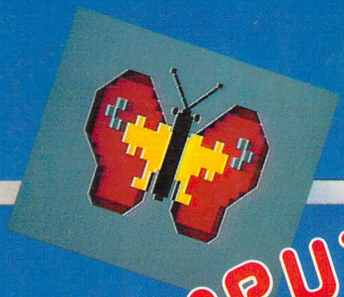
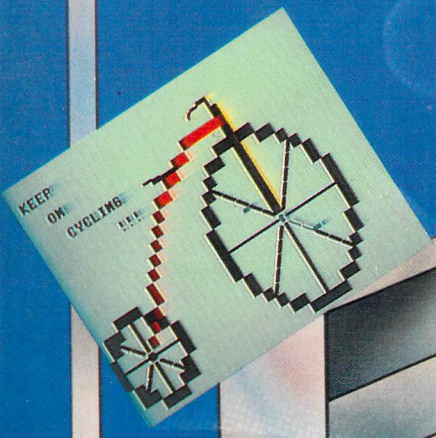
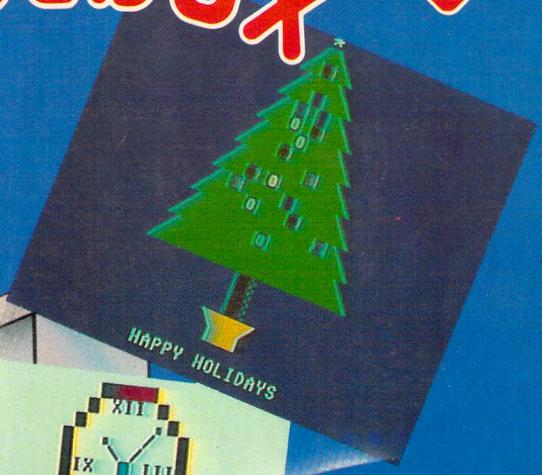
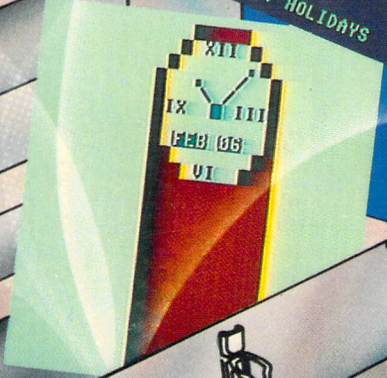
COMMODORE 64



COMPUTER GRAPHICS TOOLBOX







Russell L. Schnapp Irvin G. Stafford

P-H
PRENTICE–HALL PERSONAL COMPUTING SERIES

COMMODORE 64 COMPUTER GRAPHICS TOOLBOX

PRENTICE-HALL
PERSONAL COMPUTING SERIES

Lance A. Leventhal, series editor

FABBRI, *Animation, Games, and Graphics for the Timex-1000*
FABBRI, *Animation, Games, and Sound for the Apple II/IIe*
FABBRI, *Animation, Games, and Sound for the Commodore 64*
FABBRI, *Animation, Games, and Sound for the IBM PC*
FABBRI, *Animation, Games, and Sound for the VIC-20*
FABBRI, *Animation, Games, and Sound for the TI 99/4A*
HARRIS & SCOFIELD, *IBM PC Conversion Handbook of BASIC*
SCANLON, *EasyWriter II System Made Easy-er*
SCANLON, *The IBM PC Made Easy*
SCHNAPP & STAFFORD, *Commodore 64 Computer Graphics Toolbox*
SCHNAPP & STAFFORD, *Computer Graphics for the Timex 1000 and Sinclair ZX-81*
SCHNAPP & STAFFORD, *VIC 20 Computer Graphics Toolbox*
THRO, *Making Friends With Apple Writer II*

COMMODORE 64 COMPUTER GRAPHICS TOOLBOX

RUSSELL L. SCHNAPP
IRVIN G. STAFFORD

PRENTICE-HALL, INC., Englewood Cliffs, New Jersey 07632

Library of Congress Cataloging in Publication Data

Schnapp, Russell L. (date)
Commodore 64 computer graphics toolbox.

Includes index.
1. Computer graphics. 2. Commodore 64 (Computer)
—Programming. I. Stafford, Irvin G. II. Title.
III. Title: Commodore sixty-four computer graphics
toolbox.
T385.S32 1985 001.64'43 84-13357
ISBN 0-13-152075-X (bk)
ISBN 0-13-152091-1 (bk/dsk)
ISBN 0-13-152083-0 (cassette)

Editorial/production supervision: *Karen Skrable Fortgang*
Interior design: *Lynn Frankel*
Cover design: *Jeannette Jacobs*
Manufacturing buyer: *Gordon Osbourne*

Commodore 64 is a registered trademark of Commodore Electronics Limited.

Printed in the United States of America

10 9 8 7 6 5 4 3 2 1

ISBN 0-13-152075-X
ISBN 0-13-152091-1 {BOOK/DISK}
ISBN 0-13-152083-0 {CASSETTE} 01

Prentice-Hall International, Inc., *London*
Prentice-Hall of Australia Pty. Limited, *Sydney*
Editora Prentice-Hall do Brasil, Ltda., *Rio de Janeiro*
Prentice-Hall Canada Inc., *Toronto*
Prentice-Hall of India Private Limited, *New Delhi*
Prentice-Hall of Japan, Inc., *Tokyo*
Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*
Whitehall Books Limited, *Wellington, New Zealand*

CONTENTS

EDITOR'S FOREWORD

Where do you find the crowds at the computer shows? Usually around booths that feature graphics and its applications in flight simulators, science fiction movies, computer-aided design and engineering, drawing of human portraits, video games, and the creation of animated figures and cartoons. Even people with no interest in (and perhaps even a little fear of) computers cannot resist an opportunity to use one to draw lines, rectangles, circles, game pieces, playing cards, mazes, animals, and other pictures. Schnapp and Stafford show the beginner how to do all these things through a series of simple programs. All the reader needs is a Commodore 64 computer, a color television set, and some imagination.

The key features of this book are:

- The examples are all interesting ones that you can RUN and watch on the screen. You get the chance to draw a heart, a flag, an elephant, maps, a calendar, a Christmas tree with flashing lights, lines, rectangles, circles, a tic-tac-toe board, a hand of cards, a pair of dice, a maze, a Martian lander, and a spaceship launch.
- Complete explanations, sample results, and suggested modifications let you readily build on the initial programs.
- Full, documented listings in BASIC give you examples you can study and use for reference.
- Two picture editors provide surprisingly capable tools for drawing, saving, and changing pictures. These also illustrate the principles behind advanced graphics systems.

- An extensive turtle graphics program shows the ideas behind LOGO and provides an excellent demonstration for children and novices.
- Complete maze, tic-tac-toe, and spaceship lander programs demonstrate the principles of game design, including drawing of game pieces, animation, interaction with the players, keyboard control, scoring, and timing.

This book lets the beginner explore computer graphics easily and inexpensively. I think you'll find the subject really is just as fascinating as it looks.

