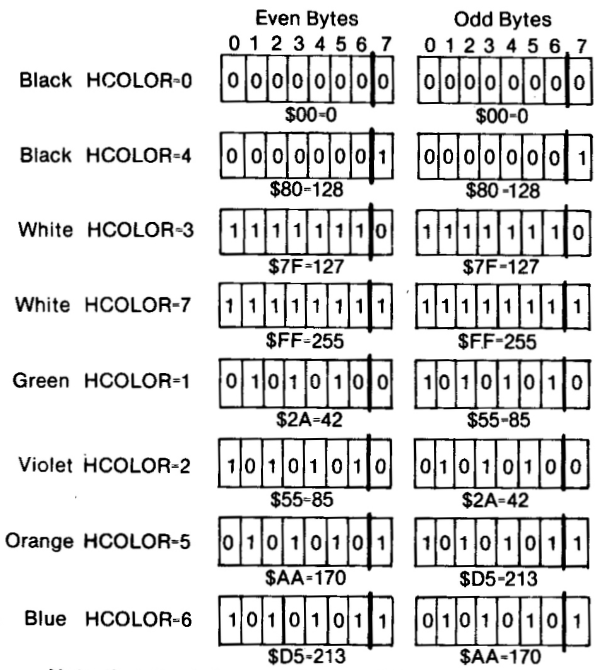
First, drag out the machine language plot routine and your hi-res lookup table from Chapter 7. Now, to fill the screen with black, all we have to do is to put the number $00 or $80 in every byte of the screen. These numbers give the two different blacks. In one, all the bits are off. In the

1. Wagner, Roger, *Assembly Lines* (North Hollywood: Softalk Publishing Inc., 1982)

other, all bits except the high bit (color flag) are off. See Figure 9.1 for this and other color patterns.

White

Similarly, to fill the screen with white, you need to store $7F or $FF in every screen byte. Listing 9.1 shows a Basic routine to do any of these with black or white, using the plot routine and the lookup table. To run a straight Basic program in parallel, instead of poking values into specific bytes, see Listing 9.2. For a quick version of these first few examples (using the "pure" Apple colors), you would more likely want to use the method in Listing 9.3.



		Even Bytes 0 1 2 3 4 5 6 7		Odd Bytes 0 1 2 3 4 5 6 7	
Black	HCOLOR=0	0 0 0 0 0 0 0 0	$00=0	0 0 0 0 0 0 0 0	$00=0
Black	HCOLOR=4	0 0 0 0 0 0 0 1	$80=128	0 0 0 0 0 0 0 1	$80=128
White	HCOLOR=3	1 1 1 1 1 1 1 0	$7F=127	1 1 1 1 1 1 1 0	$7F=127
White	HCOLOR=7	1 1 1 1 1 1 1 1	$FF=255	1 1 1 1 1 1 1 1	$FF=255
Green	HCOLOR=1	0 1 0 1 0 1 0 0	$2A=42	1 0 1 0 1 0 1 0	$55=85
Violet	HCOLOR=2	1 0 1 0 1 0 1 0	$55=85	0 1 0 1 0 1 0 0	$2A=42
Orange	HCOLOR=5	0 1 0 1 0 1 0 1	$AA=170	1 0 1 0 1 0 1 1	$D5=213
Blue	HCOLOR=6	1 0 1 0 1 0 1 1	$D5=213	0 1 0 1 0 1 0 1	$AA=170

Note that the bytes are displayed (and listed here) with bit 0 at left and bit 6 at right. Bit 7, the color flag, is not displayed. Since the bits are displayed in reverse order, the order must be reversed again to convert to hexadecimal—or you can turn the page upside down.

Figure 9.1 Standard Apple Color Patterns

Colors!

Now, to get a color other than black or white, you may recall that only half the dots are used. Blue and violet have only the dots in the even columns set. Orange and green have only the dots in the odd columns set. Blue and orange require the high bit (color flag) to be set. Violet and green require that it be off. Look at Figure 9.1 again to see the patterns. The catch is that there are only 7 dots shown per byte. That means that if the leftmost byte on a screen line (byte zero; even) has only bits 0, 2, 4, and 6 set, it will show as violet (since 0, 2, 4, and 6 will fall on screen columns 0, 2, 4, and 6). But for the next byte, bit 0 corresponds to column 7 on the screen—an odd column. So if the same pattern (0, 2, 4, 6) is used, that byte will show as green! (Bits 0, 2, 4, and 6 in that byte correspond to columns 7, 9, 11, and 13.) To continue with violet across the screen, byte 1 has to have a different pattern: bits

Listing 9.1

```
5  PRINT  CHR$ (4);"BLOAD LOOKUP"
6  PRINT  CHR$ (4);"BLOAD PLOT"
10  HGR
20  FOR Y = 0 TO 159
30  FOR X = 0 TO 39
40 C = 255: REM  255 = $FF
49  REM  COLORCANBEPOKEDINAT24592($6010)
    50  POKE 24576,X: POKE 24577,Y:POKE
    24592,C: CALL 24578
60  NEXT : NEXT
```

Listing 9.2

```
10  HGR
20  FOR Y = 0 TO 159
40  HCOLOR= 7
50  HPLOT 0,Y TO 279,Y
60  NEXT
```

Listing 9.3

```
10  HGR
20  HCOLOR= 7: HPLOT 0,0
30  CALL 62454
```

1, 3, and 5 should be set, with the others off. Continuing, byte 2 would go back to the original pattern (0, 2, 4, 6), byte 3 would go back to 1, 3, 5, and so on.

Listing 9.4 uses the plot routine to put the correct byte values on the screen for violet. By changing the even and odd values to correspond with the other patterns in Figure 9.1, you could also do green, blue, or orange. Of course, you could still take the Basic shortcuts from Listings 9.2 and 9.3.

Figure 9.1 shows the patterns for the standard colors. You could, of course, make up your own patterns. Try using different numbers in Listing 9.4—make up some. The key to a good-looking color is a regularly repeating pattern. Another hint for good patterns follows with the next example.

Alternate Rows of Colors

Here's where we play some tricks with your eyes. What we're going to do is alternate rows of colors. Not much explanation is needed; just watch what happens. Listing 9.5 does it with the plot routine and the colors orange and green. You can change the color patterns easily in the program to try others. Listing 9.6 does the same thing strictly in Basic and with only the six Apple colors. It's set up in a loop so that it will cycle through every combination of the six. You may even try playing with the color intensity control on your monitor or TV. Tweaking it up more exaggerates the effect considerably.

Listing 9.4

```
5  PRINT  CHR$ (4);"BLOAD LOOKUP"
6  PRINT  CHR$ (4);"BLOAD PLOT"
10  HGR
20  FOR Y = 0 TO 159
30  FOR X = 0 TO 39
39  REM  CHECK IF X IS EVEN
40  IF X / 2 =  INT (X / 2) THEN C = 85: GOTO
    50
44  REM  X IS ODD
45 C = 42
50  POKE 24576,X: POKE 24577,Y: POKE 24592,C:
    CALL 24578
60  NEXT : NEXT
```

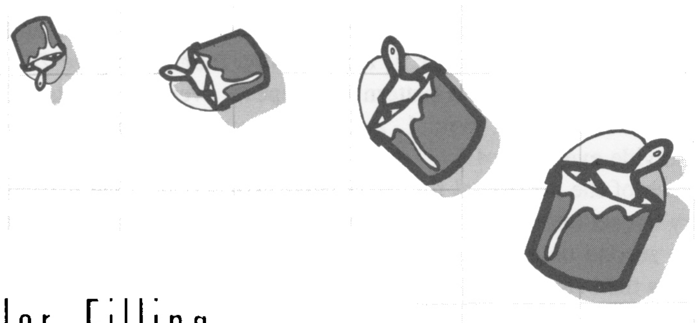
Listing 9.5

```
5  PRINT  CHR$ (4);"BLOAD LOOKUP"
6  PRINT  CHR$ (4);"BLOAD PLOT"
10  HGR
20  FOR Y = 0 TO 159
30  FOR X = 0 TO 39
31  REM  IF Y IS EVEN USE ONE PAIR, IF ODD,
    USE ANOTHER.
32  IF Y / 2 =  INT (Y / 2) THEN 40
34  IF X / 2 =  INT (X / 2) THEN C = 42: GOTO
    50
36 C = 85: GOTO 50
39  REM  CHECK IF X IS EVEN
40  IF X / 2 =  INT (X / 2) THEN C = 170: GOTO
    50
44  REM  X IS ODD
45 C = 213
50  POKE 24576,X: POKE 24577,Y: POKE 24592,C:
    CALL 24578
60  NEXT : NEXT
```

Listing 9.6

```
20  FOR C1 = 1 TO 6
30  FOR C2 = C1 TO 6
40  HGR
50  HOME : VTAB 21: PRINT "COLORS ";C1;" AND
    ";C2
60  FOR Y = 0 TO 159
70  IF Y / 2 =  INT (Y / 2) THEN  HCOLOR= C1:
    GOTO 90
80  HCOLOR= C2
90  HPLOT 0,Y TO 279,Y
100  NEXT Y
105  PRINT "PRESS ANY KEY": GET A$
110  NEXT C2: NEXT C1
```

TEN



Color Filling

In this chapter we'll describe a machine-language color fill routine. It's a routine that will let you fill any enclosed area on the screen in one of over a hundred color combinations. Because the routine is written for speed, some areas need two or three fills to fill entirely. Complete fill routines are so far much slower, and in many instances (such as the PICDRAW routine described in Chapter 14) much too slow for an application.

Fill Routine

The routine in Listing 10.1 uses the following steps to fill an area (the area must be white, bordered by black lines or the edge of the screen):

1. Given a point at which to start the fill, search directly upward until a point is found that is off, then step back down one line.
2. Start filling to the left, until a border is found. Remember the end point, then fill to the right and remember the right end point.
3. Average the left and right endpoints (giving the midpoint of that segment). Step down one line from that point. If it's not a border, go back and repeat step 2, else end.

These steps work well for most regular shapes. The only hint on using them is that the point selected for step 1 should be directly below the highest point in the area to be filled.

The complete listing of this program appears in Appendix F.

Getting Colors

How do we get all the colors? We use patterns that are four bytes wide and two bytes tall. Vertically, odd lines have one pattern, and even lines another; just like in Chapter 9. Horizontally, four bytes give 28 dots. That means we can set a pattern that repeats every four dots. (28 is evenly divisible by 4. With 1, 2, or 3 bytes, none of the 7, 14, or 21 dots divide evenly by 4.)

Each of the horizontal four-byte patterns is stored. One pattern has every dot set (all white), one has every dot off (black), and others have every second or every fourth dot set for various color patterns. Each color is then stored as a pair of these horizontal patterns: one for even rows, one for odd rows.

The routine requires the Y-lookup table again, this time loaded at $6000. The routine itself starts right after the lookup table, at $6180. To use it, you need four pokes and a call:

```
POKE 24960, color number
POKE 24961, INT(X/256)
POKE 24962, X-INT(X/256)*256
POKE 10, Y
CALL 24963
```

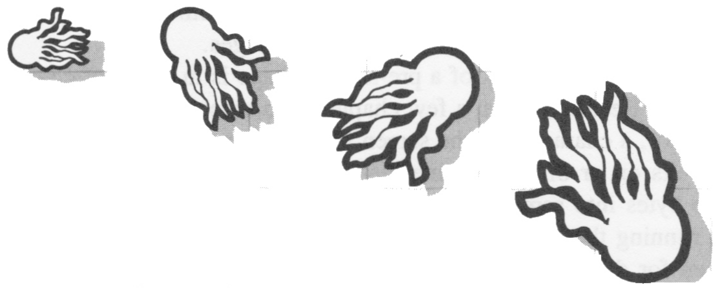
Listing 10.2 shows an example of loading and using the fill routine from a program.

Listing 10.2

```
10  REM  LISTING 10.2
20  PRINT  CHR$ (4)"BLOAD LOOKUP,A$6000"
30  PRINT  CHR$ (4)"BLOAD LISTING 10.1,A$6180"
40  HGR
45  REM  CLEAR SCREEN TO WHITE
50  HCOLOR= 7: HPLOT 0,0: CALL 62454
60  HCOLOR= 4
65  REM  DRAW A RECTANGLE IN BLACK
70  HPLOT 20,20 TO 180,20 TO 180,80 TO 20,80
    TO 20,20
75 C = 43
80 X = 40:Y = 50
85  REM  FILL THE RECTANGLE
90  POKE 24960,C: POKE 24961, INT (X / 256):
    POKE 24962,X -  INT (X /
     256) * 256: POKE 10,Y: CALL 24963
```

For this routine, as for others in this book, we do ask that you acknowledge copyright and authorship in your own programs.

ELEVEN



Fast Animation

Now we get down to some of the nitty-gritty techniques used for creating fast animation in arcade games. We covered basic animation ideas using shape tables, but they are much too slow and cause too much flickering for really professional-looking results. Here we start creating some real animation.

Pre-Shifted Shapes

One of the techniques for fast, smooth movement is an invention called a *pre-shifted shape*. It should be noted from the outset that the term *pre-shifted shape* has a few different definitions, depending on to whom you talk. It seems that a couple years ago, when everyone's ideas for fast animation techniques started filtering around, *pre-shifted shapes* was the magic phrase, although definitions didn't always accompany the phrase. So when people happened on techniques that caused fast animation, many assumed they had discovered the secrets of pre-shifted shapes, even if it wasn't the original technique. It's just as well, for now we have that many more approaches to work with.

Page Flipping

Page-flipping was, and is, another magic phrase for smooth animation. It's simple to understand: use both hi-res graphics screens, show one, erase and redraw on the other, and flip pages (display the new one). That way you don't see the flicker caused by the erase and redraw cycle. Page-flipping, to be fast, has to be used along with some form of pre-shifted shapes or a similarly fast animation method. There are, however, ways to achieve the smooth results of page-flipping within the structure of pre-shifted shapes without actually flipping pages.

The first way to think of a pre-shifted shape is in terms of character graphics. Recalling from a few chapters ago when we did a hi-res text generator, a character on the screen is nothing more than a set of bytes with bits on or off that define the shape of the character. By putting those bytes into screen memory, the shape is displayed. Try typing in and running the program in Listing 11.1. It uses characters on the text screen for doing animation like we did earlier with shape tables. The same results could be achieved on the hi-res screen using the hi-res character generator we wrote earlier.

Analyzing the animation, it is fast, but also very jumpy. The movement of the character is by large steps rather than small ones, sort of like watching someone's movements under a strobe light instead of under normal lighting. There are reasons for both the speed and the jumpiness. A very simple example explains it.

Suppose our shape is one dot. Throwing away the high bit in each byte, since it is only a color flag, we have seven bits with which to

Listing 11.1

```
1   REM  CLEAR SCREEN
2   HOME
4   REM  XC IS X-CHANGE, YC IS Y-CHANGE
5 XC = 1:YC = 1
9   REM  XO IS OLD X, YO IS OLD Y
10 X = 1:Y = 1:OX = 1:OY = 1
19  REM  ERASE
20  HTAB OX: VTAB OY: PRINT " ";
24  REM  PLOT
25  HTAB X: VTAB Y: PRINT "O";
29  REM  SAVE OLD COORDINATES
30 OX = X:OY = Y
34  REM  FIND NEW COORDINATES
35 X = X + XC:Y = Y + YC
40  IF X > 39 THEN X = 39:XC =  - 1
50  IF X < 1 THEN X = 1:XC = 1
60  IF Y > 24 THEN Y = 24:YC =  - 1
70  IF Y < 1 THEN Y = 1:YC = 1
80  GOTO 20
```

work, and our one dot shape might be stored as the byte in Figure 11.1. Now, to move it across the screen, it can be in any of the positions shown in Figure 11.2. (Figure 11.2 shows the bit that's set to 1 as a dot, and those not set as blank, which is what happens when they are displayed on the screen.) Note that when the byte is moved horizontally, our *dot shape* moves seven pixels at a time, rather than a more smooth one or two. (Aha! *Pixel,* a word we haven't yet used. Pixel is short for picture element, which is a fancy way of saying dot. Dot doesn't sound very scientific.) (Neither do I, since I usually use dot instead of pixel.) Note that vertical movement wouldn't have the same problem, since the bytes are only one dot tall when stacked on the screen. In other words, you could take Figure 11.2 and duplicate it immediately below itself and find the dot-shapes directly above and below each other. Our hi-res text generator would work better than the text screen animation in Listing 11.1 for vertical movement because of that.

How do you get smooth horizontal movement? Two ways. First, you could use the machine language ROL and ROR commands (rotate bits left and rotate bits right similar to ASL and LSR) to shift the shape into the correct column before putting the byte(s) on the screen. But that gets messy, since any shape wider than one pixel can overflow across two bytes, depending on exact screen position, and then you still have to worry about the high bit when you are ROLling and RORring things around. And worse yet, all that takes time, which you don't want to waste when animating. That's why the character animation was fast: minimal computation involved; just find the byte and put it on the screen.

Solution? *Pre*-shifted shapes. In other words, do the ROLling and RORring first, either by hand or by letting the computer help you. We'll do it by hand so you can follow along. Take that one-dot shape, for

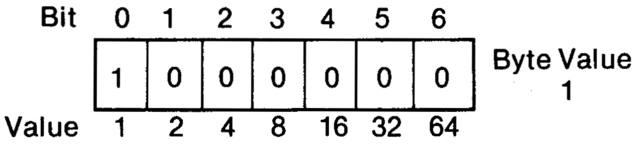


Figure 11.1 Byte for One-Dot Shape

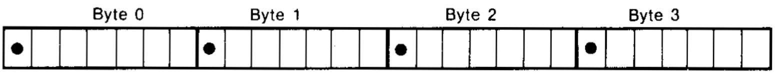


Figure 11.2 Movement of One-Dot Shape

example. We want that dot to be able to appear in any of seven positions in any byte. Hence, we store seven shapes—pre-shifted shapes. Take a look at Figure 11.3 to see how to get them. Notice that for any dot that you want set in Figure 11.2, you can find one of the pre-shifted shapes and store that byte's value in screen memory to set the dot.