LANCE A. LEVENTHAL
Series Editor

PREFACE

Graphics, the drawing of figures and pictures, is surely among the most intriguing and exciting uses of personal computers. Computer drawings of spaceships, cartoon characters, vehicles, forms, landscapes, mechanisms, and human figures always attract large audiences at shows and demonstrations. In video arcades, games with smooth animation, attractive colors, familiar or amusing characters, and exciting visual and sound effects always draw the players.

This book describes how to create simple drawings on an inexpensive Commodore 64 home computer. Rather than merely telling you how, we present a series of programs. These range from a simple one that creates a color test pattern, to more complex ones that shuffle and deal playing cards, draw rectangles and circles, and simulate a spacecraft landing. We have written all programs in BASIC and have explained each step in detail. We have also described modifications that readers may make on their own. The best way to learn about graphics is by doing it yourself, so you should try the modifications and experiment with the programs.

We have assumed no special background in either computers or mathematics. You should have read the *Commodore 64 User's Guide* and have some familiarity with BASIC before starting this book. We have used a little mathematics (particularly geometry) on occasion, but we have always explained the reasons for this and what the mathematics actually means in practice. In order to keep the programs simple, we do not include sprite graphics.

To use this book, you will need a standard Commodore 64 computer.

You will also need a television set, and a cassette recorder or floppy disk drive will be very helpful. The recorder must be Commodore's C2N Cassette unit (Datassette). The floppy disk drive can be either the Commodore VIC-1540 or VIC-1541. You can use almost any black-and-white or color television, but we recommend a 13-inch color model (such as the Commodore-1702). Several programs give you the option of using a joystick to play games or draw figures. Both Commodore and other manufacturers supply this item. All the programs are available on cassette or disk from the publisher. Ordering information is elsewhere in the book.

This book begins with a description of the program format. **Chapter 1** then explains the program notation, provides a brief overview of the Commodore 64, discusses the use of the cassette recorder and disk, and introduces the computer's graphics capabilities. The first program, "COLORBARS," not only illustrates the notation but also produces a test pattern you can use to adjust your color television set. "COLORTEST" makes it easy to experiment with the different combinations of border, screen, and character colors.

Chapter 2 contains programs that draw pictures using both characters and graphics symbols. The pictures include a heart, cartoon figures, two maps, a flag, and a Christmas tree. One program demonstrates the production of standard forms by generating a calendar for any month of any year.

Chapter 3 describes the drawing of geometrical shapes such as lines, rectangles, and circles. It also contains a program that creates the popular "turtle graphics."

Chapter 4 covers games and animation. It includes programs that throw dice, shuffle and deal playing cards and that simulate a bouncing ball, a spaceship landing, and a rocket launch.

Chapter 5 contains two artist's assistant programs that will help you draw pictures. These programs let you place sequences of characters anywhere on the screen, transfer pictures to or from tape or disk, change parts of pictures, and insert text and geometrical shapes. All commands are either single keystrokes or simple movements of a joystick.

ACKNOWLEDGMENTS

Thanks are due to the following people, who helped the authors with this book:

Our spouses, Brigid Hom-Schnapp and Rosemary Stafford, for putting up with us (yet again!) during this book's gestation process.

The editorial staff of PC Editorial Service, and Lance Leventhal and Paula Criswell for their copious and thoughtful directions and suggestions.

Pete Evaristo, for his artistic help with several drawings.

Certain members of the publisher's staff, namely Jim Fegen of Acquisitions and Karen Fortgang of Production, for their competence and serenity.

Our photographer, Carter Stafford.

Commodore Electronics Limited, for their permission to reproduce certain figures from *Personal Computing on the VIC-20*, the *Commodore 64 Programmer's Reference Guide*, and the *Commodore 64 User's Guide*.

RUSSELL L. SCHNAPP
IRVIN G. STAFFORD

PROGRAM FORMAT

All programs in this book appear in the following format:

PROGRAM NAME: Name used to load or save the program on tape or disk. This always starts and ends with a quotation mark.

PURPOSE: What the program does.

TECHNIQUES DEMONSTRATED: Programming techniques that are being introduced or that you may want to use in other programs.

PROCEDURE: How to use the program.

VARIABLES: A list of all the variables and their meanings, in alphabetical order.

SPECIAL CASES: Exceptions, limitations, and other considerations.

BRIEF DESCRIPTION: Concise description of the program, by line number.

LISTING: Complete, commented program listing. When entering a program you may simply omit the REM statements. This will not affect execution.

MODIFICATIONS: How to change the program to illustrate its approach, expand its capabilities, or demonstrate alternative methods.

NOTES: Additional information.

REFERENCES: Additional reading on the subject.

COMMODORE 64 COMPUTER GRAPHICS TOOLBOX

1

INTRODUCTION

This chapter starts with a brief overview of the Commodore 64's major features. We emphasize the special keys—what they do, how you use them, and how we refer to them. COLORBARS, which produces a color test pattern, serves as a typical example program. We then describe how to use the cassette recorder and disk drive. Finally, we introduce a standard numbering system for describing positions on the screen.

OVERVIEW OF THE COMMODORE 64

The inexpensive Commodore 64 is a complete home computer. Together with a television set and a cassette recorder or disk drive, it is capable of playing games, giving lessons, keeping accounts and files, performing business and engineering calculations, preparing letters and reports, creating charts and pictures, recording grades and attendance, and even controlling your home's lights, furnace, or appliances. It is truly the equal of many large computers of the 1960s and '70s that cost hundreds of thousands of dollars and occupied entire rooms.

By itself, the Commodore is about the size of a portable typewriter. In fact, when you first see it, it looks much like a typewriter keyboard that someone detached from its printing mechanism. A complete Commodore system consists of the following:

1. A "brain" or controller, called a *central processing unit* in computer jargon. This unit, located inside the keyboard, does the computer's "thinking" and calculating.

2. Memory, also located inside the keyboard. This is like a person's memory except that the computer forgets everything when its power is turned off. We measure computer memory in units of *bytes* (note the odd spelling). A byte can hold a single typed *character* (i.e., a letter, digit, punctuation mark, or other symbol, such as $ or +). Thus, the standard Commodore that starts with the message:

```
**** COMMODORE 64 BASIC V2 ****
64K RAM SYSTEM 38911 BASIC BYTES FREE
```

has room in its memory for 38,911 typed characters, or about 26 double-spaced typed pages.

3. Television set or, in computer terminology, *video display.* This is where you see your entries and the computer's prompts, responses, and results. The Commodore can put 25 lines on the display at a time, and each of them can be up to 40 characters long. The lines are shorter than on a typewritten page, and the characters are small but readable.

4. Cassette recorder or disk drive. These are used to "play" tapes or disks into the computer and to record them from it. A disk drive acts much like a record player, except that it plays thin, flexible pieces of plastic called *floppy disks.* As with music tapes, you can buy prerecorded disks or tapes for use with your Commodore, or you can record your own. When buying disks or tapes, however, be careful to buy only ones intended for a Commodore. Trying to use tapes or disks intended for other computers would be like trying to play the wrong size record on a phonograph.

5. Keyboard. This is the main part of the Commodore 64 and looks much like a typewriter keyboard. There is no printing mechanism, since what you type appears on the screen instead of on paper.

KEYBOARD

Now that we have briefly described all the Commodore 64's components, let us concentrate on the keyboard. If you just look at the tops of the keys, only the ones along the edges are unfamiliar. Most of the inside keys are like those on a typewriter. You will, however, notice a few extra symbol keys. The colon and semicolon are on separate keys, and +, −, @, *, and = each have their own keys. There are also arrow keys, a separate key for the British pound symbol (after all, there'll always be an England), and some extra brackets.

Some outside keys are also the same as on a typewriter. The Shift keys (one on each side) have two uses. In the normal (graphics) mode, they let you type special graphics characters (we will talk more about these later). In the other (text) mode, the Shift keys let you type capital letters. In either

mode, Shift lets you type uppercase symbols such as $, ", and ?. The SHIFT LOCK key (just above the left-hand Shift key) lets you lock Shift in, so you don't have to press it each time when entering several consecutive shifted characters. RETURN (the large key to the right of =) acts like the carriage return on a typewriter; you use it to conclude a line and go on to the next one.

How about the other keys around the edges of the keyboard? We may describe them generally as falling into three categories: program control, screen functions, and editing and color control.

Of these, the easiest to describe are the program control keys. The RUN/STOP key (just left of SHIFT LOCK) is used mainly (as you might guess) to stop a program. If this doesn't work, emphasize your point by pressing RUN/STOP and the RESTORE key (just above RETURN). The only thing to watch here is not to press RESTORE instead of RETURN; these keys are easy to confuse, since they are close together and have similar names.

Screen Functions and Editing

The screen control and editing keys are necessary because working on a screen differs from working on paper. Unlike a typewriter, the computer has no typing element or carriage to indicate the typing position. Thus, the computer itself must provide a moving indicator, which we call the *cursor*. The Commodore's cursor is a flashing square which is always just ahead of where you are typing.

We can move the cursor with the two CRSR keys located in the lower right-hand corner. The one on the left (marked with arrows pointing up and down) moves the cursor up or down a line (up if you press Shift at the same time or have SHIFT LOCK down). The one on the right (marked with arrows pointing right and left) moves the cursor right or left a column (left if you press Shift at the same time or have SHIFT LOCK down). Thus, these two keys (together with Shift) let you move the cursor anywhere on the screen. Since the CRSR keys repeat if you hold them down, you can move the cursor a long way quickly.

Once you have the cursor where you want it, entering things is simple. To enter something new, you just type it. To change what is already there, you type over the old characters. The new characters replace the old ones automatically; you do not have to erase the old ones first, as you do on a correcting typewriter.

Watch one subtle difference between the Commodore and the typewriter. Pressing the space bar on a Commodore actually puts something on the screen (that is, it puts a space in the current character position, replacing whatever was there before). This is different from a typewriter, where pressing the space bar just moves the carriage or typing element right. To

produce that effect on the Commodore, you must press the CRSR key with right and left arrows.

While pressing the space bar erases a character, it still leaves a space behind. To remove a character completely (say you typed PRIUNT instead of PRINT), move the cursor just to the right of it (e.g., on top of the N that follows the U in PRIUNT) with the CRSR keys, and press INST/DEL (the key in the top right-hand corner). This erases the character to the left of the cursor (e.g., the U) and moves the cursor and the characters to its right (e.g., NT) left automatically to fill the gap. Try it. Actually trying it and seeing what happens is essential here. DEL erases the character left of the cursor, not the one underneath it. Note the difference between DELeting a character and erasing it (that is, printing a space over it).

To insert a character (say you typed PRNT instead of PRINT), move the cursor to where you want the addition to appear (e.g., to the N in PRNT) and press Shift and INST/DEL simultaneously. A space opens up under the cursor. You can then enter the I into it. To insert a space, be sure to press the space bar; the space that appears on the screen is for display purposes only. The character that was under the cursor (and the part of the line to the right of it) move right to make room for the insertion.

Using the Commodore's delete and insert features (particularly the delete) becomes more natural with practice. Note that you can delete several characters (always to the left of the cursor) by pressing INST/DEL repeatedly. In fact, you can just hold INST/DEL down, since it repeats (rather quickly, we should warn you). The cursor moves left as the characters disappear. Similarly, you can make room for several extra characters (always to the right of the cursor) by pressing Shift and INST/DEL together repeatedly. When you do this, the cursor does not move. Instead, spaces appear to the right of it as the rest of the line moves right.

The last cursor or screen control key we will discuss is CLR/HOME, which is just left of INST/DEL. CLR (uppercase) clears the screen and moves the cursor to the top left-hand corner; HOME (lowercase) moves the cursor to the same place, but does not affect the screen. This key thus provides a quick way to remove unwanted material from the screen or return the cursor to its normal starting point.

Color

The Commodore also provides keys for changing the printing color. There are 16 colors to choose from. You can see 8 of them on the fronts of the number keys 1 through 8. These are: black (BLK, key #1), white (WHT, key #2), red (RED, key #3), cyan or greenish-blue (CYN, key #4), purple or magenta (PUR, key #5), green (GRN, key #6), blue (BLU, key #7), and yellow (YEL, key #8). To change to one of these printing colors, press its key and CTRL (Control, just above SHIFT LOCK). Eight other colors are also available

on number keys 1 to 8, although they are not marked. They are: orange (key #1), brown (key #2), light red (key #3), gray 1 or light gray (key #4), gray 2 or medium gray (key #5), light green (key #6), light blue (key #7), and gray 3 or dark gray (key #8). To change to one of these colors, press its key and the Commodore key (the key with the strange looking C in the lower left-hand corner). Try typing in all the different colors. White is quite distinct (you may even prefer to use it or cyan rather than the normal light blue), red is difficult to read, and blue is invisible. Printing blue characters on a blue background is like writing with invisible ink. To restore the normal light blue printing color, press either the Commodore key and 7 together or RUN/STOP and RESTORE.

Not only can you change the Commodore's printing color, but you can also reverse its printing and background colors. This results in characters that look like X rays or photographic negatives, since the normal foreground and background are reversed. To reverse colors, press CTRL and 9 simultaneously (note the designation RVS ON on the front of the 9 key). Type your name, first in normal characters and then in reversed characters. Note how the reversed characters stand out; programs often use them to emphasize important messages. To go back to normal printing, press CTRL and 0 (RVS OFF). Pressing RETURN also restores normal printing.

Graphics

Now we have described almost all the Commodore's keyboard except for the strange-looking symbols on the fronts of most letter and symbol keys. We refer to these as *graphics symbols,* since they serve as building blocks for drawing pictures, lines, figures, charts, and graphs. You may compare them to the parts in children's building sets or the standard shapes and sizes of pipe or lumber used in construction. You can draw a variety of forms and pictures using these simple shapes, although it takes some practice and patience.

In fact, the Commodore has two separate sets of graphics symbols. To enter a symbol on the left-hand side of a key (we call these *left-hand graphics*), you press the key together with the Commodore key. You enter the symbols on the right-hand side, called *right-hand graphics,* by pressing a key together with Shift *when the keyboard is in the graphics mode.* We will explain keyboard modes a little later.