Before going on, we'll do a crude pre-shifted shape plotter in Basic. Note that the program in Listing 11.2 has a byte counter (X) and bit counter (XB) for the X-value. That's because we are not only interested in putting a byte on the screen, we also want to be able to find the byte-value with the proper bit set. We'll store our table of seven pre-shifted,

Bit	0	1	2	3	4	5	6	
	1	0	0	0	0	0	0	Shift 0 Value 1
Value	1	2	4	8	16	32	64	
Bit	0	1	2	3	4	5	6	
	0	1	0	0	0		0	Shift 1 Value 2
Value	1	2	4	8	16	32	64	
Bit	0	1	2	3	4	5	6	
	0	0	1	0	0	0	0	Shift 2 Value 4
Value	1	2	4	8	16	32	64	
Bit	0	1	2	3	4	5	6	
	0	0	0	1	0	0	0	Shift 3 Value 8
Value	1	2	4	8	16	32	64	
Bit	0	1	2	3	4	5	6	
	0	0	0	0	1	0	0	Shift 4 Value 16
Value	1	2	4	8	16	32	64	
Bit	0	1	2	3	4	5	6	
	0	0	0	0	0	1	0	Shift 5 Value 32
Value	1	2	4	8	16	32	64	
Bit	0	1	2	3	4	5	6	
	0	0	0	0	0	0	1	Shift 6 Value 64
Value	1	2	4	8	16	32	64	

Figure 11.3 Seven Pre-Shifted Shapes

one-dot shapes in the array S%, so that S%(XB) will have the proper value for having the correct bit set. The program uses the machine language plot routine and the lookup table from Chapter 7. Note that we did not put an erase routine into the program, so our dot shape leaves a trail... most of the time. In the next chapter we'll do larger, animated shapes and a machine language routine to put them on the screen, and we'll look at a tricky technique for eliminating the erase cycle. The really clever ones among you might find the secret to that trick in the way Listing 11.2's program works; perhaps Listing 11.3, which is a variation on Listing 11.2, will give you a clue.

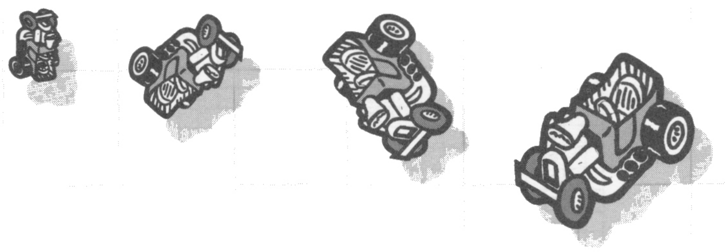
Listing 11.2

```
1  GOSUB 1000
4  REM  XC IS X-CHANGE, YC IS    Y-CHANGE
5 XC = 1:YC = 1
10 X = 0:XB = 0:Y = 0
19  REM  PLOT
20  POKE 24576,X: POKE 24577,Y: POKE 24592,S%
    (XB): CALL 24578
34  REM  FIND NEW COORDINATES
35 XB = XB + XC:Y = Y + YC
40  IF XB > 6 THEN XB = 0:X = X + XC
45  IF XB < 0 THEN XB = 6:X = X + XC
55  IF X > 39 THEN X = 39:XB = 6:XC =  - 1
60  IF X < 0 THEN X = 0:XB = 0:XC = 1
65  IF Y > 191 THEN Y = 191:YC =  - 1
70  IF Y < 0 THEN Y = 0:YC = 1
80  GOTO 20
999  REM  INITIALIZE
1000  PRINT  CHR$ (4);"BLOAD PLOT"
1010  PRINT  CHR$ (4);"BLOAD LOOKUP"
1020  HGR : POKE  - 16302,0
1029  REM  READ PRESHIFTED SHAPE DEFINITIONS,
      AS IN FIGURE 3
1030  DIM S%(6): FOR I = 0 TO 6: READ S%(I):
      NEXT : DATA  1,2,4,8,16,32,64
1040  RETURN
```

Listing 11.3

```
1  GOSUB 1000
5 XC = 1
10 X = 19:XB = 0:Y = 95
19  REM  PLOT
20  POKE 24576,X: POKE 24577,Y: POKE 24592,S%
    (XB): CALL 24578
34  REM  FIND NEW COORDINATES
35 XB = XB + XC
40  IF XB > 6 THEN XB = 5:XC =  - 1
45  IF XB < 0 THEN XB = 1:XC = 1
80  GOTO 20
999  REM  INITIALIZE
1000  PRINT  CHR$ (4);"BLOAD PLOT"
1010  PRINT  CHR$ (4);"BLOAD LOOKUP"
1020  HGR : POKE  - 16302,0
1029  REM  READ PRESHIFTED SHAPES AS IN FIGURE
      3
1030  DIM S%(6): FOR I = 0 TO 6: READ S%(I):
      NEXT : DATA  1,2,4,8,16,32,64
1040  RETURN
```

T W E L V E



Animation of Larger Pre-Shifted Shapes

Plotting pre-shifted shapes from Basic, as in last chapter, sort of defeats the purpose. Pre-shifted shapes are for speed. Basic isn't; at least not when it comes to graphics. So this chapter we'll look at building larger pre-shifted shapes (larger than the hi-res dot, last chapter) and voyage back into the world of machine language for some of the coding.

First, a Shape

We'll use a fairly easy one so that the actual number codes that we generate don't wreak havoc on the proofreaders and the poor typists. This is your chance to shine with creativity, however. Almost all the shapes in all the arcade games you play are created in this way, so make up the shape you want, plot it out, and watch it animate at the end of this chapter.

Last chapter we dropped the hint that there is a technique for creating and animating a shape that allows you to omit the erase cycle of the usual *draw-update-erase* animation. There is, and it speeds up your animation considerably. Since we're just putting bytes onto the screen, we can cause a shape to erase its old image at the time the new image is being plotted if we know the maximum single movement of the shape. That is, if we know that in any single move a shape will go no more than two dots in any direction, for example, we can create a 2-dot border the color of the background around the shape. Technically, the 2-dot border is part of the shape. Visibly, it looks like the background.

Figure 12.1 shows a box shape pre-shifted across 3 bytes. Note that although the box is less than two bytes wide, when the border is included

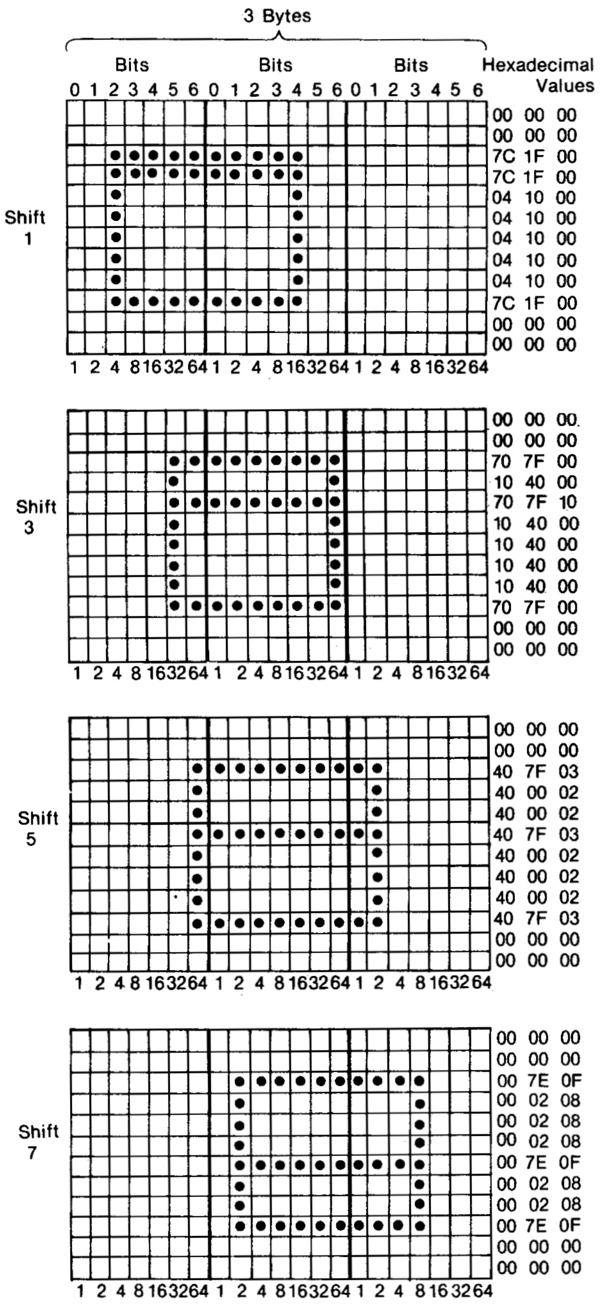
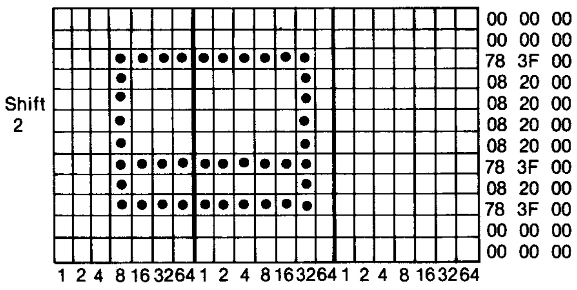
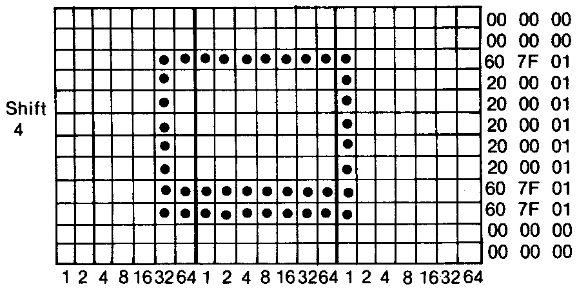
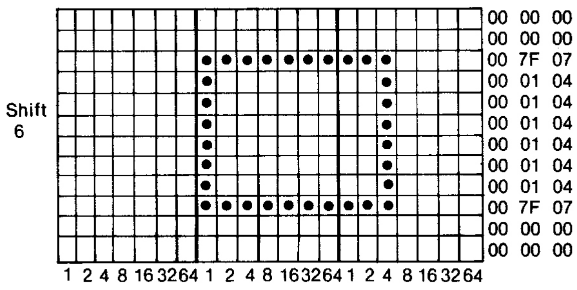


Figure 12.1 A Pre-Shifted Box Shape


Shift 2
00 00 00
00 00 00
78 3F 00
08 20 00
08 20 00
08 20 00
08 20 00
78 3F 00
08 20 00
78 3F 00
00 00 00
00 00 00
1 2 4 8 16 32 64 1 2 4 8 16 32 64 1 2 4 8 16 32 64


Shift 4
00 00 00
00 00 00
60 7F 01
20 00 01
20 00 01
20 00 01
20 00 01
20 00 01
60 7F 01
60 7F 01
00 00 00
00 00 00
1 2 4 8 16 32 64 1 2 4 8 16 32 64 1 2 4 8 16 32 64


Shift 6
00 00 00
00 00 00
00 7F 07
00 01 04
00 01 04
00 01 04
00 01 04
00 01 04
00 01 04
00 7F 07
00 00 00
00 00 00
1 2 4 8 16 32 64 1 2 4 8 16 32 64 1 2 4 8 16 32 64

and it is shifted through its seven possible positions relative to the byte boundaries, we need the third byte in width as part of the definition.

In defining the seven pre-shifted boxes, you'll notice that we also played some games with horizontal lines cutting through the middle of each box. Remember that when using pre-shifted shapes, the actual shape used depends on the horizontal position on the screen. We can take advantage of the actual shape definition displayed being swapped through the seven pre-shifts and modify each pre-shift slightly to cause the object to animate as it moves! A man walking can have his legs moving as he moves across the screen, helicopter blades can swirl, lights and radar on a spaceship can blink and turn, and on and on. In a way, it's double animation, with the shape you want moving around the screen smoothly, and meanwhile the shape itself animates within.

Moving in Twos

Okay, but that doesn't explain why those lines in the boxes seem oddly out of sequence. The intention is for the line to scroll vertically inside the box while the box moves. Shouldn't the line move in smaller increments between frame one and frame two, and so on? Well, it does. We're going to animate the box in 2-dot moves. Besides moving faster around the screen, there's a very good reason for moves in multiples of two. Odd number movements (along the horizontal) mess up any colors you might have. Bits that were in even columns (blue or violet) get put in odd columns (orange or green), and vice versa. It is possible for your shape definition to contain every Apple color, including blended colors like we used in the fill routines. So although our box is white, it's usually favorable to move in 2s for color presentation. The result is that instead of the shapes animating in a 1, 2, 3, 4, 5, 6, 7 sequence, they animate in a 1, 3, 5, 7, 2, 4, 6 sequence. If you look at the shapes in Figure 12.1 again, you'll see that in this order the little line inside the box moves in much shorter steps.

The hexadecimal numbers defined by the shape are shown alongside the shape itself. For our routine, we also need the width and height of the shape in bytes (3 by 12), and for speed, we'll also store offsets that point to the start of each individual shape in the table of seven that we'll produce. Here's the format of the table:

First 14 bytes (Starting address + 0 through starting address + 13): two-byte offsets for each of the 7 shapes, telling what to add to the starting address of the table to find the address of each individual shape.

Bytes 14 and 15: height and width of the shape, in bytes.
Starting at byte 16 (starting address + 16): Row by row byte values for each of the seven shapes.

The first 16 bytes of the table generated by our shape are shown in Figure 12.2. Note that the offset to the first shape (the first two bytes) is always 16 (since the first shape always starts at starting address + 16). 16 is 00 10 in two-byte hexadecimal. Since the width and height are 3 bytes and 12 bytes, 3 times 12 gives the length of each individual shape in bytes, and we can find the other offsets by adding this length to each previous offset. The second offset is 16 + 36, or 52, or 00 34 in hexadecimal. The third is 52 + 36, or 88, and so on. If hexadecimal makes your stomach queasy, the short program in Listing 12.1 will compute and poke in the first 16 values for any shape you make.

First offset	00 10
Second offset	00 34
Third offset	00 58
Fourth offset	00 7C
Fifth offset	00 A0
Sixth offset	00 C4
Seventh offset	00 E8
Width, height	03 0C

Figure 12.2 A Table Index

Listing 12.1

```
10   HOME : INPUT "WIDTH : ";W
20   INPUT "HEIGHT : ";H
30 L = W * H
40 F = 16
50   FOR I = 0 TO 12 STEP 2
59   REM  QUOTIENT OF DIV. BY 256
60   POKE 32768 + I, INT (F / 256)
69   REM  REMAINDER OF DIV. BY 256
70   POKE 32769 + I,F -  INT (F / 256) * 256
80 F = F + L
90   NEXT I
100   POKE 32782,W: POKE 32783,H
```

Entering the Shape Data

After the first 16 bytes, you have to enter the shape data itself. Following the steps in Figure 12.3, you can enter the data directly into memory. Press Return after each line; the first character in each line (] and *) is a prompt which you don't type. Note by the address that we started the table at address $8000 hex, or 32768. The last step is to BSAVE your shape with the command:

```
BSAVE BOX,A$8000,L268
```

The length is computed by taking the shape length, times 7 for the pre-shifts, then adding 16 (for the first 16 bytes). Or, just given the width and height of the shape (in bytes):

Width * Height * 7 + 16

All of this, by the way, is much more easily done through editors that do all the calculations and set-up of shapes. Diehards still may enter a lot of the values directly in hexadecimal, but there are utilities on the market that to do a lot of this work for you.

The machine language shape-plotting routine is given in Listing 12.2. It is similar in function to the one in Basic last chapter, except it allows shapes that are more than one byte long. It is set up so that you tell it the X and Y coordinates at which you want to plot the shape by poking those values into locations 4 and 2, respectively. The first thing the routine does (lines 14-25) is a quick "division" by 7 to determine the X-bit and X-byte that the X location you used corresponds to. The division just uses subtraction until the result is smaller than seven. The number of subtractions (counted in the Y-register) is the quotient, giving the X-byte, and the number left over (in the accumulator) is the remainder, or the X-bit. From there the routine pulls the height and width out of the table, then using two times the X-bit value, finds the location of the offset values for the desired shape. The offset is added to the starting address of the table, giving the actual location of the start of the shape definition being used. That address is then inserted into the machine code (at BLOCK1 + 1 and BLOCK1 + 2). This is called *self modifying code,* where the program actually rewrites part of itself; in this case an address. The advantage gained here is in speed. Sometimes the technique is also used to save programming code. The rest of the routine is a pair of loops that put each row of bytes onto the screen while counting down the screen vertically. We use the good old Y-lookup table for finding the starting address of each screen line.

```
]CALL-151

*8000:00 10 00 34 00 58 00 7C
*8008:00 A0 00 C4 00 E8 03 0C

*8010:00 00 00 00 00 00 7C 1F
*8018:00 7C 1F 00 04 10 00 04
*8020:10 00 04 10 00 04 10 00
*8028:04 10 00 7C 1F 00 00 00
*8030:00 00 00 00 00 00 00 00
*8038:00 00 78 3F 00 08 20 00
*8040:08 20 00 08 20 00 08 20
*8048:00 78 3F 00 08 20 00 78
*8050:3F 00 00 00 00 00 00 00

*8058:00 00 00 00 00 00 70 7F
*8060:00 10 40 00 70 7F 00 10
*8068:40 00 10 40 00 10 40 00
*8070:10 40 00 70 7F 00 00 00
*8078:00 00 00 00 00 00 00 00
*8080:00 00 60 7F 01 20 00 01
*8088:20 00 01 20 00 01 20 00
*8090:01 20 00 01 60 7F 01 60
*8098:7F 01 00 00 00 00 00 00

*80A0:00 00 00 00 00 00 40 7F
*80A8:03 40 00 02 40 00 02 40
*80B0:7F 03 40 00 02 40 00 02
*80B8:40 00 02 40 7F 03 00 00
*80C0:00 00 00 00 00 00 00 00
*80C8:00 00 00 7F 07 00 01 04
*80D0:00 01 04 00 01 04 00 01
*80D8:04 00 01 04 00 01 04 00
*80E0:7F 07 00 00 00 00 00 00

*80E8:00 00 00 00 00 00 00 7E
*80F0:0F 00 02 08 00 02 08 00
*80F8:02 08 00 7E 0F 00 02 08
*8100:00 02 08 00 7E 0F 00 00
*8108:00 00 00 00

]3D0G
```

Figure 12.3 Entering the Shape Table

Listing 12.2

```
                       ORG   $6000
             XBYTE     EQU   0
             XBIT      EQU   1
             YLOC      EQU   2
             XCOUNT    EQU   3
             SCX       EQU   $4
             ;SCREEN X
             SLO       EQU   $06
             ;SCREEN LINE ADDRESS
             SHI       EQU   $07
             WIDTH     EQU   $08
             HEIGHT    EQU   $09
             THI       EQU   $7000
             ;Y LOOKUP TABLE
             TLO       EQU   $70C0
             SHAPE     EQU   $8000
             ;SHAPE LOCATION
6000: A0 00  START     LDY   #0
6002: A5 04            LDA   SCX
6004: 18               CLC
6005: C9 07  DLOOP     CMP   #7
      ;THIS IS A CHEAP DIVISION
6007: 90 08            BCC   DDONE
      ;THAT CAN BE AVOIDED
6009: 38               SEC
600A: E9 07            SBC   #7
600C: C8               INY
600D: 18               CLC
600E: 4C 05 60         JMP   DLOOP
6011: 85 01  DDONE     STA   XBIT
      ;DIVISION DONE
6013: 84 00            STY   XBYTE
6015: A0 0E            LDY   #$0E
      ;GET HEIGHT AND WIDTH
6017: B9 00 80         LDA   SHAPE,Y
601A: 85 08            STA   WIDTH
601C: C8               INY
601D: B9 00 80         LDA   SHAPE,Y
6020: 85 09            STA   HEIGHT
```

Listing 12.2 (continued)

```
6022: A5 01                 LDA  XBIT
      ;GET INDIVIDUAL SHAPE
6024: 0A                    ASL
      ;OFFSET AND ADD TO START
6025: A8                    TAY
      ;OF SHAPE"S TABLE
6026: B9 00 80              LDA  SHAPE,Y
6029: AA                    TAX
602A: C8                    INY
602B: B9 00 80              LDA  SHAPE,Y
602E: 18                    CLC
602F: 69 00                 ADC  #<SHAPE
6031: 8D 5B 60              STA  BLOCK1+1
      ;MODIFY CODE FOR SPEED
6034: 8A                    TXA
6035: 69 80                 ADC  #>SHAPE
6037: 8D 5C 60              STA  BLOCK1+2
603A: A2 00                 LDX  #0
      ;TO START SHAPE
603C: A5 08      LOOP1      LDA  WIDTH
      ;# OF HOR. BYTES
603E: 85 03                 STA  XCOUNT
6040: A4 02                 LDY  YLOC
6042: C0 C0                 CPY  #$C0
      ;CHECK IF OFF SCREEN
6044: 90 04                 BCC  YTABLE
6046: A0 29                 LDY  #$29
      ;OFF SCREEN, END LINE
6048: B0 0C                 BCS  LOOP2
604A: B9 C0 70   YTABLE     LDA  TLO,Y
604D: 85 06                 STA  SLO
604F: B9 00 70              LDA  THI,Y
6052: 85 07                 STA  SHI
6054: A4 00                 LDY  XBYTE
6056: C0 28      LOOP2      CPY  #$28
      ;CHECK IF OFF SCREEN
6058: B0 05                 BCS  CONT
605A: BD 00 80   BLOCK1     LDA  SHAPE,X
      ;THIS CODE GETS MODIFIED
```

Listing 12.2 (continued)

```
605D: 91 06       61                STA   (SLO),Y
      ;PUT IT ON SCREEN
605F: E8          62    CONT        INX
      ;NEXT SHAPE BYTE
6060: C8          63                INY
      ;NEXT SCREEN LOCATION
6061: C6 03       64                DEC   XCOUNT
      ;KEEP TRACK OF WIDTH
6063: D0 F1       65                BNE   LOOP2
6065: E6 02       66                INC   YLOC
      ;NEXT Y
6067: C6 09       67                DEC   HEIGHT
      ;KEEP TRACK OF HEIGHT
6069: D0 D1       68                BNE   LOOP1
606B: 60          69                RTS
```

Putting It All Together

The last listing puts all of the above together. Listing 12.3 is a Basic program that loads the Shape Plot routine, the Y-lookup table (at $7000), and the box shape we created (or any other that you design). Line 50 pokes in the X and Y coordinates and calls the routine. 60 reads the paddles, and 70 and 80 compute the new X and Y values. Note that the weird expressions in the latter two lines use an interesting little fact: an expression such as (XP<80) generates a value of 1 if true, or 0 if false.

Listing 12.3

```
10  PRINT  CHR$ (4);"BLOAD LISTING 12.2"
20  PRINT  CHR$ (4);"BLOAD LOOKUP,A$7000"
30  PRINT  CHR$ (4);"BLOAD BOX"
40 X = 128:Y = 128: HGR : POKE  - 16302,0
50  POKE 2,Y: POKE 4,X: CALL 24576
60 XP =  PDL (0):YP =  PDL (1)
70 X = X - 2 * (XP < 80) + 2 * (XP > 160)
80 Y = Y - 2 * (YP < 80) + 2 * (YP > 160)
90  GOTO 50
```

You can use that 0 or 1 in a computation! In line 70, for example, it evaluates as follows:

if XP<80, (joystick to the left):

```
  X - 2*1 + 2*0, which is X-2
```

if XP is between 80 and 160, (joystick in middle):

```
  X - 2*0 + 2*0, which is X
```

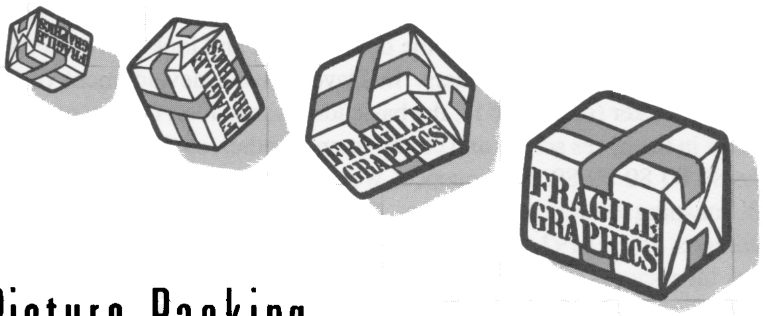
and if XP>160, (joystick to the right):

```
  X - 2*0 + 2*1, which is X+2
```

Neat, huh?

One last note: the Basic routine is written to accept values between 0 and 255 for X and Y. That means that if you let the shape go too far to the right, or up to the top of the screen, you'll get an error—the program just stops. Once again, Basic becomes cumbersome. The reason is that poke statements accept values only from 0 to 255 (the values that fit in a byte). IF statements and value checks have been left out to give an idea of how fast you can get the animation going from Basic. Unfortunately, the more Basic statements you add, the more you slow down the animation. An alternative is to perform the movement calculations in machine language, or to have the movements pre-generated in a path table. Either way you can easily increase the speed enough to be driven from Basic, although you always have to be careful about how much code you put into an animation loop.

THIRTEEN



Picture Packing

When dealing with graphics on the Apple, or any other computer for that matter, one of the limiting factors is the great amount of storage that graphic information requires. On the Apple, a high resolution picture in RAM takes almost one-fourth of the available programming memory (8K of approximately 36K, after DOS and other scratch areas are subtracted). On an Apple disk, you can typically fit only 12 hi-res screen pictures. Although there's not much to do about the amount of display RAM required, there are ways to scrunch more pictures onto disks.

The Original Recipe

As far as I know, Dave Lubar wrote the first picture packing routines for the Apple a few years ago. Since then several others have been written, mostly spinoffs and modifications of Lubar's original routines. What's a picture packer? It's a program that takes a standard picture, stored in its full, glorious 8,192 bytes of memory, and looks for patterns that allow condensing of the information. (The implication is that one also needs a picture unpacker that will take the packed picture and put it back the way it was.) Simple? Ah, but how does one look for the patterns? Which patterns pack the most efficiently?

One of the best packing routines to date is one written by Dav Holle (you may recognize the name as that of the author of *Zoom Grafix,* coauthor in charge of graphics and various and sundry other details in *Sherwood Forest,* and for the very astute, the *Pie Man* cartoonist). As Holle puts it, his routines, as well as most others, are basically variations on Lubar's original. Simply evolution at work.

Variations on a Theme

The easiest way to explain reasons some packing routines are more efficient than others is to trace the development and look at why certain techniques improved on others. In Dave Lubar's development of the original packer (method #1), the basic idea was to look for any repeated values in the hi-res screen and clump them together, so that screen values like 80 80 80 80 80, in sequence, would generate the packed code 05 80, meaning 80 repeated 5 times. Since the screen takes the memory addresses from 8192 to 16383 ($2000-3FFF in hexadecimal), we can pack the pictures sequentially in RAM rather than worrying about where on the screen the values are displayed. We do know that sequential bytes are displayed next to each other horizontally on the screen, so, for example, if the top third of the screen was black, each of the horizontal lines in that area would pack nicely.

Problem #1 arises from the fact that colors other than black or white don't create a byte pattern that repeats every byte. As we discovered in Chapter 9, the colors that have every other bit set require one value in even bytes and a different value in odd bytes. See Figure 13.1 for a refresher on the type of pattern for these colors. The result is that colors other than black or white wouldn't pack at all using method #1. Solution #1 is to have the program try packing twice, once checking every byte sequentially, and the second time trying every other byte for patterns and zipping through the screen twice, once for all even bytes, and again for all odd bytes. The two trials could then be compared to see which was more efficient.

That works better, but all the single, unique bytes still cause a problem. Since we're using pairs of bytes, one to tell how many repetitions and another to tell what repeats, patterns such as 55 55 55 55 55 00 00 00 42 42 42 42 pack nicely into 05 55 03 00 04 42. Unfortunately, patterns such as 55 00 46 93 FF A8 become 01 55 01 00 01 46 00 93 00 FF 00 A8. Not much savings there.

A close representation of Lubar's final form (close because a few minor changes have been omitted here for clarity's sake) is to have the

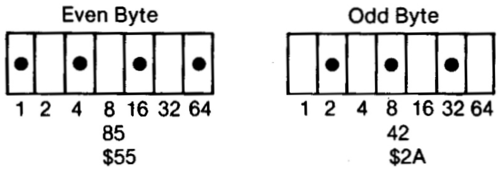


Figure 13.1 Pattern for Violet

first byte in the pair count up to 127 repetitions (call this number N). 127 can be represented in 7 bits. The eighth bit tells whether the following are N unique bytes, or one byte repeated N times. Using this technique, a pattern such as 85 79 A2 55 55 55 55 00 00 34 21 would be packed as (83) 85 79 A2 (04) 55 (02) 00 (82) 34 21. The bytes that give the repetitions are in parentheses. Note that the 8 in the first position tells you that the high bit is set, so in the example we have 3 unique bytes first, then 4 repetitions of 55 followed by 2 repetitions of 00, and then 2 more unique bytes.

Dramatic Improvement

Well, this was really exciting because suddenly we were able to get 25 to 40 pictures on a disk, sometimes even more. Pictures that used to take 34 sectors of disk storage now took somewhere between 7 and 25 sectors, in most cases. But alas, there are always better ways. Analyzing a sample hi-res screen will show what the next step was, as stumbled upon by a few observers. In Figure 13.2, note how someone in the art department at Softalk has tried to faithfully reproduce a quick sketch of the hi-res screen. (After all the fun little pre-shifted shape illustrations last chapter with dots and tiny little numbers all over the place, they

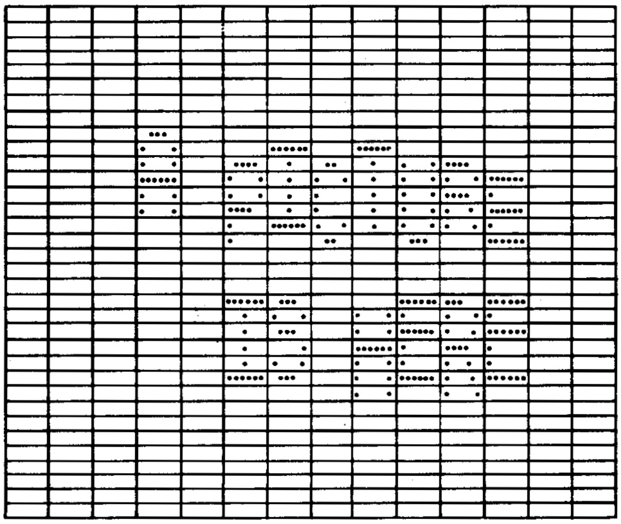


Figure 13.2 A Hypothetical Hi-Res Picture

must really like me. Fortunately, I'm 2,000 miles away. Back in the old days they used to chase me out of the art department with their T-squares sharpened, baring their teeth and screaming when I came in with all these goofy computer diagrams—at the last minute, of course.) Anyway, in the figure, notice that the little lines which are supposed to divide up the picture into bytes, approximately, are very close as you move down the screen and not so close moving across the screen. Bytes, when displayed, are one dot tall and seven dots wide. Okay, big deal. Well, look at the illustration and figure out approximately what percent of the lines across the screen go the full 40 bytes uninterrupted by the writing. Then scan the area vertically and figure approximately what percent of the area can be covered by patches of repetition one byte wide and at least the height of the lettering (which proportionately would be about 40 bytes tall on the hi-res screen). The answer is that you are more likely to find more repetitions of a pattern when scanning vertically instead of horizontally. When scanning horizontally, you have to go uninterrupted all the way across the screen to get only 40 repeating bytes. Scanning vertically, you are more likely to find repeating bytes because they are all in the same general area of the picture. In fact, as this picture is drawn, you would have over 400 bytes repeating on the left side before you ran into the letter A, and even then, the space between the words (and even letters such as H) would contain a good amount of repetition.

So the trick is that scanning according to the screen instead of in memory sequence *does* make a difference, and it's on this concept that Dav Holle based his packing routine. Holle did make a few other modifications to improve efficiency. The first is similar to modification #1 to Lubar's routine. As we found in experimenting with color combinations, every color routine that creates more than the six Apple colors used a two-line pattern. Remember when we made the odd horizontal lines orange and the even ones green and came out with a simulated yellow? That yellow, and every other blended color, would easily defeat the above idea for vertical packing unless it was done in two passes. The first pass does the odd lines and the second pass does the even lines.

Holle's second modification was to eliminate the necessity for repeat factors before unique values. To do so, he used a little trickery. For each repeated section he uses three bytes: a 00 to signal repeated values, then a number 1-255 to tell how many repetitions, and then the repeated value. Unique bytes are just put directly into the table. When the unpacker finds a zero, it knows that something following will repeat. What about screen values of 0? Holle just changed them to 80. 00 is black with the high bit off, and 80 is black with the high bit on, but they both look the same on the screen. Actually, it turns out that the major savings in this approach is that you can have repetitions up to 255 times instead of Lubar's 127, and as often as that occurs, it probably doesn't matter much. Technically, it does eliminate one possibility in Lubar's: in his, it is actually possible for the packed picture to take more space than the original. This occurs if few or none of the values repeat; since there are flags every 128 bytes that say that the following N bytes are unique, those extra bytes would make the picture longer. Practically, I've yet to see that happen, and Holle's 3 bytes for repetitions equally balance Lubar's 1 for non-repeating values.

The Packing Program

So what we have following is Dav Holle's variation on Dave Lubar's original packing ideas, slightly modified by Mark Pelczarski (had to get my two cents worth in *somewhere*). The routine as listed is assembled at $6000, decimal 24576, just above hi-res page 2. It is 248 bytes long, and contains both the packer and unpacker. There are three locations to set before calling either routine.

Poke the starting address of your packed picture table in locations 0 and 1. This is the address where you want the packer to put the packed information, or the address where you BLOADed a packed picture file. The address is stored in low/high format, which means that if your address is A, you'd use the following pokes (in line 10 we give A a

Listing 13.1

```
5  HGR
10  PRINT  CHR$ (4);"BLOAD PICTURE.PIC"
20  POKE 0,0: POKE 1,97: POKE 230,32
25  REM  The pokes in 0 and 1 give the
    address 24832 in shorter form.
30  CALL 24700
40 L =  PEEK (0) +  PEEK (1) * 256 - 24831
45  REM  L is the length of the resulting
    packed picture.
50  PRINT L
60  PRINT  CHR$ (4);"BSAVE PICTURE.PAC,A";
    24831; ",L";L
```

value of 24832, which is as good a location as any, since it's right above the pack/unpack routines):

```
10    A = 24832
20    POKE 0, A - INT (A / 256) * 256
30    POKE 1, INT(A / 256)
```

In location 230 ($E6), poke the value corresponding to the hi-res screen you want packed, or onto which you want the packed picture unpacked. POKE 230,32 for page 1, and POKE 230,64 for page 2. Note that this location in most operations also tells the computer which hi-res screen should be drawn upon, which gives you a nifty trick for displaying one page while drawing on the invisible one.