Try entering some graphics symbols. First press RUN/STOP and RESTORE simultaneously to clear the screen and put the message READY. at the top. Enter some left-hand graphics by holding the Commodore key down and pressing A, S, D, F, G, H, J, K, and L (the center row, from left to right) in succession. Enter some right-hand graphics by engaging SHIFT LOCK and pressing the same keys. Thus, for example, pressing Commodore and J produces a dark bar in the left-hand part of the character position, while press-

ing the Commodore and L keys produces an equivalent dark bar in the right-hand part. Similarly, shifted A is a playing card spade symbol, while shifted K is a curved line in the upper left-hand corner.

Some graphics symbols look almost like typed letters or symbols. Note, for example, the differences between:

- The graphics cross (the right-hand symbol on the + key) and the + sign (+ is much smaller).
- The graphics slash (the right-hand symbol on the N key) and the / symbol just left of Shift (/ is smaller).
- The graphics X (the right-hand symbol on the V key) and X (X is smaller).

Keyboard Modes

We must explain now that the Commodore keyboard has two modes: text and graphics. This is similar to a typical radio that has both AM and FM bands. To tune in a station, you must select the proper band as well as the correct frequency. In the text mode, pressing letter keys produces lowercase letters, while pressing them with Shift produces capital letters. You would use this mode for writing letters or reports, and for doing bookkeeping or business calculations. In the graphics mode, pressing letter keys produces capital letters, while pressing them with Shift produces right-hand graphics. This is the mode the Commodore starts in, and is the mode we will use throughout this book.

If you are not sure which mode the Commodore is in, you can always put it back in graphics mode by pressing RUN/STOP and RESTORE simultaneously. To change modes (i.e., to go from graphics mode to text mode or text mode to graphics mode), press the Commodore and Shift keys together. This works like the on/off buttons often used to control room lights or television sets. When the unit is off, pressing the button turns it on and vice versa.

As we noted, you can change modes by pressing the Commodore and Shift keys together. Let's try this. Put the keyboard in the graphics mode by pressing RUN/STOP and RESTORE simultaneously. The message READY. should appear at the top of the screen. Now engage SHIFT LOCK and type OPOPOPOP. You should see a series of graphics symbols that look like a simple wooden fence. Now press Commodore and Shift together. What happens on the screen is quite remarkable. READY. changes instantaneously to ready. and the wooden fence changes to OPOPOPOP. Now press Commodore and Shift together again, and everything changes back to the way it was. This is like a museum or amusement park exhibit in which a trick lens or a mirror is suddenly inserted between you and the screen, thus changing ancient skeletons into live animals or vice versa.

An obvious question is, "How do I know which mode the keyboard is

in?" Often you can tell by looking at the screen. If program lines or messages such as READY. or **** COMMODORE 64 BASIC V2**** are in lowercase letters, the keyboard is in text mode. If program lines or messages are in capital letters, the keyboard is in graphics mode. If the screen is blank or you cannot determine the mode from its contents, press a letter key with SHIFT LOCK up (unengaged). If a lowercase letter appears on the screen, the keyboard is in text mode; if a capital letter appears, the keyboard is in graphics mode.

In general, what happens on the screen when you change the keyboard mode from graphics to text is:

Capital letters change to lowercase letters.

Right-hand graphics change to capital letters.

Most other symbols (e.g., numbers and most left-hand graphics) stay the same.

Obviously, changing the keyboard mode from text to graphics produces the opposite changes. Since we will use only the graphics mode, all you must know is how to return to it if you accidentally put the keyboard in text mode. The method, as we noted, is to press the Commodore and Shift keys simultaneously.

NOTATION

The Commodore's wide selection of special keys and graphics symbols results in serious notational problems. We could easily fill an entire book with statements like "press Shift and the CRSR key with the up-and-down arrows four times" or "enter the right-hand graphics symbol on the K key." Clearly, we need a shorthand.

Furthermore, we can actually tell the computer to do what a cursor control or color key indicates while it is running a program. If, for example, we want the computer to move the cursor up a line, all we must put in the program is:

```
PRINT "(press Shift and CRSR with up and down arrows)"
```

That is, after typing the quotation mark, you simply press the keys that would move the cursor up a line. Similarly, you can make the computer change the printing color, clear the screen, or reverse printing and background colors during a program.

The problem is how to designate all this in a way that is compact and readable. We have chosen to follow the lead of *Compute* magazine and use the following notation:

1. Braces indicate a command such as CLR or HOME, cursor move-

ment, color change, or a sequence of consecutive spaces. They look like {and}.

2. Inside the braces, we put the name of the function, using the logical UP, DOWN, RIGHT, and LEFT for cursor moves; RVS for REVERSE ON; and OFF for REVERSE OFF. Otherwise, we use the marking on the key, such as BLK, WHT, CLR, HOME, DEL, or INST. Note that we use only the uppercase or lowercase marking. For example, DEL and INST are actually the same key; DEL is lowercase, while INST is uppercase.

3. An underline indicates Shift. For example, J means shifted J, the right-hand graphics symbol on the J key.

4. Special brackets indicate the Commodore key. They look like ❴ and ❵. For example, ❴P❵ means the character obtained by pressing Commodore and P together, the left-hand graphics symbol on the P key. A digit alone inside these brackets indicates a change to an alternate color. For example, ❴3❵ makes the printing color light red.

5. A number in front of a character or function inside braces or brackets indicates how many times to press that key. For example, {20 DOWN} means to press the DOWN key (CRSR with arrows pointing up and down) 20 times, thus moving the cursor down 20 rows. Similarly, ❴5 T❵ means to press Commodore and T together five times, thus entering five copies of the left-hand graphics symbol on the T key.

6. A caret (^) indicates a space. A number in front of a caret inside braces indicates the number of spaces. For example, {5 ^} means to press the space bar 5 times. We use this notation only when it would otherwise be unclear how many spaces are needed.

Table 1–1 summarizes all this notation, along with what actually appears on the screen. The computer itself uses a shorthand to indicate the various special keys. Unfortunately, its shorthand is not meaningful to a human reader.

Clearly, this notation takes a little time to learn. It helps if you have experience with ancient languages and are used to working in hieroglyphics or Sanskrit, English, and perhaps French or German simultaneously. Seriously, practice will accustom you to the intricacies of entering Commodore graphics. This may also give you new insight into the problems faced by Japanese companies writing manuals for American or European consumers.

EXAMPLE PROGRAM

The program COLORBARS serves three purposes. First, it provides practice with our notation. It is brief, so try typing it in even if you have the tape or disk of programs for this book. Second, it illustrates the use of FOR . . . NEXT loops to repeat program lines a specific number of times. Third, the COLORBARS pattern is like the test pattern used by television technicians to adjust colors. You can therefore use it to adjust your television.