After the above pokes, CALL 24576 if you want to unpack a picture, or CALL 24700 if you want a picture packed. After return from the routine, locations 0 and 1 hold the last address in the packed table. Listings 13.2 and 13.3 are examples of using the packer and unpacker, respectively.

Note in the program listings that we used .PIC at the end of the name to denote a picture in standard format, and .PAC to denote a packed picture file. When playing with graphics you tend to get a lot of different types of binary files on your disks. Using something in the name to designate what type of file it is helps determine if something called FROG is a picture, a packed picture, a pre-shifted shape, an Applesoft shape, or that new machine language routine you developed.

Also, for those of you who want to tinker in machine language and need to relocate the packing routine, there are only six places that need changes, all referring to location $605A or $60EF.

Listing 13.2

```
5  HGR
10  PRINT  CHR$ (4);"BLOAD PICTURE.PAC"
20  POKE 0,0: POKE 1,97: POKE 230,32
30  CALL 24576
```

Listing 13.3

```
10 D$ =  CHR$ (4):PACK = 24700:UNPACK = 24576:
   LOC = 24831
20  PRINT D$;"BLOAD LISTING 13.4"
30  HOME : HTAB 10: PRINT "PACKER/UNPACKER
    UTILITY"
40  PRINT : PRINT "PACK OR UNPACK (P/U)?";
```

Listing 13.3 (continued)

```
50   GET P$: IF P$ <  > "P" AND P$ <  > "U"
     THEN 50
60   PRINT P$
70   PRINT : PRINT "HI-RES SCREEN (1/2)?";
80   GET S$: IF S$ <  > "1" AND S$ <  > "2"
     THEN 80
90   PRINT S$
100   PRINT : INPUT "FILENAME: ";F$
110   PRINT : PRINT : PRINT "OKAY (Y/N)? ";
120   GET A$: IF A$ <  > "Y" AND A$ <  > "N"
      THEN 120
130   IF A$ = "N" THEN 30
140   POKE  - 16304,0: POKE  - 16297,0: POKE  -
      16302,0: POKE 230,32
150   IF S$ = "2" THEN  POKE  - 16299,0: POKE
      230,64
160   IF P$ = "P" THEN 200
170   GOTO 300
200   REM  Pack and Save
210   POKE 0,LOC - ( INT (LOC / 256) * 256):
      POKE 1, INT (LOC / 256)
220   CALL PACK
230 LN =  PEEK (0) +  PEEK (1) * 256 - LOC:
     REM  Length
240   PRINT : PRINT D$;"BSAVE";F$;".PAC,A";LOC;
      ",L";LN
250   GOTO 400
300   REM  Load and Unpack
310   POKE 0,LOC - ( INT (LOC / 256) * 256):
      POKE 1, INT (LOC / 256)
320   PRINT : PRINT D$;"BLOAD ";F$;".PAC,A";LOC
330   CALL UNPACK
400   REM  Done, what's next?
410   POKE  - 16368,0: GET A$
420   TEXT : HOME : VTAB 10: PRINT "QUIT
      OR CONTINUE (Q/C)?";
430   GET C$: IF C$ <  > "Q" AND C$ <  > "C"
      THEN 430
440   IF C$ = "C" THEN 30
450   HOME : END
```

Listing 13.4

```
POINTS TO PACKED DATA
                                     ORG   $6000
                        LINES        EQU   192
                   ;160 OR 192 LINES
                        TBL          EQU   0
                   ;POINTS TO PACKED DATA
                        XCOORD       EQU   2
                        YCOORD       EQU   3
                        OFFSET       EQU   4
                        GBAS         EQU   5
                        PREV         EQU   7
                   ;PREVIOUS DATA
                        REPEAT       EQU   8
                   ;REPEAT COUNT
6000:   A5 E6           UNPACK       LDA   $E6
6002:   09 04                        ORA   #$04
6004:   85 06                        STA   GBAS+1
                   ;$2400 (X=0,Y=1)
6006:   A2 01                        LDX   #1
6008:   86 04                        STX   OFFSET
600A:   A0 00                        LDY   #0
600C:   84 02                        STY   XCOORD
600E:   84 05                        STY   GBAS
6010:   84 08                        STY   REPEAT
6012:   B1 00           PROC         LDA   (TBL),Y
                   ;GET DATA
6014:   D0 18                        BNE   STORE
                   ;SIMPLE DATA
6016:   E6 00                        INC   TBL
6018:   D0 02                        BNE   GETREP
601A:   E6 01                        INC   TBL+1
601C:   B1 00           GETREP       LDA   (TBL),Y
                   ;GET COUNT
601E:   85 08                        STA   REPEAT
6020:   E6 00                        INC   TBL
6022:   D0 02                        BNE   GETPREV
6024:   E6 01                        INC   TBL+1
6026:   B1 00           GETPREV      LDA   (TBL),Y
                   ;GET DATA
6028:   85 07                        STA   PREV
```

Listing 13.4 (continued)

```
602A:  A5 07       DOREP   LDA  PREV
602C:  C6 08               DEC  REPEAT
602E:  A4 02       STORE   LDY  XCOORD
6030:  91 05               STA  (GBAS),Y
6032:  E8                  INX
6033:  E8                  INX
6034:  E0 C0               CPX  #LINES
               ;OFF BOTTOM?
6036:  90 12               BCC  YOK
               ; NO
6038:  E6 02               INC  XCOORD
603A:  A4 02               LDY  XCOORD
603C:  C0 28               CPY  #40
               ;END OF PASS?
603E:  90 08               BCC  XOK
               ; NO
6040:  C6 04               DEC  OFFSET
               ;MORE PASSES?
6042:  30 15               BMI  DONE
               ; NO
6044:  A0 00               LDY  #0
6046:  84 02               STY  XCOORD
6048:  A6 04       XOK     LDX  OFFSET
604A:  20 5A 60    YOK     JSR  BASEC
604D:  A4 08               LDY  REPEAT
               ;REPEATING?
604F:  D0 D9               BNE  DOREP
               ; YES
6051:  E6 00               INC  TBL
6053:  D0 BD               BNE  PROC
6055:  E6 01               INC  TBL+1
6057:  D0 B9               BNE  PROC
               ;ALWAYS
6059:  60          DONE    RTS
605A:  8A          BASEC   TXA
               ;CALC BASE ADDR
605B:  29 C0               AND  #$C0
605D:  85 05               STA  GBAS
605F:  4A                  LSR
6060:  4A                  LSR
```

Listing 13.4 (continued)

```
6061:   05 05                   ORA   GBAS
6063:   85 05                   STA   GBAS
6065:   8A                      TXA
6066:   85 06                   STA   GBAS+1
6068:   0A                      ASL
6069:   0A                      ASL
606A:   0A                      ASL
606B:   26 06                   ROL   GBAS+1
606D:   0A                      ASL
606E:   26 06                   ROL   GBAS+1
6070:   0A                      ASL
6071:   66 05                   ROR   GBAS
6073:   A5 06                   LDA   GBAS+1
6075:   29 1F                   AND   #$1F
6077:   05 E6                   ORA   $E6
6079:   85 06                   STA   GBAS+1
607B:   60                      RTS
607C:   A0 01          PACK     LDY   #1
607E:   84 04                   STY   OFFSET
6080:   84 03                   STY   YCOORD
6082:   88                      DEY
6083:   84 02                   STY   XCOORD
6085:   A5 E6                   LDA   $E6
6087:   09 04                   ORA   #4
6089:   85 06                   STA   GBAS+1
608B:   84 05                   STY   GBAS
608D:   B1 05                   LDA   (GBAS),Y
608F:   D0 02                   BNE   MORE
6091:   09 80                   ORA   #$80
6093:   A2 01          MORE     LDX   #1
6095:   86 08                   STX   REPEAT
               ;COUNT=1
6097:   85 07                   STA   PREV
6099:   A4 02          TRAVEL   LDY   XCOORD
609B:   A6 03                   LDX   YCOORD
609D:   E8                      INX
609E:   E8                      INX
609F:   E0 C0                   CPX   #LINES
               ;OFF BOTTOM?
```

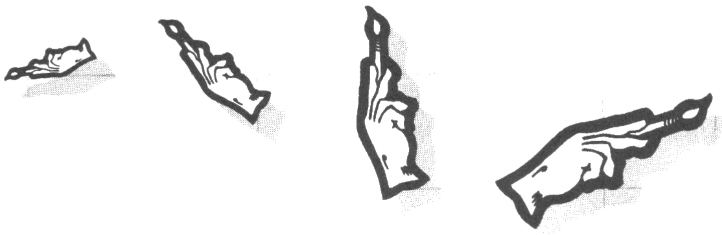
Listing 13.4 (continued)

```
60A1:   90 0                           BCC   YOL
                 ; NO
60A3:   C8                             INY
                 ;BUMP XCOORD
60A4:   C0 28                          CPY   #40
                 ;OFF RIGHT?
60A6:   90 06                          BCC   XOL
                 ; NO
60A8:   C6 04                          DEC   OFFSET
                 ;DONE?
60AA:   30 1B                          BMI   NOTEQ
                 ; YES, FINISH UP
60AC:   A0 00                          LDY   #0
60AE:   84 02          XOL             STY   XCOORD
60B0:   A6 04                          LDX   OFFSET
60B2:   86 03          YOL             STX   YCOORD
60B4:   20 5A 60                       JSR
                 BASEC
60B7:   B1 05                          LDA   (GBAS),Y
                 ;GET FROM SCREEN
60B9:   D0 02                          BNE   PROC2
60BB:   09 80                          ORA   #$80
                 ;$00 --> $80
60BD:   C5 07          PROC2           CMP   PREV
                 ;SAME?
60BF:   D0 06                          BNE   NOTEQ
                 ; NOPE
60C1:   E6 08                          INC   REPEAT
60C3:   D0 D4                          BNE   TRAVEL
60C5:   C6 08                          DEC   REPEAT

                 ;
60C7:   48             NOTEQ           PHA
                 ;SAVE NEW DATA
60C8:   A0 00                          LDY   #0
60CA:   A6 08                          LDX   REPEAT
60CC:   F0 0E                          BEQ   BIG
60CE:   E0 04                          CPX   #4
                 ;BIG ENUF TO USE CODE?
```

Listing 13.4 (continued)

```
60D0:  B0 0A                         BGE  BIG
60D2:  A5 07                         LDA  PREV
60D4:  20 EF 60       LITTLE         JSR  OUT
60D7:  CA                            DEX
60D8:  D0 FA                         BNE  LITTLE
60DA:  F0 0D                         BEQ  FRESH
                ;
60DC:  98             BIG            TYA
                ;REPEAT CODE=0
60DD:  20 EF 60                      JSR
                OUT
60E0:  8A                            TXA
                ;REPEAT COUNT
60E1:  20 EF 60                      JSR
                OUT
60E4:  A5 07                         LDA  PREV
                ;REPEAT DATA
60E6:  20 EF 60                      JSR  OUT
                ;
60E9:  68             FRESH          PLA
                ;GET NEW DATA
60EA:  24 04                         BIT  OFFSET
                ;DONE?
60EC:  10 A5                         BPL  MORE
                ; NO
60EE:  60                            RTS
                ;
60EF:  91 00          OUT            STA  (TBL),Y
                ;STORE IN TABLE
60F1:  E6 00                         INC  TBL
60F3:  D0 02                         BNE  DONE2
60F5:  E6 01                         INC  TBL+1
60F7:  60             DONE2          RTS
```

FOURTEEN



Packing by Saving Artist's Moves

While the method of packing described in Chapter 13 can easily quadruple the number of pictures you can fit on a disk, it is not the solution for applications that require more than, say, 50 pictures per disk. The type of program that immediately comes to mind is the adventure game, where you want to be able to display views of dozens to hundreds of locations easily. Any kind of program that requires a large amount of graphic information, though, requires even better packing techniques. (On the other hand, *Sherwood Forest,* an adventure game for which Dav Holle did the graphics, does use the packing routine presented last time). When Ken and Roberta Williams wrote the first graphic adventure, *Mystery House* (with line drawings), and followed with *Wizard and the Princess* (adding color), they used an interesting technique: they didn't store the pictures at all, just the information needed to reassemble them. Why store the pictures if you can tell the computer how to draw them, especially if telling the computer how to draw them requires much less information (in bytes)? Okay, so we tell the computer how to draw the picture. Maybe that saves some space.

The Graphics Magician

Using a product that's a little close to home, because there's no other utility that does exactly this, we'll look at part of *The Graphics Magician.* In *The Graphics Magician* there is a picture-drawing utility, which in many ways is similar to others around. With it you can use paddles, joystick, or a graphics tablet to draw lines, fill areas with color, or draw with a set of "paintbrushes." The difference is, it doesn't just show what you've drawn on the screen, it saves what you do: the artist's moves. It takes these moves and, unbeknownst to the artist, puts them into a little program. The program is saved into a binary file, and to reconstruct the drawing a special interpreter called PICDRAW is used. PICDRAW reads through the binary program and recreates the artist's moves at the speed of the computer, reassembling the picture right before your eyes, just as the artist had originally drawn it.

Drawing lines

How's it done? We'll take a simplified example of creating a picture with just lines and a fill routine (omitting some of the options in *Magician* for easier explanation). Start with the artist's four possible actions: (1) set the starting point of a line, (2) draw a line from the starting point to a given endpoint, (3) choose a color for filling, and (4) fill an area with color. Give each of those actions an operation code (or *opcode,* as it's called in computerese). We'll actually add one more operation, *end of picture,* for our use.

Opcode needed	*Action*	*Information needed*	*Bytes needed*
0	Picture's finished	None	1
1	Start a line	X,Y location to start at	3
2	Draw a line	X,Y location to draw to	3
3	Choose fill color	Color number	2
4	Fill an area	X,Y location of point in area	3

If you're starting to get the idea, what we're doing is writing our own computer language. In a way it's much like Basic, except instead of writing programs by editing lines of code, you draw using a joystick or paddles and the picture editor saves the appropriate opcodes and data.

Here's a sample "program," with the actual information on the left and description on the right:

01	10	20	Start line at 10,20
02	20	20	Draw line to 20,20

02	20	30	Draw line to 20,30
02	10	30	Draw line to 10,30
02	10	20	Draw line to 10,20
03	05		Set fill color to 5
04	15	25	Fill at 15,25
00			End of Picture

The first five instructions draw a square on the screen. Then the next two instructions fill the square with color #5 (for conversation, let's say color 5 is orange). The result is an orange square on the screen. If that's all we want in the picture, we've now compacted a hi-res screen from 8192 bytes to 21! Of course, the more you draw, the more space it takes, but it's not unreasonable to get nice, detailed pictures in hundreds of bytes, not thousands.

The "program" above is what the computer would see. You, I, or the artist would just see the results. To create the "program," the artist moves the joystick so that a cursor is at the desired position on the screen, pushes a button, and command #1 is automatically generated internally. Move the cursor to another position on the screen, push another button, a line appears, and command #2 is generated inside the computer, and so on.

Adding brushes

The PICDRAW routine in *The Graphics Magician* takes several other commands, most notably *brushes*. What are brushes on the computer? They're a neat little way to use character graphics. Remember the character generator we did? Well, suppose we have a character generator that will plot the characters in any X, Y position on the screen, not just in the regular text columns. Suppose also that we can do this in any color we want. Now let's redefine the character patterns so that instead of As, Bs, and Cs, we've got big dots, little dots, feathered dots, and whatever else looks neat. Use the character generator to plot these wherever we want on the screen in any color we desire, and we have paintbrushes! How useful are they? Well, instead of coloring-book pictures with outlined figures filled with certain colors, now the images can be shaded, boundary lines disappear, extra detail can be added—in other words, a lot more sophistication can be added to the pictures created.

Pictures on pictures

There are other tricks and benefits to using pictures created this way. One was designed, another discovered by accident after several months of use. The *picture interpreter,* PICDRAW, gives two options. It can clear the screen before drawing the next picture, or it can draw the next picture over what's already there. Advantage: as with adventure games, where objects can be moved from one room to another, object pictures can be drawn after the room picture, right on top of it, giving the illusion of being part of that same picture. Thus you can save innumerable extra pictures by assembling a few components in different ways. Of course, adventure games use this technique a lot. Another example is in a game called *Police Artist.* In it, numerous variations of facial parts are stored and then put together in different combinations to create thousands of different faces (I think they say millions, and they're probably right).

Surprise animation

The other neat trick, unbeknownst to us when *Magician* was written, is that you can animate using pictures created this way! The first person who discovered it told me that he was drawing a man, finished the drawing, then decided that the eyes weren't exactly right. Instead of going back and deleting the moves that made the eyes in the first place, he just redrew over the eyes to get them right. When the picture was redrawn with PICDRAW, though, the man in the picture blinked! Recreating the artist's moves, the eyes were drawn one way, then, a split second later, redrawn another way. After this discovery, we've now seen complete choreographed animations done by creating a picture and drawing back over it dozens of times, with all the moves saved in the little binary *picture program* file. Neat stuff!

Saving picture files

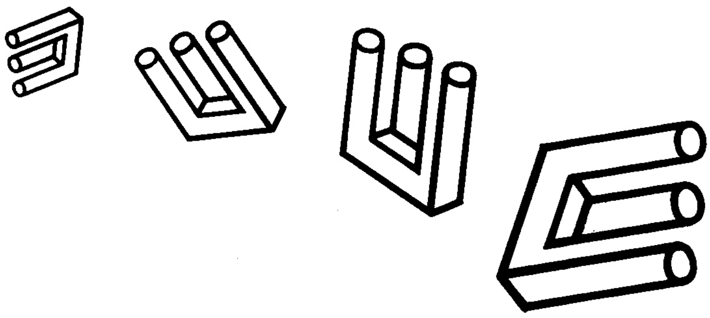
For those of you who are interested in exactly how the picture files are saved in *The Graphics Magician,* below is a breakdown of the currently used commands and their structure. Note that there is room for extra commands, and they'll be used for optional results and on other computers so that the picture programs can be transferred back and forth from an Apple to an Atari or Commodore or IBM, and so forth, with a minimum of fooling around required. The PICDRAW routine itself consists of a line subroutine, a fill subroutine, a brush and character subroutine, and an interpreter that reads and interprets the instructions and calls the appropriate subroutines to correctly redraw the picture.

Instructions

Hex Opcode (4 bits)	*Bytes*	*Meaning*
0	1	Picture end
1	3	Set text cursor
2	1	Set line color
3	2	Plot reverse text character
4	1	Set brush number
4	2	Plot color text character
6	2	Set brush/fill color
8	3	Start line
A	3	Draw line
C	3	Plot brush
E	3	Fill

In the first byte of the instruction, the left 4 bits (4 bits = 1 nybble) are used for the opcode. The right 4 bits are used for data. For example, the 3-byte commands all need X and Y values, with X requiring more than one byte of storage. The high end of the X value is stored in the right nybble of the 1st byte. The low end of the X is stored in the second byte, and Y is stored in the third byte. In the line color and brush number commands, the right nybble is used to store that data. Since there are over 100 fill colors used, the *set fill color* command needs a second byte to store the appropriate number.

FIFTEEN



3-Dimensional Graphics

There are a few ways to make figures on your computer appear to have 3-dimensional qualities. Obviously, they all require some tricks, since you start with a 2-dimensional screen.

3-D Perceptions

Each technique involves using the perceptions that make you think something is truly 3-dimensional. The most obvious of these perceptions is that an object that appears large when close becomes very tiny when it is at a distance. Drawing a small object on the computer screen, then successively drawing it larger and larger, can give the appearance of it coming at you from a distance. The second quality is rotational: turn a 3-dimensional object, and you should be able to see its sides and back. The third quality gives us *vanishing points*. Stand in middle of a set of railroad tracks and look down them at a distance. Even though the tracks are parallel, they look like they meet way off in the distance. (This perception, if you think about it, is somewhat related to the first: size. The distance between the rails remains the same, but appears smaller at a distance.) A fourth perception, which isn't used often on small computers, is that of coloring, shading, and shadowing.

The perceptions listed are given generally in order of easiest to most difficult to simulate on the computer. Each of these techniques can be effective on its own, but the illusion can be even more striking when they are combined. Let's look at some examples of using each.

Size

A couple of games that use strictly size for 3-D appearances are Larry Miller's *Epoch* and *Hadron*. Both are space-type games that use the 3-D size perception very well. Because the other techniques aren't used, Miller accomplished two very nice things graphically: color and speed. His shapes each have their own colored detail, and they move fast enough to make the game play succeed.

Rotation and perspective

Turning an object to see its sides and back requires something more than scaling a shape. The problems of vanishing-point appearances require similar solutions. The programmer's choices are to store one shape and do a series of calculations to determine its appearance at various angles (generally slow and not allowing much detail), or to store a different shape for each of many points of rotation of the object (very, very space-consuming). The first option, calculation, is the one most often used. Examples are games like *Battlezone* in the arcades, and Bill Budge's 3-D games and utilities. To minimize calculations and complexity, the shapes are usually line drawings only, and usually vanishing point calculations are omitted for speed's sake (*Flight Simulator* is an example of one that does use vanishing point computations, also). Another game, *Way Out*, uses an interesting approach with line graphics. By not erasing the old lines as you move through a 3-D maze, the colored trail of the old lines give the walls a solid color, looking like full-color 3-D.

The helicopter in Dan Gorlin's *Choplifter* is an example of storing different shapes for rotation points. As you turn the helicopter, the program changes the shapes that are drawn to the screen, and you visibly turn. Look for more programs using this technique more fully in the future. While we're on examples, though, an example of using only vanishing points to give a 3-D effect is the titling in the Star Wars movies—one can almost imagine it rolling down the railroad tracks.

Shading and shadowing

The fourth quality, that of shading and shadowing, is perhaps the least used because of complexity. Only in very sophisticated graphics machines, like those used in making the movie Tron, do you see it used well. You could conceivably use shadowing if you are storing different shapes for rotations of an object, and you'll probably see that used to some extent soon. An interesting, neat (and relatively easy) application of shadowing is used in the arcade game *Zaxxon* to give a stronger 3-D image. The shadow of the plane you control is seen on the ground below you, giving a nice illusion of height. Anyone who's seen Mattel's newer baseball game for the *Intellivision* will recognize the same technique in use.

Creating 3-D Shapes

As you can see, the most promising avenue for animation and game-play is to create multiple shapes and use a lot of tricks to fool one's perceptions. Oddly enough, in using these it is the programmer, not the computer, doing all the work by pre-designing all the shapes. Here, we'll pursue ways to get the computer to do the work. We'll try designing a 3-D shape and let the computer show us what it will look like from different views (actually, a lot of programmers who do the 3-D effects by hand start with a computer-design program to generate the views, then select the views that they want to save and use in their animations). To do this, we'll limit ourselves to line drawings, the so-called *wire-frame* 3-D graphics.

Simulating 3-D line graphics is not too difficult if you know a few formulas and understand a couple of underlying basic concepts. (*Know* may be too strong; let's say *use*. *Know* implies some understanding of exactly why they work, which may take a while.)

First, how do we take a 3-D object and put it on a 2-dimensional screen? There are a few ways to approach it, but let's take the idea of a window placed between you and an actual 3-D object—the house across the street, for example. Figure 15.1 shows how, in tracing the house

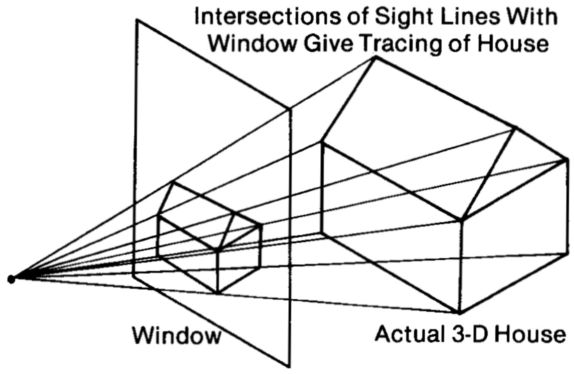

Intersections of Sight Lines With Window Give Tracing of House
Window
Actual 3-D House

Figure 15.1 A Three-Dimensional Image

onto your window as you view it, you actually create a 3-D projection. Very important is that only the endpoints of lines need be projected. A projected line will always connect its projected endpoints. What that means for us is that in a program, we'll only have to worry about endpoints of lines. The fewer things we have to rotate and manipulate, the better!

Next, we need a way to describe the points that we'll have floating around in space. 3-D coordinates in the form (X,Y,Z) are typically used, so that's what we'll use. Figure 15.2 shows how we'll use these coordinates to describe points. For convenience, we'll put the X and Y axes on the screen, and use Z to describe depth. The point (0,0,0) will be in the middle of the screen, on the screen. X will increase to the right, Y will increase as you move up, and Z will increase as an object moves away from you (into your monitor, out the back, through the wall, and doing a lot of damage in the process if it's of any size).

We still have to get that 3-dimensional coordinate projected onto our 2-dimensional screen. Actually, the projection formula is pretty simple, using proportions. The only additional number you need is some hypothetical distance that your eye is from the screen (or, at what Z-coordinate is your eye located?). Since we have no idea what units we're measuring, any number will do (and it will also affect the results in ways that you can play with later). We'll call that number ID (for eye distance?—Oh, well). Take a look at Figure 15.3 for a sketch of how we get the proportions, in this case for Y. Y is the 3-D point's Y-coordinate, Z is the 3-D point's Z-coordinate, and PY is the projected Y-coordinate. The projected Z-value is zero, since it's on the screen. The same sketch could be used for the projected X-coordinate. The formula is the same when X is plugged in instead of Y.

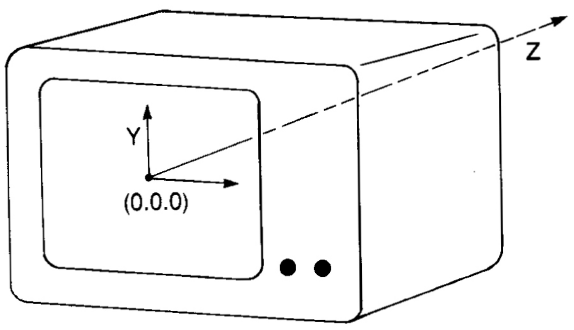


Figure 15.2 Orientation of Three-Dimensional Axes

Anyway, looking at the sketch, there are two similar triangles, whose sides must be proportional. (Okay, geometry students, why are the triangles similar? Did you say because each of the corresponding angles are congruent? Right!) The proportions you get are:

$$\frac{Y}{ID+Z} = \frac{PY}{ID}$$

With a little manipulation, that's equivalent to:

$$PY = \frac{Y * ID}{ID + Z}$$

With the same approach, you also get:

$$PX = \frac{X * ID}{ID + Z}$$

And that's how you get your projected coordinates (PX,PY) out of any old 3-D coordinate set.

So what happens is you take the coordinates of all the endpoints in a figure, store them in some kind of array in the computer, remember (better yet, tell the computer) which coordinates get connected by lines (sort of like connect-the-dots), project all the coordinates onto the screen, and then finally connect all the projected coordinates with lines.

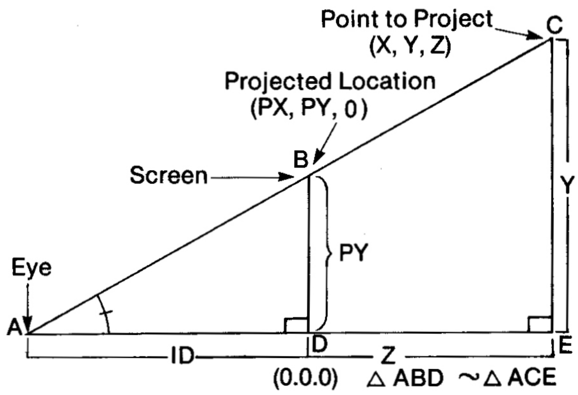


Figure 15.3 Computing Proportions for a Projected Point

The neat part now is that once you have all these coordinates in an array, you can do things to them, such as turning them upside down, sideways, inside out, or whatever you please.

A Few Things You Can Do With Your Coordinates

First, let's put the coordinates somewhere it's convenient to talk about them. Use three arrays, X(?), Y(?), and Z(?). If you have 10 endpoints, X will go from X(1) to X(10), and so on. The first point will have coordinate (X(1),Y(1),Z(1). Got the general idea?

Move it

You can move your object in any direction by just adding some number to any set of coordinates. Add 5 to all the X values, for example, and the object will move to the right 5 units. Add 100 to all the Z values, and the object should move back 100 units (through the monitor, the wall, the garage...), and should appear much smaller when next displayed.

Give it a center

Before anything else, pick a point in the center of the object and remember it. When you scale or rotate an object, you'll need a center as a reference point.

Why it's a good idea to give it a center. What happens if the center isn't very close to the object? Let's use a rotation for an example. Say the center we choose is somewhere near the sun. Rotate the object 45 degrees around the center. Where's its display point? Is it anywhere near your monitor screen? Nooooooo. Riding around on the Earth, your monitor would catch up to it in about 46 days, over 1.5 billion miles away in space. Better to rotate your object in place; that is, around a center in its center.

Make it BIGGER or smaller

You can easily scale your object so that its actual size (not perceived size, as controlled by distance away) changes. Say you want to double the size; your multiplier is two. First, subtract the coordinate of the center from each of your points. Then multiply each coordinate by your multiplier, in this case, 2. Last, add the center coordinate back onto each of the now-scaled coordinates. The deal with the center is similar to what happens with rotating. If you don't do it, your figure can be zapped way out into space. This way, it's *scaled in place*. Hmmm. For some real fun, try scaling only one dimension, your X-coordinates, for example. It's great fun mushing your objects, then stretching them back out again. The multiplier can be any number (except zero; zero is bad). Numbers like 0.5 will scale an object so that it's smaller.

Rotate it

Here's where you need some old formulas from trigonometry. Suppose you want to rotate d degrees. First find the sine and cosine of d, using S=SIN(d), C=COS(d).

If you want to rotate clockwise or counterclockwise, note that the Z-coordinate won't change. Each depth value remains the same. If (NX,NY,NZ) are the new coordinates, here's the rotation formula:

```
NX = C * X - S * Y
NY = C * Y + S * X
NZ = Z
```

To rotate left or right (around the Y-axis, sort of like a merry-go-round):

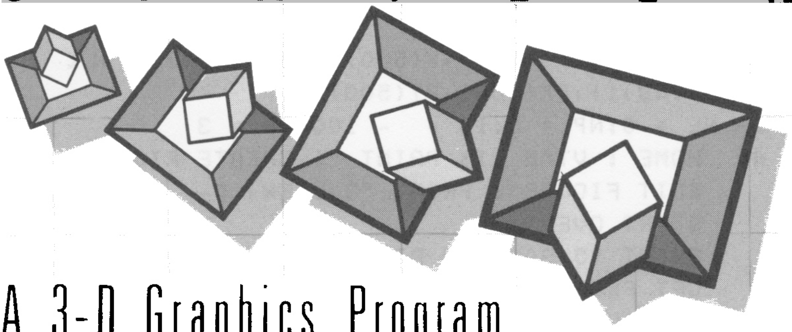
```
NX = C * X - S * Z
NY = Y
NZ = C * Z + S * X
```

And to rotate around the X-axis (up or down, like looking at the back of a paddlewheel on a steamboat):

```
NX = X
NY = C * Y - S * Z
NZ = C * Z + S * Y
```

So much for that. In the next chapter there's a listing of a Basic program that does all of the above.

SIXTEEN



A 3-D Graphics Program

The 3-D program chapter (Listing 16.1) allows you to enter the coordinates for a 3-D line drawing, enter the dot-to-dot information for the lines, then lets you view your object, rotate it 3-dimensionally, move and scale it, stretch and compress it, and save and retrieve it to and from disk. The program is in Basic, so don't expect animation speed in the movement. It does, however, give a good introduction to how 3-D figures can be handled in a computer.

The program uses exactly the same calculations we discussed last chapter. The calculation for projecting a 3-D coordinate onto the 2-dimensional screen can be found in lines 840-860 (note that since the values used are coordinates, not distances as in our formula, ID - Z(I) is actually the distance between ID and Z, since one of the actual values is negative and the other positive). The calculations for moving the figure in each direction are in lines 1290-1310. The scaling calculations are in lines 780-810, and the rotation calculations are in lines 740-760. The rest of the program is essentially window dressing, allowing you to enter, edit, save, and load your figures, and to make your choices.