TABLE 1-1 KEYBOARD NOTATION

Notation	Keys	Function	On screen
{CLR}	Shift CLR/HOME	Clears screen and moves cursor to top left-hand corner	
{HOME}	CLR/HOME	Moves cursor to top left-hand corner	
{UP}	Shift CRSR UP/DOWN	Moves cursor up one line	
{DOWN}	CRSR UP/DOWN	Moves cursor down one line	
{LEFT}	Shift CRSR LEFT/RIGHT	Moves cursor left one column	
{RIGHT}	CRSR LEFT/RIGHT	Moves cursor right one column	
{INST}	Shift INST/DEL	Makes room to insert a character	
{DEL}	INST/DEL	Deletes a character	
{RVS}	CTRL 9/RVS ON	Reverses printing and background colors	
{OFF}	CTRL Ø/RVS OFF	Returns printing and background colors to normal	
{BLK}	CTRL 1/BLK	Makes printing color black	
{WHT}	CTRL 2/WHT	Makes printing color white	
{RED}	CTRL 3/RED	Makes printing color red	
{CYN}	CTRL 4/CYN	Makes printing color cyan	
{PUR}	CTRL 5/PUR	Makes printing color purple	

TABLE 1-1 KEYBOARD NOTATION (*continued*)

Notation	Keys	Function	On screen
{GRN}	CTRL 6/GRN	Makes printing color green	
{BLU}	CTRL 7/BLU	Makes printing color blue	
{YEL}	CTRL 8/YEL	Makes printing color yellow	
{key}	Commodore/ character key	Left-hand graphics symbol	
{key}	Shift/ character key	Right-hand graphics symbol	
^ or {3 ^}	SPACE BAR	Space	
{1}	Commodore/1	Makes printing color orange	
{2}	Commodore/2	Makes printing color brown	
{3}	Commodore/3	Makes printing color light red	
{4}	Commodore/4	Makes printing color dark gray	
{5}	Commodore/5	Makes printing color medium gray	
{6}	Commodore/6	Makes printing color light green	
{7}	Commodore/7	Makes printing color light blue	
{8}	Commodore/8	Makes printing color light gray	

Program Name: "COLORBARS"

Purpose

Produces a color test pattern (see Figure 1–1) you can use to adjust the television set to the computer's color signal.

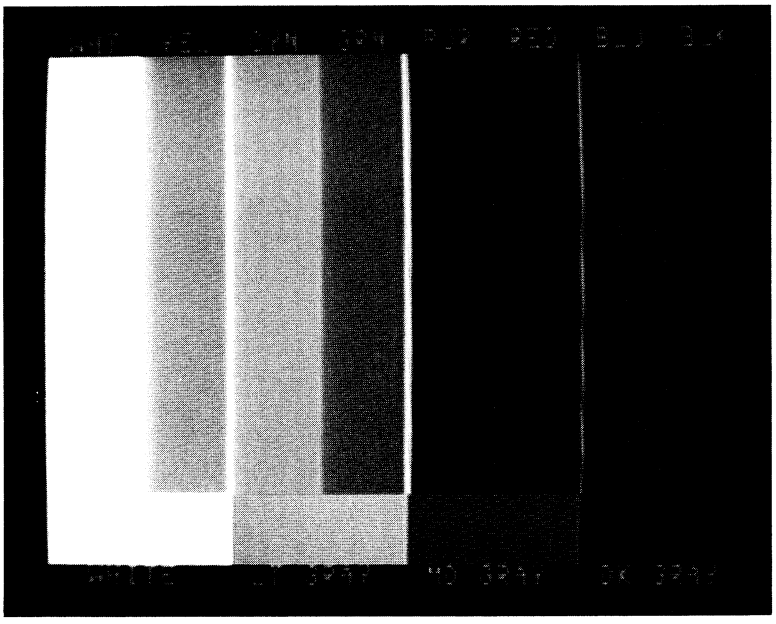


Figure 1–1 Test pattern from COLORBARS

Techniques Demonstrated

Using a FOR . . . NEXT loop to repeat program lines a specific number of times.

Procedure

Enter and RUN the program. It produces a color test pattern you can use to adjust the brightness, contrast (picture), color (gain), and hue (tint) controls on your television set.

Warning: You may have to change these adjustments if you also use your set for watching television programs.

Locate the controls on your television set. Some sets lack an adjustable brightness control. The color (gain) control determines the picture's color intensity.

Turn the brightness, contrast, and color controls counterclockwise or to their minimum effect. Set the hue or tint control to its middle position. The television screen will be black or have a gray bar on the left with no color. Slowly turn the brightness clockwise or up until you just begin to see light on the border. Turn the control back and forth until you are sure the background is a true black. Now turn the contrast control clockwise or up until the left-most bar and the letters are true white with no distortion.

Adjust the contrast so you can see all the bars. The left white bar should be true white, the right bar and the border true black, and the bottom row of gray bars should be descending shades of gray.

Now move the color control to its center position. Adjust the hue control until each bar's color matches its label. Readjust the color control to suit your taste. If you cannot get the colors to look right, the fine tuning may be wrong. If your set has automatic fine tuning (AFT) or other automatic color tracking, turn it off and readjust the fine tuning.

Variables

J: Counter

K$: User key entry, used to terminate the program.

Brief Description

Line 10 makes both the background and screen colors black.

Line 20 prints the color bar labels.

Lines 30 to 50 create the color bars.

Lines 60 to 80 create the gray bars.

Line 100 prints the gray bar labels.

Line 110 branches to itself until the user presses any key.

Lines 120 to 130 restore the standard screen, background, and printing colors.

Listing

```
5 REM "COLORBARS"
10 POKE 53280,0: POKE 53281,0
20 PRINT "^WHT^^YEL^^CYN^^GRN^^PUR^^RED^^BLU^^BLK"
30 FOR J=1 TO 19
40 PRINT "{RVS}{WHT}{5 ^}{YEL}{5 ^}{CYN}{5 ^}{GRN}{5 ^}
         {PUR}{5 ^}{RED}{5 ^}{BLU}{5 ^}"
50 NEXT J
55 REM PUT GRAY BARS NEAR BOTTOM
60 FOR J=1 TO 3
70 PRINT "{RVS}{WHT}{10 ^}{8}{10 ^}{5}{10 ^}{4}{9 ^}"
80 NEXT J
90 PRINT "{7}";
100 PRINT "^^WHITE^^^^LT GRAY^^^MD GRAY ^^DK GRAY^^"
110 GET K$: IF K$="" THEN 110
120 POKE 53280,14: POKE 53281,6
130 PRINT "{CLR}{7}"
140 END
```

Notes

POKE 53280,CC and POKE 53281,CC change the border and screen (background) colors respectively to the one given by CC in Table 1–2. There are 16 possible colors.

Note the following about the combinations:

1. Using the same border and background color produces a solid-colored screen, since the border and background blend.

2. POKE 53280,14 and POKE 53281,6 produces the normal (startup) state—a blue screen with light blue border and letters.

3. The screen color determines which printing colors are reasonable. If the two colors are not sufficiently different you will not be able to read the printing. Be careful not to lose the cursor when you enter POKE 53281,CC; it will disappear or become difficult to see if the printing color is the same as or indistinct from the new screen color.

Lines 40 and 70 produce solid bars by printing reversed spaces in a particular printing color. Since a space is simply a character position entirely in background color, a reversed space is a solid square in the printing color.

Lines 110 through 130 illustrate the standard approach we use in this book for terminating programs. The aim here is to end the program without distorting the picture, but still return control with the standard color combination.