The only section to note mathematically is that from 970-1080. Although it looks complex, it's actually just broken down into cases. Those instructions perform the calculations necessary for clipping. When a line is ready to be drawn, what happens if one of the endpoints is off the physical screen? The easy options are to just not draw it at all (even though half or most of it might actually be in the visible range) or to have the endpoints wrap around (creating a real mess, but it's fast). The nice way, albeit slower, is to figure out at which point the line would have gone off the screen, and then use that as the endpoint for drawing the line. The calculations use the slope of the line to figure the point at

(text continued on page 130)

Listing 16.1

```
10  LOMEM: 16384: HGR
20  DIM XR(1),YR(1),X(500),Y(500),Z(500),L%
    (749,1),PX(500),PY(500)
30 NL = 0:NP = 0:ID =  - 100:VS = 3
40  HOME : VTAB 21: PRINT "1-CREATE FIGURE, 2-
    EDIT FIGURE": PRINT "3-VIEW, 4-
    START OVER":
    PRINT "5-SAVE ON DISK, 6-
    GET FROM DISK": PRINT
    "7-SAVE 2 DIMENSIONAL IMAGE, 8-QUIT";
50  INPUT C: IF C < 1 OR C > 8 THEN 40
60  IF  NOT NF AND C > 1 AND C < 6 THEN  PRINT
    : PRINT "THERE IS NO FIGURE IN MEMORY.":
    PRINT
    "<PRESS ANY KEY>";: GET A$: GOTO 40
70  ON C GOTO 510,100,580,30,250,350,460,80
80  TEXT : END
90  REM  EDIT FIGURE
100  TEXT : HOME
110  PRINT "1-POINTS,2-LINES,3-CHANGE,4-DONE
     EDITING";: INPUT C: IF C < 1 OR C > 4
     THEN 110
120  ON C GOTO 130,160,190,40
130  PRINT "#,X,Y,Z:":S1 = 0:SW = 0: FOR I = 1
     TO NP
140  PRINT I;: HTAB 8: PRINT  LEFT$ ( STR$
     (X(I)),6);: HTAB 16: PRINT  LEFT$ (
     STR$(Y(I))
     ,6);: HTAB 24: PRINT  LEFT$ ( STR$
     (Z(I)),6):
     S1 = S1 + 1: IF S1 = 20 THEN  PRINT
     "<PRESS A
     KEY>";: GET A$:S1 = 0: PRINT
150  NEXT : GOTO 110
160  PRINT "#,FROM,TO":SW = 1:S1 = 0: FOR I =
     1 TO NL
170  PRINT I,L%(I,0),L%(I,1):S1 = S1 + 1: IF
     S1
     = 20 THEN  PRINT "<PRESS A KEY>";: GET
     A$:S1 =
     0: PRINT
```

Listing 16.1 (continued)

```
180  NEXT : GOTO 110
190  IF SW THEN 220
200  INPUT "POINT #";I: IF I < 1 OR I
     > NP THEN 110
210  INPUT "X:";X(I): INPUT "Y:";Y(I): INPUT
     "Z:";Z(I): GOTO 110
220  INPUT "LINE #";I: IF I < 1 OR I > NL THEN
     110
230  INPUT "FROM #";L%(I,0): INPUT "TO #";L%
     (I,1): GOTO 110
240  REM  SAVE 3-D FILE
250  ONERR  GOTO 440
260  INPUT "NAME : ";A$
270  PRINT CHR$(4);"OPEN";A$
280  PRINT CHR$(4);"WRITE";A$
290  PRINT NP: PRINT NL
300  FOR I = 1 TO NP: PRINT X(I): PRINT Y(I):
     PRINT Z(I): NEXT
310  FOR I = 1 TO NL: PRINT L%(I,0): PRINT L%
     (I,1): NEXT
320  PRINT CHR$(4);"CLOSE"
330  POKE 216,0: GOTO 40
340  REM  READ 3-D FILE
350  ONERR  GOTO 440
360  INPUT "NAME : ";A$
370  PRINT : PRINT CHR$(4);"OPEN";A$
380  PRINT CHR$(4);"READ";A$
390  INPUT NP: INPUT NL
400  FOR I = 1 TO NP: INPUT X(I),Y(I),Z(I):
     NEXT
410  FOR I = 1 TO NL: INPUT L%(I,0): INPUT L%
     (I,1): NEXT
420  PRINT CHR$(4)"CLOSE"
430  POKE 216,0:NF = 1: GOTO 40
440  PRINT "DISK ERROR"; PEEK (222): PRINT
     "<PRESS ANY KEY>";: GET A$: POKE 216,0:
     GOTO 40
450  REM  SAVE PICTURE
460  ONERR  GOTO 440
470  INPUT "NAME : ";A$
480  PRINT CHR$(4)"BSAVE";A$;",A8192,L8192"
```

Listing 16.1 (continued)

```
490  POKE 216,0: GOTO 40
500  REM  CREATE FIGURE
510 NF = 1: HOME : TEXT : PRINT "TYPE "D" OR
    "DONE" WHEN NO MORE POINTS.": ONERR  GOTO
    520
520  PRINT "POINT #";NP + 1: INPUT "X:";A$: IF
     LEFT$ (A$,1) = "D" THEN 540
530 I =  VAL (A$):NP = NP + 1:X(NP) = I: INPUT
    "Y:";Y(NP): INPUT "Z:";Z(NP): GOTO 520
540  PRINT "TYPE "D" OR "DONE" WHEN NO MORE
     LINES.": ONERR  GOTO 550
550  PRINT "LINE #";NL + 1: INPUT "FROM POINT
     #";A$: IF  LEFT$ (A$,1) = "D" THEN  POKE
     216,0:
     GOTO 40
560 I =  VAL (A$):NL = NL + 1:L%(NL,0) = I:
    INPUT "TO POINT #";L%(NL,1): GOTO 550
570  REM  FIND APPROXIMATE CENTER
580 XH =  - 999:YH =  - 999:ZH =  - 999:XL =
    999:YL = 999:ZL = 999
590  FOR I = 1 TO NP
600  IF X(I) < XL THEN XL = X(I)
610  IF X(I) > XH THEN XH = X(I)
620  IF Y(I) < YL THEN YL = Y(I)
630  IF Y(I) > YH THEN YH = Y(I)
640  IF Z(I) < ZL THEN ZL = Z(I)
650  IF Z(I) > ZH THEN ZH = Z(I)
660  NEXT
670 XC = (XL + XH) / 2:YC = (YL + YH) / 2:ZC =
    (ZL + ZH) / 2
680 C = 4
690  REM COMPUTE NEW POINT COORDINATES
700  FOR I = 1 TO NP
710  IF C = 4 THEN 840
720 X(I) = X(I) - XC:Y(I) = Y(I) - YC:Z(I) =
    Z(I) - ZC:XT = X(I):YT = Y(I):ZT = Z(I)
730  ON C GOTO 740,750,760,840,770
```

Listing 16.1 (continued)

```
740 YT = C1 * Y(I) + S1 * Z(I):ZT = C1 * Z(I)
    - S1 * Y(I): GOTO 820
750 XT = C1 * X(I) - S1 * Z(I):ZT = C1 * Z(I)
    + S1 * X(I): GOTO 820
760 XT = C1 * X(I) - S1 * Y(I):YT = C1 * Y(I)
    + S1 * X(I): GOTO 820
770  ON S1 GOTO 790,800,810
780 XT = M * X(I):YT = M * Y(I):ZT = M * Z(I):
    GOTO 820
790 XT = M * X(I): GOTO 820
800 YT = M * Y(I): GOTO 820
810 ZT = M * Z(I)
820 X(I) = XT + XC:Y(I) = YT + YC:Z(I) = ZT
    + ZC
830  REM  TRANSLATE TO 2-D
840  IF ID - Z(I) >  - .001 THEN K = 10000:
     GOTO 860
850 K = ID / (ID - Z(I))
860 PX(I) = K * X(I):PY(I) = K * Y(I)
870  NEXT
880  REM  DRAW FIGURE ON SCREEN
890  HGR : HCOLOR= 7
900  FOR I = 1 TO NL
910 SW = 0
920  FOR I1 = 0 TO 1
930 XR(I1) = PX(L%(I,I1)) * VS:YR(I1) = PY(L%
    (I,I1)) * VS
940  NEXT
950  FOR I1 = 0 TO 1
960  IF SW THEN 1090
970  IF  ABS (XR(I1)) <  = 139 THEN 1040
980  IF  ABS (YR(I1)) <  = 95 THEN 1010
990  IF YR(0) = YR(1) THEN 1070
1000 YP =  SGN (YR(I1)) * 95:XP = (YP - YR(1))
     * (XR(0) - XR(1)) / (YR(0) -
      YR(1)) + XR(1): IF
     ABS (XP) <  = 139 THEN 1080
```

Listing 16.1 (continued)

```
1010  IF XR(0) = XR(1) THEN 1070
1020 XP =  SGN (XR(I1)) * 139:YP = (XP -
      XR(1))
     * (YR(0) - YR(1)) / (XR(0) -
      XR(1)) + YR(1): IF
     ABS (YP) <  = 95 THEN 1080
1030  GOTO 1070
1040  IF  ABS (YR(I1)) <  = 95 THEN 1090
1050  IF YR(0) = YR(1) THEN 1070
1060 YP =  SGN (YR(I1)) * 95:XP = (YP - YR(1))
     * (XR(0) - XR(1)) / (YR(0) -
      YR(1)) + XR(1): IF
     ABS (XP) <  = 139 THEN 1080
1070 SW = 1: GOTO 1090
1080 XR(I1) = XP:YR(I1) = YP
1090  NEXT
1100  IF  NOT SW THEN  HPLOT 140 + XR(0),96 -
      YR(0) TO 140 + XR(1),96 - YR(1)
1110  NEXT
1120  REM  GET NEXT OPERATION
1130  HOME : VTAB 21: PRINT "1-ROTATE, 2-
      SHIFT,
      3-SCALE OBJECT,": PRINT "4-DISTORT, 5-
      NEW
      CENTER, 6-SCALE VIEW": PRINT "7-
      EDIT, SAVE, OR
      QUIT, 8-FULL SCREEN";
1140  INPUT C: ON C GOTO 1210,1260,1180,1170,
      1340,1390,40,1370
1150  GOTO 1130
1160  REM SCALE FIGURE
1170  PRINT : INPUT "1-WIDTH, 2-HEIGHT, OR
      3-DEPTH?";S1: IF S1 < 1 OR S1
      > 3 THEN 1170
1180  IF C = 3 THEN S1 = 0
1190  INPUT "MULTIPLY BY? ";M:C = 5: GOTO 700
1200  REM  ROTATE
```

Listing 16.1 (continued)

```
1210  HOME : VTAB 21: PRINT "ROTATE  1-UP,
      2-DOWN, 3-LEFT, 4-RIGHT,": PRINT "5-
      CLOCKWISE,
      6-COUNTERCLOCKWISE ";: INPUT C: IF C
      < 1 OR C > 6 THEN 1210
1220  INPUT "ANGLE (0 - 180) ? ";AN: IF AN < 0
      OR AN > 180 THEN 1220
1230 AN = 3.14 * AN / 180: IF  INT (C / 2) * 2
     <  > C THEN AN =  - AN
1240 S1 =  SIN (AN):C1 =  COS (AN):C =  INT
     ((C + 1) / 2): GOTO 700
1250  REM  SHIFT
1260  HOME : VTAB 21: PRINT "SHIFT  1-LEFT,
      2-RIGHT, 3-DOWN, 4-UP,": PRINT "5-
      CLOSER, 6-
      FARTHER ";: INPUT C: IF C < 1 OR C
      > 6 THEN 1260
1270  INPUT "HOW MANY UNITS? ";AN:
      IF  INT (C / 2) * 2 <  > C
      THEN AN =  - AN
1280  ON  INT ((C + 1) / 2) GOTO 1290,1300,
      1310
1290 XC = XC + AN: FOR I = 1 TO NP:X(I) = X(I)
     + AN: NEXT : GOTO 1320
1300 YC = YC + AN: FOR I = 1 TO NP:Y(I) = Y(I)
     + AN: NEXT : GOTO 1320
1310 ZC = ZC + AN: FOR I = 1 TO NP:Z(I) = Z(I)
     + AN: NEXT
1320 C = 4: GOTO 700
1330  REM  NEW CENTER
1340  PRINT "POINT # (1-";NP;") ";: INPUT C:
      IF C < 1 OR C > NP THEN 1340
1350 XC = X(C):YC = Y(C):ZC = Z(C): GOTO 1130
1360  REM  FULL SCREEN
1370  POKE  - 16302,0: GET A$: POKE  -
       16301,0:
      GOTO 1130
1380  REM  SCALE VIEW
1390  INPUT "MULTIPLY BY? ";M:VS = VS * M:C =
      4: GOTO 700
```

which the line should end. Note also that since we're using a coordinate system that has the point (0,0) in the middle of the screen, with X and Y increasing to the right and up, respectively, when the line is actually drawn in 1100, it recomputes the coordinates to fit Apple's (0,0) in the upper left, with Y increasing downward.

A Breakdown of the Program

Lines 10-30 initialize the arrays and variables.

Lines 40-80 give and act on the main editing choices.

Lines 90-230 allow you to list the coordinates and lines in your figure and edit them.

Lines 240-440 let you save or read a 3-D file from disk.

Lines 450-490 let you save the screen image from your projected object as a picture file.

Lines 500-560 allow you to enter the points and lines for a new figure.

Lines 570-680 compute a center for the figure by finding the largest and smallest X, Y, and Z coordinates, then averaging each pair.

Lines 690-820 contain the rotation and scaling formulas for movement. C is set for the type of movement, down in lines 1180-1240.

Lines 830-870 project the 3-D coordinates into the 2-dimensional plane of the monitor screen.

Lines 880-1110 take the projected points, scale them to viewing size, then draw the lines of the figure on the screen.

Lines 1120-1390 get the operation that you want to perform and the parameters of the operation, if necessary, and usually jump back up to line 700 to redraw.

Variables

A$	a general input string
AN	numeric quantity, used for angle of rotation and for number of units to shift an object
C	numeric input choice
C1	cosine of angle of rotation
I	generally a loop counter
I1	another loop counter
ID	"eye distance," as discussed last chapter
K	intermediate multiplier used in finding projected points
L%(*,*)	integer array of line endpoints. Up to 749 lines can be used. Note that the zero element of the second dimension is used, so line 5's endpoints are stored in L%(5,0) and L%(5,1).

M : multiplier for scaling
NF : number of figures, 0 or 1
NL : number of lines
NP : number of points
PX(*) : projected X-values
PY(*) : projected Y-values
S1 : sine of angle of rotation
SW : a switch variable, used in 130-190 to tell if lines or points are being edited, and in 910-1100 to tell if a line is entirely off the screen or not.
VS : viewing scale
X(*),Y(*),Z(*) : X,Y,Z coordinates of your figure
XC,YC,ZC : X,Y,Z coordinates of center
XH,YH,ZH : largest each of X,Y,Z coordinates
XL,YL,ZL : smallest each of X,Y,Z coordinates
XP,YP : interim computational values used for clipping
XR(*),YR(*) : scaled X,Y projected coordinates, used in preparing for line display
XT,YT,ZT : temporary X,Y,Z coordinate with center subtracted off, used in rotation and scaling computations.

SEVENTEEN

A Hi-Res Drawing Program

We've talked about the various Applesoft commands available for drawing on the hi-res screen, but so far we've used them mostly for animation and computed pictures. Here's a program that makes use of all the Applesoft graphics commands for freehand drawing with paddles or a joystick. It gives you two types of line-drawing modes, color control (using the standard six Apple colors), and a paintbrush mode using a shape table. It also lets you save and load your pictures to and from disk.

The program uses a few interesting programming tricks. A couple deal with refining joystick control for drawing, and one shows a quick way to poke in some hexadecimal machine-language code from Basic. It's also fairly short, and has room for adding things like the packing and fill routines from earlier in the series. Doing so is a not-too-complicated task in Basic.

Line and Fill

The two line-drawing modes are called *line* and *fill*. The latter may be somewhat of a misnomer, but it does allow a manual fill. The regular line mode puts two round cursors on the screen. One is a *start point* and the other the *end point* for your line. The start point is stationary. The end point moves under joystick control. When you push button 0 on the joystick, a line is drawn between the start point and end point, *and* the end point becomes the new start point. This way it is possible to draw successive lines, such as around a box, without moving the start point manually each time. Pressing button 1 at any time moves the current start point to the end point position without drawing a line.

(text continued on page 137)

Listing 17.1

```
10 D$ =  CHR$ (4)
20  GOSUB 1000
30  POKE 232,0: POKE 233,3
40 NS = 1:GS = 0:PS = 0:BS = 0:CB = 0:CD = 7:
   SW = 0:ZR = 10:XS = 0
50  HCOLOR= 7: ROT= 0: SCALE= 1
100  HGR
105  POKE  - 16303,0: HOME
108  PRINT "DRAWING PAGE OPTIONS:"
109  PRINT
110  PRINT "L : LINE MODE"
120  PRINT "F : FILL MODE"
121  PRINT "1,2,OR 3 : BRUSH NUMBER/
     BRUSH MODE"
122  PRINT
130  PRINT "C : NEW DRAWING COLOR (0-7)"
140  PRINT "B : NEW BACKGROUND COLOR (0-7)"
141  PRINT
150  PRINT "G : GET A PREVIOUS DRAWING"
160  PRINT "S : SAVE DRAWING"
161  PRINT
162  PRINT "<ESC> : FULL SCREEN/TEXT SWITCH"
163  PRINT "Z : ZERO-IN CURSOR SWITCH"
164  PRINT
165  PRINT "H : HELP; RETURNS TO THIS PAGE"
167  PRINT
170  PRINT "PRESS ANY KEY TO GO TO DRAWING
     PAGE": GET A$
190  POKE  - 16304,0
195  HOME : VTAB 22: HTAB 1
196  PRINT "       TYPE "H" FOR HELP"
199  IF PS = 0 THEN 202
200  IF BS = 0 THEN  PRINT "PAINT MODE, BRUSH
     UP": GOTO 205
201  PRINT "PAINT MODE, BRUSH DOWN": GOTO 205
202  IF SW = 1 THEN  PRINT "FILL MODE": GOTO
     205
203  PRINT "LINE MODE"
205  PRINT "COLOR: ";CD;"   BACKGROUND: ";CB;
```

Listing 17.1 (continued)

```
210 X =  PDL (0):Y =  PDL (1): IF  NOT ZC THEN
    220
215 X = ZX +  INT (X / 128 * ZR) - ZR:Y = ZY +
    INT (Y / 128 * ZR) - ZR: GOTO 225
220  IF X > 245 THEN XS = 30
221  IF X < 10 THEN XS =  - 5
222  X = X + XS:Y = INT (Y / 256 * 202) - 5
225  IF X < 0 THEN X = 0
226  IF Y < 0 THEN Y = 0
227  IF X > 279 THEN X = 279
228  IF Y > 191 THEN Y = 191
230  XDRAW NS AT X,Y
232  IF  NOT PS THEN  XDRAW NS AT X1,Y1
233  IF PS AND BS THEN  DRAW NS AT X,Y
235  IF  PEEK ( - 16384) < 128 THEN 290
240  GET A$: POKE  - 16384,0
245  IF  NOT PS THEN  XDRAW NS AT X1,Y1
246  IF  NOT PS OR  NOT BS THEN  XDRAW NS AT
     X,Y
247  IF A$ = "H" THEN 105
250  IF A$ = "C" THEN 400
260  IF A$ = "B" THEN 500
261  IF A$ = "S" THEN 950
262  IF A$ = "G" THEN 900
265  IF A$ = "Z" THEN ZX = X:ZY = Y:ZC =  NOT
     ZC: GOTO 195
266  IF A$ = "1" OR A$ = "2" OR A$ = "3" THEN
     NS =  VAL (A$): POKE 233,64:BS = 0:PS =
     1: GOTO 195
268  IF  ASC (A$) = 27 THEN  POKE  - 16302 +
     GS,0:GS =  NOT GS: GOTO 195
270  IF A$ = "F" THEN SW = 1: GOTO 850
280  IF A$ = "L" THEN SW = 0: GOTO 850
285  GOTO 210
290  IF  NOT PS OR  NOT BS THEN  XDRAW NS AT
     X,Y
291  IF  NOT PS THEN  XDRAW NS AT X1,Y1
292  IF  PEEK ( - 16286) < 128 THEN 310
296  IF PS THEN BS = 0: GOTO 195
```

Listing 17.1 (continued)

```
300 X1 = X:Y1 = Y
310  IF  PEEK ( - 16287) < 128 THEN 210
316  IF PS THEN BS = 1: GOTO 195
320  HPLOT X1,Y1 TO X,Y: IF  NOT SW THEN X1 =
     X:Y1 = Y
330  GOTO 210
400  HOME : VTAB 22: PRINT "0-7?";
404  GET A$: IF  ASC (A$) > 47 AND  ASC (A$) <
     56 THEN CD =  VAL (A$): HCOLOR= CD: GOTO
     195
406  GOTO 504
500  HOME : VTAB 22: PRINT "0-7?";: GET A$
502  IF  ASC (A$) > 47 AND  ASC (A$) < 56 THEN
     CB =  VAL (A$): HCOLOR= CB: HPLOT 0,0:
     CALL 62454: HCOLOR= CD: GOTO 195
504  PRINT "INVALID COLOR; COMMAND CANCELLED."
     : FOR I = 1 TO 2000: NEXT I: GOTO 195
750  PRINT "THAT FILE ISN"T ON DISK.": FOR I =
     1 TO 2000: NEXT : GOTO 195
850  POKE 233,3:NS = 1: IF PS THEN  XDRAW 1 AT
     X1,Y1:PS = 0
855  GOTO 195
900  HOME : VTAB 22: INPUT "UNDER WHAT NAME IS
     IT SAVED? ";A$
905  ONERR  GOTO 750
910  PRINT D$;"BLOAD "A$;"A,$2000
920  POKE 216,0: GOTO 195
950  HOME : VTAB 22: INPUT "UNDER WHAT NAME?
     ";A$
955  ONERR  GOTO 750
960  PRINT D$;"BSAVE ";A$;",A8192,L8192"
970  POKE 216,0: GOTO 195
1000 L = 768: GOSUB 1100
1010 L = 16384: FOR I = 1 TO 4: GOSUB 1100:
     NEXT
1020  RETURN
1100  READ A$: FOR X = 1 TO  LEN (A$) STEP 2
1120 Y =  ASC ( MID$ (A$,X,1)) - 48: IF Y > 9
     THEN Y = Y - 7
1130 Z =  ASC ( MID$ (A$,X + 1,1)) - 48: IF Z
     > 9 THEN Z = Z - 7
```

Listing 17.1 (continued)

```
1140  POKE L,Y * 16 + Z
1150 L = L + 1: NEXT : RETURN
1200  DATA  "01000400123F20642D15361E0700"
1210  DATA "030008000B001E00"
1220  DATA "353700"
1230  DATA "0D09111B1F130909311B1B1709091111B1F
      1300"
1240  DATA "292D153F3F372D2D353F3F372D2D353B3F
      1700"
```

The *fill* version of line mode works the same way, except the start point doesn't change when you draw a line. The result is that until you change the start point yourself (by pressing button 1), all lines originate from the same point. By holding down button 0 as you move the joystick, you can manually fill an area, albeit slowly.

Brushes

There is a shape table of three brush shapes numbered 1, 2, and 3. Number 1 is a small solid brush, number 3 is a larger solid brush, and number 2 is an *airbrush,* made up of random dots in a small area. *Brushes* were discussed earlier when we talked about the PICDRAW routine from *The Graphics Magician* that used a character generator which allowed use of all the blended colors. The same technique can be used with a shape table, as in this program, except you become limited to the standard six colors.

Keypress Operations

A *help* page lists all the keypress options for the program and what they do. Among other options are the GET and SAVE commands for loading and saving pictures to disk (L, for *load,* was already used for *line* mode), and the Escape key, which toggles between full-screen graphics and mixed text and graphics. C and B control the drawing and background colors, the latter clearing the screen when it sets the new background.

Joystick Control

Z is one of the joystick commands. It *zeroes-in* your joystick to the area immediately around the cursor. Pressing it again toggles back to full-screen control.

Another feature added to joystick control is that you are given the full range of X, from 0 to 279, even though the joystick only returns values from 0-255. It's done with an offset, XS, which is added to the joystick value each time. If you are working on the left 9/10ths of the screen, this offset is -5. This is negative, instead of zero, to compensate for joysticks that are slightly worn and don't go all the way down to zero. (Also for products such as the *Koala Pad,* which, although it's an interesting new graphics input device, has even less resolution than a joystick.) When you are working on the right 9/10ths of the screen, the offset 30 is added to the joystick value, giving it a maximum value of 285. All values are then put in actual range (0-279) before drawing. Note that you don't notice this change in offsets unless you move your cursor from one extreme edge to the other.

Here are the variable names used in the program:

A$: a general input string
BS : *brush switch,* tells whether the brush is up (0) or down (1)
CB : color of the background
CD : color used for drawing
D$: a Control-D, for disk commands
GS : full-screen graphics switch; 0 for mixed text and graphics, 1 for full-screen graphics
I : loop counter
L : location, or address, for poking in shape table data
NS : the number shape that is being drawn
PS : paintbrush mode switch; 0 for not brush mode, 1 for brush mode
SW : line mode switch; 0 for normal line mode, 1 for *fill* line modeX,Y
: movable cursor X,Y position
X1,Y1 : start point position
XS : X offset for joystick adjustment
Z : intermediate variable used in poke subroutine
ZC : zero switch; 0 for full screen cursor, 1 for zeroed-in cursor
ZX,ZY : coordinate around which zeroed cursor is

And here's a breakdown of the sections of the program:
Lines 10-100 initialize everything.
Lines 105-190 display the *help* screen.
Lines 195-205 print the current status (mode, colors) at the bottom of the drawing screen.
Lines 210-228 read the paddle controls, zero-in if in zero mode, adjust X with XS, and check the coordinates for proper range.

Lines 230 and 232 flash the cursors, the start point only if needed. Note that in line 232, the statement IF NOT PS means to us "if not in paintbrush mode." To the computer, IF NOT PS is equivalent to saying IF PS=0. IF PS is the same as the statement IF PS<>0, or "if PS is not equal to zero," or, in our case, "if PS is 1." This logic syntax is used in several places in the program, and does save a lot of bytes in long programs.

Line 233 draws the brush if in paintbrush mode and the brush is down.

Lines 235-285 check for a keypress and process the keypress, after erasing the flashing cursors.

Lines 290-330 erase the flashing cursors, check for a button press, and process any button press.

Lines 400-504 let the user input a new drawing or background color.

Line 750 is a disk error message.

Lines 850-855 turn off the paintbrush switches, if necessary, if one of the line modes is selected. Note that two shape tables are used, one for the flashing round cursor, the other for the brushes. Line 850 pokes the location of the round cursor into the shape table pointer (the second byte is zero for both tables, so it need not be changed). Line 266 has a similar poke for setting the brush shape table.

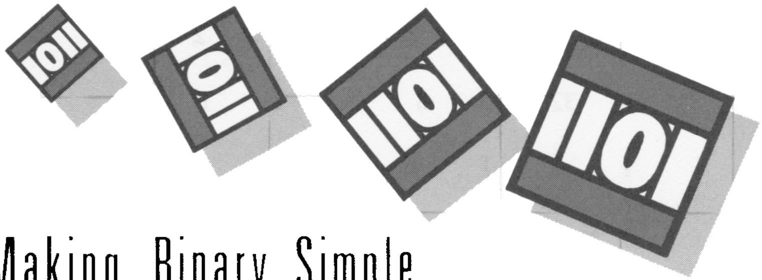
Lines 900-970 let you save or load a picture.

Lines 1000-1020 load the shape table data by setting the starting locations, L, and calling the subroutine at 1100.

Lines 1100-1150 contain a routine that will read a string of character data and interpret that string as hexadecimal numbers (by using the ASCII code of each character). Once each number is interpreted, it is poked into memory, starting at location L.

Lines 1200-1240 contain the hex data for our shape tables. Line 1200 is the cursor shape table, and lines 1210-1240 contain the three brushes. How'd we get the data? Brute force—sorry. It seemed the simplest way to do it in an Applesoft program without having to mess with BLOADing binary files. The original BLOADed shapes were made with the Shape Maker program, but if we printed that data here it would not make for very meaningful typing.

APPENDIX A



Making Binary Simple

Binary isn't really very difficult; it's just awkward to use. In a way, it works very much like the standard base 10 system (decimal), in which each place stands for a power of 10. In decimal, the number 3,286 means three thousands (3 times 10 to the third), two hundreds (2 times 10 squared), eight tens (8 times 10 to the first), and six ones.

Decimal - Breakdown of the number 3286

10^3	10^2	10^1	10^0
1000's	100's	10's	1's
3	2	8	6

In binary, each place designates a power of two. The rightmost place is 2 to the zero power, or 1. The next place is 2 to the first, or 2. The next is 2 squared, or 4. The next is 2 to the third, or 8, and so on. The binary number 1101 is equivalent to the base 10 number 13, since it means that there is one 8, one 4, no 2s, and one 1.

Binary - Anatomy of the number 1101

Power of 2	2^3	2^2	2^1	2^0	
Decimal Values	8	4	2	1	
Binary Number	1	1	0	1	
Decimal Equivalent	8	+ 4	+ 0	+ 1	= 13

Another example, you say? How about 1011011?

2^6	2^5	2^4	2^3	2^2	2^1	2^0
64	32	16	8	4	2	1
1	0	1	1	0	1	1

$64 + 0 + 16 + 8 + 0 + 2 + 1 = 91$

So 1011011 in binary is the same as 91 decimal.

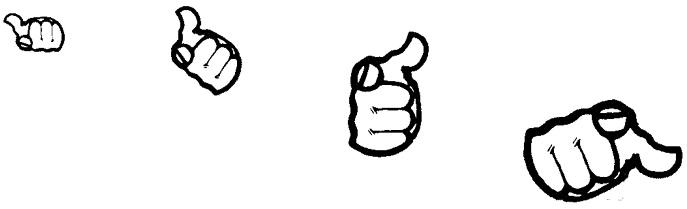
In one byte you have eight bits (Binary digITS), and the largest number possible is every bit set (= 1)

2^7	2^6	2^5	2^4	2^3	2^2	2^1	2^0
128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1

$128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255$

And that's the easy way to convert unsightly binary numbers to the nice, familiar decimal variety.

APPENDIX B



Applesoft Hi-Res Graphics Commands

HGR: clears page 1 of graphics to black and sets the display to page 1 with four text lines at the bottom; 280 dots by 160 dots.

HGR2: clears page 2 of graphics to black and sets the display to full-screen page 2; 280 dots by 192 dots.

HCOLOR = C: sets the current hi-res plotting color to C, where C has a value of 0-7.

HPLOT X,Y: plots a dot at point X,Y in the current HCOLOR.

HPLOT X1,Y1 TO X2,Y2: plots a line in the current HCOLOR from X1,Y1 to X2,Y2. You can add extra TO Xn,Yn clauses. If X1,Y1 is omitted, the line starts at the point last plotted.

POKE 233,INT(L/256) : POKE 232,L - PEEK(233) *256: pokes in the starting address, L, of a shape table.

DRAW S AT X,Y: draws shape number S from a shape table at location X,Y in the current HCOLOR.

XDRAW S AT X,Y: draws shape number S at location X,Y in reverse of whatever's on the screen.

ROT = R: sets the current shape rotation to R. 0 is normal, 16 is 90 degrees clockwise, 32 is 180 degrees, 48 is 270 degrees, and 64 is normal again. Intermediate values give other rotations, depending on the SCALE of the shape.

SCALE=S: sets the shape scale to S. 1 is normal, and S must be an integer from 1 to 255.

CALL 62450: clears current hi-res screen to black.

CALL 62454: clears current hi-res screen to the HCOLOR most recently HPLOTted. You must do an HCOLOR= and an HPLOT X,Y first.

Soft Switches

The following POKEs affect which graphics screen is displayed without clearing the screen, and let you switch from full-screen graphics to mixed text and graphics, and back.

POKE -16304,0: Full-screen text to current graphics mode (lo-res or hi-res).

POKE -16303,0: Graphics mode to text mode; usually only works if you are viewing page 1 graphics.

POKE -16302,0: Mixed text and graphics to full-screen graphics.

POKE -16301,0: Full-screen graphics to mixed text and graphics. Usually only works on page 1.

POKE -16300,0: Switches from page 2 to page 1.

POKE -16299,0: Switches from page 1 to page 2.

POKE -16298,0: Switches from hi-res graphics to lo-res graphics.

POKE -16297,0: Switches from lo-res graphics to hi-res graphics.

APPENDIX C

Machine Language Entry Points For Applesoft Graphics Routines

For those of you who tinker in machine language and who want to try out some machine-language graphics, here are the access points for the Applesoft graphics routines. The hi-res line routine comes in handy when you don't want to write your own—or when you don't have the space for one.

Command	*Equivalent JSR*
HGR	JSR $F3E2
HGR2	JSR $F3D8
HPLOT	JSR $F457

needs Y value in accumulator, X-low in X register, X-hi in Y register.

HPOSITION	JSR $F411

sets the starting point for a line, as does HPLOT, without actually plotting a point. HPLOT or HPOSITION must be used to start drawing with a new HCOLOR, and should be used before the first HPLOT TO. Y-value goes in the accumulator, X-low goes in the X register, and X-hi goes in the Y register.

HPLOT TO	JSR $F53A

takes Y value in Y register, X-low in accumulator, X-hi in X register.

Hi-res Clear	JSR $F3F2

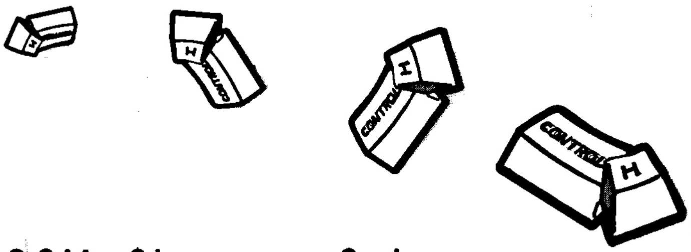
clears the hi-res screen to black.

Background Set	JSR $F3F4

clears the hi-res screen to the color in the accumulator. Instead of 0-7, use 00, 2A, 55, 7F, 80, AA, D5, and FF. These are the hex values that correspond to the various bit masks used for the colors. You can actually use any value and get different (and strange) color results.

A couple of locations that are of interest are $E4, which holds the current HCOLOR value, as above, and $E6, which tells us which hi-res page to draw on. The latter is useful for changing the page on which drawing is done without actually displaying it. Changes can be made on the hi-res screen that's not displayed; then you can use the switch to show that page while drawing on the other (the hex values for those switches, given in decimal as -16300 and -16299 earlier, are $C054 and $C055). $E6 contains $20 if hi-res page 1 is being drawn on, or $40 if hi-res page 2 is in use. In Basic, you can use POKE 230,32 and POKE 230,64 for the same results.