Line 110 simply waits for the user to press a key. It branches to itself as long as K$ is empty (that is, equal to the so-called *null string*). You can

TABLE 1-2 SCREEN AND BORDER COLORS

Color Code	Color
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow
8	Orange
9	Brown
10	Light red
11	Dark gray
12	Medium gray
13	Light green
14	Light blue
15	Light gray

therefore terminate the program by pressing any key; the space bar is a safe choice. After you press a key, line 120 restores the standard screen and border colors, while line 130 restores the standard printing color.

You can also exit programs by pressing RUN/STOP or RUN/STOP and RESTORE. RUN/STOP alone, however, leaves the colors as they were in the program.

Watch the following when using the single-key exit feature:

1. Both it and RUN/STOP distort the picture, since the computer immediately prints its READY message.
2. Both it and RUN/STOP–RESTORE change the screen and border colors back to their usual blue and light blue, respectively.
3. If you start typing (e.g., you decide to list, erase, or change the program) without exiting first, you will lose the first keystroke. This is because the program will use that keystroke as its signal to exit from line 110.

Program Name: "COLORTEST"

Purpose

Allows you to set the border, screen, and character colors (16 choices are available). Makes it easy to experiment with the different color combinations.

Techniques Demonstrated

Using DIMensioned arrays and DATA statements to produce color commands and codes in response to user's requests.

Procedure

Enter and RUN the program. Type the name of the color you want for the BORDER, SCREEN, and CHARACTERs. If you do not want to change the current color, just press RETURN. To see how text looks with a given color combination, LIST the program. If you end up with a combination that is hard to read and want to start over, just press the RUN/STOP and RESTORE keys together.

Variables

CNAME$: Array of 16 color names
KOMD$: Array of 16 commands for changing the printing color
K: Counter
NAME$: Color name entered by the user

Special Cases

The program does not check to see if the inputs are reasonable. If you misspell or mistype a color name the program ignores the input.

Brief Description

Lines 10 to 20 make room for the arrays (lists) of color names and color change commands.

Lines 30 to 60 contain the names of the colors and the color change commands. The color names are ordered as in Table 1–2.

Line 70 constructs the lists of color names and commands.

Line 80 asks for the border color.

Lines 90 to 110 change the border color by finding the code corresponding to its name.

Line 120 asks for the screen color.

Lines 130 to 150 change the screen color by finding the code corresponding to its name.

Line 160 asks for the character color.

Lines 170 to 190 change the printing color by finding the command corresponding to its name.

Listing

```
5 REM "COLORTEST"
10 DIM CNAME$(15)
20 DIM KOMD$(15)
30 DATA BLACK,"{BLK}",WHITE,"{WHT}",RED,"{RED}",
        CYAN,"{CYN}",PURPLE,"{PUR}"
40 DATA GREEN,"{GRN}",BLUE,"{BLU}",YELLOW,"{YEL}",
        ORANGE,"{1}",BROWN,"{2}"
50 DATA LT. RED,"{3}",DARK GRAY,"{4}",MED GRAY,"{5}",
        LT. GREEN,"{6}"
60 DATA LT. BLUE,"{7}",LT. GRAY,"{8}"
70 FOR K=0 TO 15: READ CNAME$(K), KOMD$(K): NEXT K
75 REM BORDER COLORS
80 INPUT "BORDER^COLOR";NAME$
90 FOR K=0 TO 15
100 IF NAME$=CNAME$(K) THEN POKE 53280,K
110 NEXT K
115 REM SCREEN COLORS
120 INPUT "SCREEN^COLOR";NAME$
130 FOR K=0 TO 15
140 IF NAME$=CNAME$(K) THEN POKE 53281,K
150 NEXT K
155 REM CHARACTER COLORS
160 INPUT "CHARACTER^COLOR"; NAME$
170 FOR K=0 TO 15
180 IF NAME$=CNAME$(K) THEN PRINT KOMD$(K);
190 NEXT K
200 END
```

Modifications

If the color names are printed on the screen you can see how they are spelled and minimize input errors. Add the following lines:

```
76 PRINT "{CLR}"
77 FOR K=1 TO 15: PRINT CNAME$(K)
78 NEXT K
79 PRINT
```

Notes

Some color combinations make reading difficult, for example, a BROWN screen with RED characters. There is not enough contrast between the screen and the characters. Other unsatisfactory combinations are green characters on a blue screen or cyan characters on a yellow screen. The best way to determine which combinations look good is by experimenting.

References

The *Commodore 64 Programmer's Reference Guide* published by Commodore Business Machines, Inc. and Howard W. Sams & Co., Inc. contains additional information on Commodore color combinations. Refer to the table on page 152 that shows which color combinations to avoid and which to favor.

USING THE CASSETTE RECORDER OR DISK DRIVE

Obviously, typing long programs in this strange Commodore notation can be error-prone. Once you have a program that works, you will surely want to save it on a disk or cassette. In fact, you should save a new program after every 60 lines or so to avoid losing a large amount of work. The aim here is to avoid a disaster if the power fails, your cat discovers the power cord is a wonderful new toy, or (shudder!) you absentmindedly turn the computer off.

Using the Cassette Recorder (Datassette)

To save a program on cassette, first move the tape to a blank spot, then enter

```
SAVE "PROGNAME"
```

and press RETURN. Here PROGNAME is the name of your program (say, CIRCLE for a program that draws a circle). Simply follow the computer's instructions and wait for the recording to finish. You may verify that the program has been recorded correctly by entering the command

```
VERIFY "PROGNAME"
```

To load a program from tape, move the tape where you estimate the program is and enter

```
LOAD
```

The lack of a name indicates that the computer should load the next program it finds. You can also use

```
LOAD "PROGNAME"
```

if you are sure the recorder has not already passed the program. Be careful—if the recorder starts beyond the program, the computer will read to the end of the tape. If you are attempting to LOAD a program and you think the cassette has passed it, press RUN/STOP. This will interrupt the LOADing process, allowing you to rewind the tape and try again.

When purchasing blank cassettes, always select short ones (30 minutes or less). Searching a long tape for a program can be time-consuming. For the best results, save only one or a few programs on each side of a tape.

The Commodore tape recorder has a position counter. You should mark down its count whenever you record a program. This will allow you to position the tape with reasonable accuracy. Be sure to press the index reset button immediately after rewinding the cassette. This will ensure that you are measuring the index number consistently.

Warning: When you are not using the recorder, release the play button by pressing the stop button. Do this each time the screen shows "READY.". Following this simple rule will extend the life of the recorder.

If you buy the cassette of programs for this book, you will find that we recorded many programs on one side. This keeps the cost low but makes the tape inconvenient for repeated use. We suggest you LOAD each program you expect to use often and SAVE it on a separate tape.

Using the Disk Drive

Before saving programs on a new disk, you must prepare it for computer use (called *formatting*). To do this, insert the disk into the drive and enter:

```
OPEN 15,8,15,"N0:NAME,DD: CLOSE 15
```

You must type this line exactly as shown, except that NAME can be anything you care to call the disk (say, FIGURES for a disk with figure drawings on it) and DD can be any two digits (this is called the identification or ID code). Be particularly careful to type commas and colons (*not* semicolons) where indicated. Also note the fixed sequence N0: (the second character is a zero, not the letter O) after the opening quotation mark. Be patient; it takes the Commodore well over a minute to format a disk.