APPENDIX D

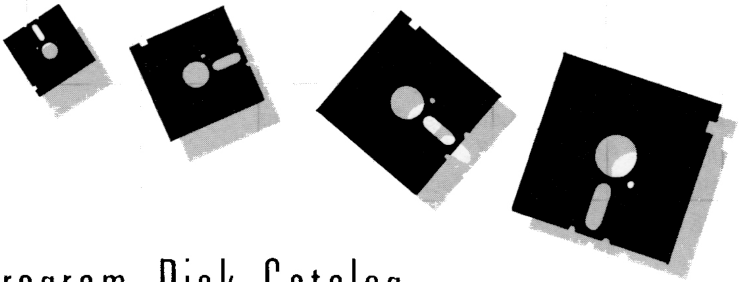


ASCII Character Codes

Number	What to Type		
0	Control-@	26	Control-Z
1	Control-A	27	Escape
2	Control-B	28	n/a
3	Control-C	29	Control-shift-M
4	Control-D	30	Control-rub-on
5	Control-E	31	n/a
6	Control-F		
7	Control-G (bell)	32	space
8	Control-H or left arrow	33	!
9	Control-I or TAB	34	"
10	Control-J or down arrow	35	#
11	Control-K or up arrow	36	$
12	Control-L	37	%
13	Control-M or Return	38	&
14	Control-N	39	'
15	Control-O	40	(
16	Control-P	41	)
17	Control-Q	42	*
18	Control-R	43	+
19	Control-S	44	,
20	Control-T	45	-
21	Control-U or right arrow	46	.
22	Control-V	47	/
23	Control-W	48	0
24	Control-X	49	1
25	Control-Y	50	2

51	3	90	Z
52	4	91	[
53	5	92	Y(reverse slash)
54	6	93	]
55	7	94	hyphen
56	8	95	-
57	9		
58	:	96	‘(reverse apostrophe)
59	;	97	a
60	<	98	b
61	=	99	c
62	>	100	d
63	?	101	e
		102	f
64	@	103	g
65	A	104	h
66	B	105	i
67	C	106	j
68	D	107	k
69	E	108	l
70	F	109	m
71	G	110	n
72	H	111	o
73	I	112	p
74	J	113	q
75	K	114	r
76	L	115	s
77	M	116	t
78	N	117	u
79	O	118	v
80	P	119	w
81	Q	120	x
82	R	121	y
83	S	122	z
84	T	123	{
85	U	124	|
86	V	125	}
87	W	126	~
88	X	127	inverse blank or DELETE
89	Y		

APPENDIX E



Program Disk Catalog

Listing 1.1 : Hi-Res Display

Listing 2.1 : Drawing Simple Rectangles
Listing 2.2A : Drawing More Rectangles
Listing 2.2B : Drawing Complex Shape
Listing 2.3 : Using Coordinates in Arrays

Listing 3.1 : Drawing and Storing a Single-Shape Table
Listing 3.2 : Simple Animation With Shape Maker
Listing 3.3 : Joystick-controlled Shape Maker Animation

Listing 4.1 : Loading and Initializing With Shape Maker
Listing 4.2 : Poking and Initializing Circle Shapes
Listing 4.3 : Joystick-controlled Shape
Listing 4.4 : Joystick-controlled Shape Draw
Listing 4.5 : Keyboard-controlled Shape Draw
Listing 4.6 : Formula-controlled Animation
Listing 4.7 : Pre-defined Path for Shape
Listing 4.8 : Random Computer Scribbling

Listing 5.1 : Explosion
Listing 5.2 : Laser Blast
Listing 5.3 : Laser Blast With Explosion
Listing 5.4 : Bouncing Ball

Listing 6.1 : Computing Starting Addresses
Listing 6.2 : Entering X & Y Coordinate Values
Listing 6.3 : Repetitive Onscreen Pattern for X & Y Coordinate Values

Listing 10.1

```
                          ORG  $6180
                COLORE    EQU  $02
                COLORO    EQU  $03
                MASK      EQU  $04
                BYTEL     EQU  $05
                PZ2       EQU  $08 ;2 BYTE SCRATCH
                YPOS      EQU  $0A
                BYTEN     EQU  $0B
                BITN      EQU  $0C
                VALUE     EQU  $FC
                LS        EQU  $FD
                RS        EQU  $FE
                COLORP    EQU  $FF
                LKHI      EQU  $6000
                LKLO      EQU  $60C0
6180: 00        CNO       DFB  00 ;FILL COLOR, USED BY BASIC
6181: 00        XHI       DFB  0
6182: 00        XLO       DFB  0
6183: 4C 4D 62                JMP  FILL
6186: 43 4F 50  COPYRT    ASC  "COPYRIGHT 1983, MARK PELCZARSKI"
6189: 59 52 49 47 48 54 20 31
6191: 39 38 33 2C 20 4D 41 52
```

Listing 10.1 (continued)

```
6199: 4B 20 50 45 4C 43 5A 41
61A1: 52 53 4B 49
                   *FIND Y BASE ADDRESS,
                   GIVEN YPOS
                   *USING Y-LOOKUP TABLE
                   *ALSO SET COLORP TO COLORE
                   OR
                   *COLORO, DEPENDING ON EVEN
                   OR
                   *ODD Y-ROW.
61A5: A4 0A        BASER    LDY  YPOS
61A7: B9 00 60              LDA  LKHI,Y
61AA: 85 09                 STA  PZ2+1
61AC: B9 C0 60              LDA  LKLO,Y
61AF: 85 08                 STA  PZ2
61B1: A5 0A                 LDA  YPOS
61B3: 4A                    LSR
61B4: B0 05                 BCS  ODROW
61B6: A5 02                 LDA  COLORE
61B8: 4C BD 61              JMP  STCOL
61BB: A5 03        ODROW    LDA  COLORO
61BD: 0A           STCOL    ASL
```

```
61BE: 0A                      ASL
61BF: 85 FF                   STA  COLORP
61C1: 60                      RTS
                   *FIND XBIT/BYTE, GIVEN XHI/LO
61C2: A0 FF        SETX       LDY  #$FF
61C4: AD 82 61                LDA  XLO
61C7: 85 08                   STA  PZ2
61C9: AD 81 61                LDA  XHI
61CC: 85 09                   STA  PZ2+1
61CE: A5 08                   LDA  PZ2
61D0: 38                      SEC
61D1: E9 07        SLOOP      SBC  #7
61D3: C8                      INY
61D4: B0 FB                   BCS  SLOOP
61D6: 38                      SEC
61D7: C6 09                   DEC  PZ2+1
61D9: 10 F6                   BPL  SLOOP
61DB: 84 0B                   STY  BYTEN
61DD: 18                      CLC
61DE: 69 07                   ADC  #7
61E0: 85 0C                   STA  BITN
61E2: 60                      RTS
61E3: 03 03 06     DOTS       DFB  3,3,6,12,24,48,96
61E6: 0C 18 30 60
```

Listing 10.1 (continued)

```
                  *CHECK A DOT
                  *BASER IN PZ2, BYTE IN
                  BYTEN, BIT IN BITN
                  *RESULT RETURNED IN Z BIT,
                  O=CONTINUE
                  *BOTH DOTS SET, <>0 MEANS
                  AT LEAST ONE OFF.
                  *USES "DOTS" TABLE ABOVE
                  FOR COMPARISON.
61EA: A6 0C       CHECK     LDX  BITN
61EC: BD E3 61              LDA  DOTS,X
61EF: A4 0B                 LDY  BYTEN
61F1: 31 08                 AND  (PZ2),Y
61F3: DD E3 61              CMP  DOTS,X
61F6: 60                    RTS
                  *SET COLOR IN BYTE, MASK IN ACC
61F7: 85 04       SETC      STA  MASK
61F9: 45 FC                 EOR  VALUE
61FB: 29 7F                 AND  #$7F
61FD: 85 FC                 STA  VALUE
61FF: 98                    TYA
6200: 29 03                 AND  #3
```

```
6202: 05 FF                 ORA  COLORP
6204: AA                    TAX
6205: BD F8 63              LDA  BASECB,X
6208: 25 04                 AND  MASK
620A: 05 FC                 ORA  VALUE
620C: 91 08                 STA  (PZ2),Y
620E: 60                    RTS
                  *SET COLOR REGISTERS, GIVEN CNO
620F: AD 80 61    SETCR     LDA  CNO
6212: 0A                    ASL
6213: A8                    TAY
6214: B9 20 63              LDA  BASEC,Y
6217: 85 02                 STA  COLORE
6219: C8                    INY
621A: B9 20 63              LDA  BASEC,Y
621D: 85 03                 STA  COLORO
621F: 60                    RTS
                  *SEARCH LEFT FOR 0 BIT
6220: 29 7F       LBITS     AND  #$7F
6222: C9 7F                 CMP  #$7F
6224: D0 04                 BNE  LOOKL
6226: A9 FF                 LDA  #$FF
6228: 30 0C                 BMI  SLMASK
622A: A2 07       LOOKL     LDX  #7
```

Listing 10.1 (continued)

```
622C: 0A                      ASL
622D: CA          LOOPL       DEX
622E: 0A                      ASL
622F: B0 FC                   BCS  LOOPL
6231: 86 FD                   STX  LS
6233: BD 19 63                LDA  LMASK,X
6236: 60          SLMASK      RTS
                  *SEARCH RIGHT FOR 0 BIT
6237: 29 7F       RBITS       AND  #$7F
6239: C9 7F                   CMP  #$7F
623B: D0 04                   BNE  LOOKR
623D: A9 FF                   LDA  #$FF
623F: 30 0B                   BMI  SRMASK
6241: A2 FF       LOOKR       LDX  #$FF
6243: E8          LOOPR       INX
6244: 4A                      LSR
6245: B0 FC                   BCS  LOOPR
6247: 86 FE                   STX  RS
6249: BD 12 63                LDA  RMASK,X
624C: 60          SRMASK      RTS
                  *FILL ROUTINE
624D: 20 0F 62    FILL        JSR  SETCR ;SET COLOR REGISTERS
```

```
6250: 20 C2 61                JSR  SETX ;FIND XBYTE/XBIT
6253: 20 A5 61       TLOOP    JSR  BASER ;START LOOP, FIND Y LOC.
6256: 20 EA 61                JSR  CHECK ;CHECK IF DOTS SET
6259: D0 09                   BNE  STFILL ;IF OFF, START FILL
625B: A5 0A                   LDA  YPOS
625D: F0 0A                   BEQ  STFILL1 ;IF TOP, START FILL
625F: C6 0A                   DEC  YPOS ;MOVE UP ONE MORE LINE
6261: 4C 53 62                JMP  TLOOP
6264: E6 0A          STFILL   INC  YPOS
6266: 20 A5 61                JSR  BASER
6269: A4 0B          STFILL1  LDY  BYTEN
626B: A9 FF          MIDDLE   LDA  #$FF ;FILL THE MIDDLE BYTE
626D: 85 FD                   STA  LS
626F: A9 07                   LDA  #7
6271: 85 FE                   STA  RS
6273: B1 08                   LDA  (PZ2),Y
6275: 85 FC                   STA  VALUE
6277: A6 0C                   LDX  BITN
6279: 1D 19 63                ORA  LMASK,X
627C: 20 20 62                JSR  LBITS
627F: 85 04                   STA  MASK
6281: A6 0C                   LDX  BITN
6283: BD 12 63                LDA  RMASK,X
6286: 05 FC                   ORA  VALUE
```

Listing 10.1 (continued)

```
6288: 20 37 62            JSR  RBITS
628B: 25 04               AND  MASK
628D: 20 F7 61            JSR  SETC
6290: A5 FD       LEFT    LDA  LS ;NOW MOVE TO THE LEFT
6292: C9 FF               CMP  #$FF
6294: D0 13               BNE  RIGHT
6296: 88                  DEY
6297: 30 0D               BMI  RIGHT1
6299: B1 08               LDA  (PZ2),Y
629B: 85 FC               STA  VALUE
629D: 20 20 62            JSR  LBITS
62A0: 20 F7 61            JSR  SETC
62A3: 4C 90 62            JMP  LEFT
62A6: C8          RIGHT1  INY
62A7: 84 FD               STY  LS
62A9: 84 05       RIGHT   STY  BYTEL
62AB: A4 0B               LDY  BYTEN
62AD: A5 FE       RLOOP   LDA  RS ;NOW FILL TO THE RIGHT
62AF: C9 07               CMP  #7
62B1: D0 15               BNE  DOWN
62B3: C8                  INY
62B4: C0 28               CPY  #$28
62B6: F0 0D               BEQ  DOWN1
```

```
62B8: B1 08            LDA  (PZ2),Y
62BA: 85 FC            STA  VALUE
62BC: 20 37 62         JSR  RBITS
62BF: 20 F7 61         JSR  SETC
62C2: 4C AD 62         JMP  RLOOP
62C5: 88        DOWN1  DEY
62C6: C6 FE            DEC  RS
62C8: 98        DOWN   TYA ;AVERAGE ENDPOINTS AND
62C9: 18               CLC ;MOVE DOWN
62CA: 65 05            ADC  BYTEL
62CC: 85 05            STA  BYTEL
62CE: A0 00            LDY  #0
62D0: 8C 81 61         STY  XHI
62D3: 0A               ASL
62D4: 0A               ASL
62D5: 2E 81 61         ROL  XHI
62D8: 0A               ASL
62D9: 2E 81 61         ROL  XHI
62DC: 38               SEC
62DD: E5 05            SBC  BYTEL
62DF: B0 03            BCS  SKIPD
62E1: CE 81 61         DEC  XHI ;*7
62E4: 18        SKIPD  CLC
62E5: 65 FE            ADC  RS
```

Listing 10.1 (continued)

```
62E7: 90 04      196            BCC  SKIPI
62E9: EE 81 61   197            INC  XHI
62EC: 18         198            CLC ;+RS+LS
62ED: 65 FD      199   SKIPI    ADC  LS
62EF: 90 03      200            BCC  SKIPI2
62F1: EE 81 61   201            INC  XHI
62F4: 4E 81 61   202   SKIPI2   LSR  XHI ;/2
62F7: 6A         203            ROR
62F8: 8D 82 61   204            STA  XLO
62FB: 20 C2 61   205            JSR  SETX
62FE: E6 0A      206            INC  YPOS
6300: A5 0A      207            LDA  YPOS
6302: C9 C0      208            CMP  #$C0
6304: F0 0B      209            BEQ  FDONE
6306: 20 A5 61   210            JSR  BASER
6309: 20 EA 61   211            JSR  CHECK
630C: D0 03      212            BNE  FDONE ;IF DOT OFF, DONE
630E: 4C 6B 62   213            JMP  MIDDLE ;ELSE REPEAT FILL LOOP
6311: 60         214   FDONE    RTS
6312: 80 81 83   215   RMASK    HEX  808183878F9FBF
6315: 87 8F 9F BF
6319: FE FC F8   216   LMASK    HEX  FEFCF8F0E0C080
631C: F0 E0 C0 80
```

```
6320: 03 07 16  217  BASEC     HEX  030716071A
6323: 07 1A
6325: 1D 1C 17  218            HEX  1D1C17080B001B0004031B03061A06
6328: 08 0B 00 1B 00 04 03 1B
6330: 03 06 1A 06
6334: 00 06 00  219            HEX  0006001102061C131013100702 1B02
6337: 11 02 06 1C 13 10 13 10
633F: 07 02 1B 02
6343: 07 02 17  220            HEX  07021702091A04100402051217 1A07
6346: 02 09 1A 04 10 04 02 05
634E: 12 17 1A 07
6352: 03 17 16  221            HEX  031716190305030D1A0D1A05100500
6355: 19 03 05 03 0D 1A 0D 1A
635D: 05 10 05 00
6361: 0D 00 17  222            HEX  0D0017080516050105160B01070117
6364: 08 05 16 05 01 05 16 0B
636C: 01 07 01 17
6370: 01 09 01  223            HEX  01090104160 40C0F011B01110C170C
6373: 04 16 04 0C 0F 01 1B 01
637B: 11 0C 17 0C
637F: 04 16 13  224            HEX  041613010616060C1107070404071B
6382: 01 06 16 06 0C 11 07 07
638A: 04 04 07 1B
638E: 1B 1D 07  225            HEX  1B1D071106071706061B06060406 11
```

Listing 10.1 (continued)

```
6391: 11 06 07 17 06 06 1B 06
6399: 06 04 06 11
639D: 13 04 11  226             HEX  1304111707170B171905071705070D
63A0: 17 07 17 0B 17 19 05 07
63A8: 17 05 07 0D
63AC: 05 05 05  227             HEX  0505050D0D0F040D1704051B050603
63AF: 0D 0D 0F 04 0D 17 04 05
63B7: 1B 05 06 03
63BB: 03 16 03  228             HEX  031603030C0000081A02161A1C0310
63BE: 03 0C 00 00 08 1A 02 16
63C6: 1A 1C 03 10
63CA: 02 03 02  229             HEX  0203021A0202121C001A121A101200
63CD: 1A 02 02 12 1C 00 1A 12
63D5: 1A 10 12 00
63D9: 10 03 1A  230             HEX  10031A161A16120102161801030 11A01
63DC: 16 1A 16 12 01 02 16 18
63E4: 01 03 01 1A 01
63E9: 16 01 01  231             HEX  160101010016001 60C160E0C0E000C
63EC: 01 00 16 00 16 0C 16 0E
63F4: 0C 0E 00 0C
63F8: 00        232  BASECB     HEX  00
63F9: 00 00 00  233             HEX  000000552A552A2A552A557F7F7F7F80
63FC: 55 2A 55 2A 2A 55 2A 55
```

```
6404: 7F 7F 7F 7F 80
6409: 80 80 80  234           HEX  808080D5AAD5AAAAD5AAD5FFFFFFFF33
640C: D5 AA D5 AA AA D5 AA D5
6414: FF FF FF FF 33
6419: 66 4C 19  235           HEX  664C19B3E6CC994C193366CC99B3E611
641C: B3 E6 CC 99 4C 19 33 66
6424: CC 99 B3 E6 11
6429: 22 44 08  236           HEX  22440891A2C48844081122C48891A222
642C: 91 A2 C4 88 44 08 11 22
6434: C4 88 91 A2 22
6439: 44 08 11  237           HEX  440811A2C48891081122448891A2C4C9
643C: A2 C4 88 91 08 11 22 44
6444: 88 91 A2 C4 C9
6449: A4 92 89  238           HEX  A49289241249247776E5D3BF7EEDDBB5D
644C: 24 12 49 24 77 6E 5D 3B
6454: F7 EE DD BB 5D
6459: 3B 77 6E  239           HEX  3B776EDDBBF7EE6E5D3B77EEDDBBF73B
645C: DD BB F7 EE 6E 5D 3B 77
6464: EE DD BB F7 3B
6469: 77 6E 5D  240           HEX  776E5DBBF7EEDD
646C: BB F7 EE DD
```

INDEX

Mark Pelczarski

This book is a compilation of the Graphically Speaking tutorial columns that originally appeared in *Softalk* magazine. Using the programs in this book you will be able to create art, do animation for games, and have a bunch of fun on your Apple II, II+, or IIe.

Founder and President of Penguin Software, the company that brings you the best in computer graphics, Pelczarski has created such classic software as *The Graphics Magician*, *The Complete Graphics System*, and *Special Effects*.

After earning degrees in Computers and Education, he taught mathematics, computer programming, and computer science at both the high school and university levels.

He now continues to produce new software and update programs for his company, while working and consulting with other major software developers and publishers on graphics and graphics routines in their products.

Once you learn the fundamentals of creating hi-res, 3-D, and animation with *Graphically Speaking*, you will be limited only by your imagination.