To save a program on a formatted disk, enter:

```
SAVE "PROGNAME",8
```

where PROGNAME is the name you give your program. To load a program from a disk, enter:

```
LOAD "PROGNAME",8
```

Watch the red disk light (the *drive indicator*) as well as the screen when saving or loading programs. A flashing red light indicates that something has gone wrong.

One problem with saving a program this way is that the Commodore will not overwrite an old program with the same name. Say, for example, you have a program called TRIANGLE on your disk. You may want to change or correct that program and then save the new version. The Commodore will not overwrite the old TRIANGLE; instead, it will just flash the disk light and indicate READY. If you are not watching the disk light, you may think the new version has been saved.

To force the Commodore to save a program even if it must overwrite an existing program, type @0: ahead of the program's name. Note that the second character here is zero, not the letter O. Thus, to insure that the Commodore saves the new version of TRIANGLE on the disk you must enter:

```
SAVE "@0:TRIANGLE",8
```

This forced overwriting doesn't work correctly on some Commodore disk drives. Occasionally, the computer overwrites the wrong program. If you find this happening you must use the equivalent sequence:

```
OPEN 15,8,15,"S0:TRIANGLE": CLOSE 15
SAVE "TRIANGLE",8
```

The first line erases the old TRIANGLE (S means "scratch," computer slang for delete).

To see whether a program is on a particular disk or whether it has

actually been saved, you need the disk's table of contents (or *directory*). To load the directory, enter:

```
LOAD "$",8
```

After the computer finishes loading, type LIST to show the directory. If it is long, you may need to press RUN/STOP or CTRL to keep the listing from rushing by. Holding CTRL down will make the entries appear slowly, one at a time. The list is not alphabetical; entries simply appear in the order in which they were recorded.

You can use the directory to determine the proper spelling of a program and which names are already in use. If you misspell a name when LOADing a program, the message NOT FOUND will appear and the red light will flash. If you have a program in memory, you should save it before obtaining a directory listing. This is because loading a directory (or a program) clears memory.

Handling Disks

Watch the following when using disks:

1. Be sure the disks are the right ones for the Commodore. In technical terms, they must be 5¼ inches in diameter (so-called *minidisks*), soft-sectored, single-sided, and single-density.
2. Remember to format blank disks before attempting to store programs or data on them. *Do not format a disk that has programs you want to keep.* Formatting erases all programs and data.
3. Insert disks into the drive carefully. Be sure you have the label up and toward you.
4. Never force a disk into or out of the drive.
5. Close the drive door before letting the computer use a disk. Close the door (or open it) gently; don't force it.
6. Never turn the disk drive off or remove a disk while the computer is using the drive.
7. Use the *write-protect tab* on the disk to prevent accidental loss of valuable programs. This tab is a gummed sticker that comes with the boxed disks. Place it over the square notch in the disk's cover. Note: If you try to SAVE a program on a write-protected disk, the red light will flash.

Handle disks carefully. Keep them in their covers and away from magnets, electric motors, transformers, dirt, dust, or other contaminants. Label each disk with a felt tip pen. Indicate its contents, the date you prepared

it, and the name and identification number you gave it during formatting. Store disks upright in their paper jackets; any of the widely available disk holders or boxes are suitable storing containers.

Always keep back-up copies of important disks. If, for example, you buy the disk for this book, copy the programs before using them and then write-protect the original (the *master*). You can copy a disk by LOADing programs into the computer from the master disk, then SAVEing them on formatted, blank disks.

SCREEN DIVISIONS AND NUMBERING

The final introductory material we need is a standard description of the television screen. Let us think of the screen as a grid, as shown in Figure 1–2. As we mentioned, it can hold 25 lines, each containing up to 40 characters. We will use the following numbering system to describe positions on the screen:

1. 0 to 39 for columns (the horizontal dimension) from left to right. Note that the left-most column is 0, not 1.
2. 0 to 24 for rows or lines (the vertical dimension) from bottom to top. Note again that the bottom row is 0, not 1.

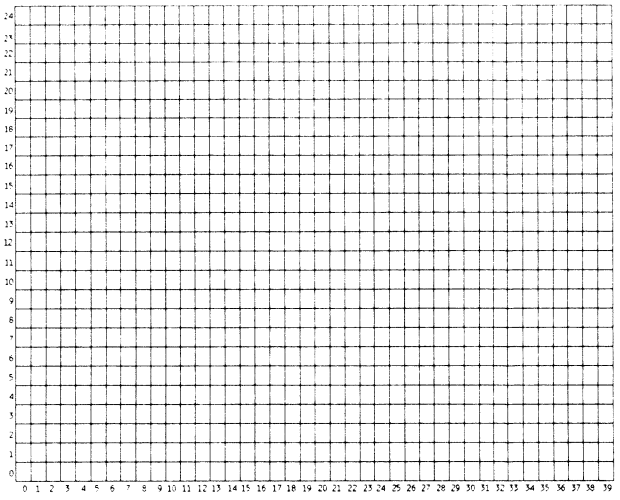


Figure 1–2 Character position chart

In this system, for example:

1. Row 0, column 39, is in the lower right-hand corner.
2. Row 12, column 19, is near the center of the screen. Row 12 is halfway between top and bottom, while column 19 is just left of center.
3. Row 24, column 39, is in the top right-hand corner.

Watch that we are placing the *origin* (row 0, column 0) in the lower left-hand corner, where it normally is on a piece of graph paper. This is different from the origin's location in most Commodores, where they generally put it in the upper left-hand corner.

2

CHARACTER DRAWINGS

The simplest way to draw a picture on a computer is as follows: (1) sketch the picture on a grid (2) examine each character position and decide which character (if any) should go there. While this approach involves a lot of manual work, it is a good place to start. We will discuss ways to let the computer do more of the work later.

The simplest implementation of this approach uses a single character. That is, all we do is decide which positions are occupied by the outline of the picture. HEART, the first program in this chapter, uses a single character to draw the outline of a heart.

To draw more detailed and more interesting pictures, we must use more characters. We can also take advantage of the Commodore 64's graphics symbols and its ability to reverse characters and change colors. Program FLAG uses color, reverse, and a few graphics symbols to draw a red, white, and blue United States flag. CARTOON uses a more extensive selection of graphics symbols to draw an elephant, AFRICA creates a solid map of Africa, and TEXAS uses lines to produce an outline map of the state of Texas.

While our primary interest here is pictures, we can use a similar approach to design business forms. Program CALENDAR illustrates this by producing a calendar for any month of any year. A modification shows how to put ruler lines on the calendar.

We can even redraw parts of a picture to produce simple animation. After all, remember that a motion picture is just a series of still photographs shown one after another. Program TREE not only draws a Christmas tree, but also places twinkling colored lights on it.

SINGLE CHARACTER DRAWINGS

Our first drawing is a simple pattern created with a single character. We use the Character Position Chart (Figure 1–2) to decide where characters should go. As Figure 2–1 shows, we first draw the heart on a grid, and then fill each space the outline crosses. We determine the TABs for each line by counting spaces. Remember that the column numbers start with 0 at the left edge and increase moving right.

Program Name: "HEART"

Purpose

Draws a heart on the screen. Figure 2–2 shows the result, using a heart as the printing character.

Techniques Demonstrated

Using TABs to position characters horizontally. Drawing with ordinary typewriter characters.

Procedure

Enter or LOAD the program and RUN it.

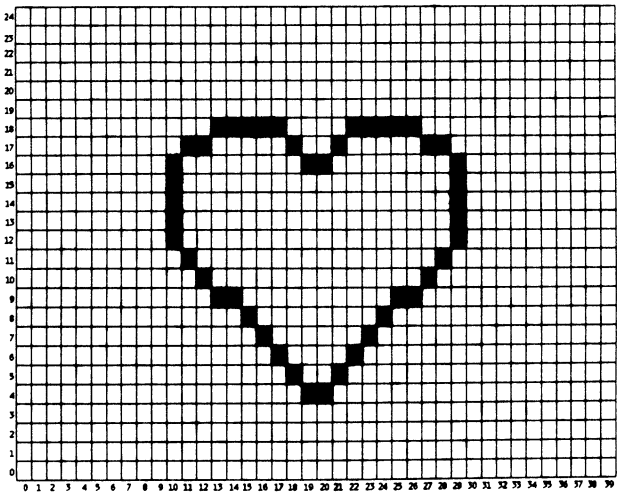


Figure 2–1 Heart drawing on the Character Position Chart

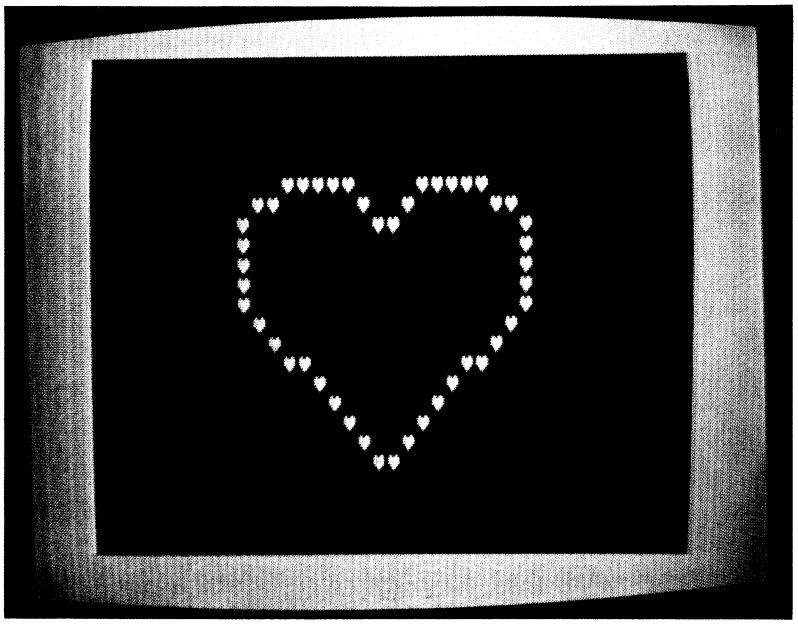


Figure 2–2 Heart drawing from Program HEART

Variables

C$: Character used in drawing
K$: Keyboard input used to terminate program

Brief Description

Lines 20 to 30 ask for the character to use in drawing the heart.
Line 40 clears the screen.
Line 50 moves the cursor down four lines to the start of the drawing.
Lines 60 to 200 draw the heart.
Line 210 waits for the user to press a key before exiting.

Listing

```
5 REM "HEART"
10 PRINT "{CLR}"
20 PRINT "WHICH^CHARACTER?"
30 INPUT C$
40 PRINT "{CLR}"
45 REM MOVE CURSOR DOWN 4 LINES
```

```
50 PRINT "{4 DOWN}"
60 PRINT TAB(12)C$C$C$C$C$ TAB(21)C$C$C$C$C$
70 PRINT TAB(10)C$C$ TAB(17)C$ TAB(20)C$ TAB(26)C$C$
80 PRINT TAB(9)C$ TAB(18)C$C$ TAB(28)C$
90 PRINT TAB(9)C$ TAB(28)C$
100 PRINT TAB(9)C$ TAB(28)C$
110 PRINT TAB(9)C$ TAB(28)C$
120 PRINT TAB(9)C$ TAB(28)C$
130 PRINT TAB(10)C$ TAB(27)C$
140 PRINT TAB(11)C$ TAB(26)C$
150 PRINT TAB(12)C$C$ TAB(24)C$C$
160 PRINT TAB(14)C$ TAB(23)C$
170 PRINT TAB(15)C$ TAB(22)C$
180 PRINT TAB(16)C$ TAB(21)C$
190 PRINT TAB(17)C$ TAB(20)C$
200 PRINT TAB(18)C$C$
210 GET K$: IF K$="" THEN 210
220 END
```

Modifications

Insert:

```
7 POKE 53281,1: PRINT "{BLK}"
```

to produce black printing on a white background.

Entering S (shifted S) in response to "WHICH CHARACTER?" will draw the outline of the heart with tiny playing-card heart symbols. Entering "{RED}S{BLK}" in response to "WHICH CHARACTERS?" will produce the same result in red instead of black. Don't forget the quotation marks; they make the color change occur when you RUN the program, instead of taking effect immediately.

We can also alternate characters to produce unusual effects. Add the following lines:

```
20 PRINT "WHICH CHARACTER?"
25 INPUT CH$(1)
30 PRINT "SECOND CHARACTER"
35 INPUT CH$(2)
36 REM START CHARACTER NUMBER AT 1
37 CNUM = 1

51 REM SELECT CHARACTER TO USE THIS TIME
52 C$=CH$(CNUM)

205 REM SWITCH CHARACTERS (#1 TO #2 OR #2 TO #1
210 CNUM=3-CNUM
215 REM KEEP DRAWING ON SCREEN FOR A WHILE
220 FOR K=1 TO 100: NEXT K
230 PRINT "{HOME}"
240 GET K$: IF K$="" THEN 50
```

These instructions make the computer alternate between using the first character and the second character to draw the heart. The key is line 210;

it sets the character number (CNUM) to 2 if it was 1 and to 1 if it was 2, thus making line 52 alternate C$ between the two characters. Line 220 simply keeps a particular drawing on the screen for a while; you can change the final value (the 100) to adjust the timing.

Try the following pair of characters:

1. FIRST CHARACTER = E
   SECOND CHARACTER = C

The heart appears to beat because of the alternation of horizontal lines at different heights. To make the beating horizontal rather than vertical, use G and H or {G} and {M} as the characters.

2. FIRST CHARACTER = "{RED}S{BLK}"
   SECOND CHARACTER = "{RED}Q{BLK}"

Now the individual tiny hearts appear to beat. Other pairs you can try include V and X, Q and W, * and ., O (letter) and 0 (zero), + and ± (shifted +), and {G} and {K}.

3. FIRST CHARACTER = "{RED}S{BLK}"
   SECOND CHARACTER = "{GRN}S{BLK}"

Now the characters stay the same, but the color changes. Try other color combinations such as {YEL} and {GRN}, {1} (orange) and {YEL}, {4} (dark gray) and {8} (light gray), and {GRN} and {6} (light green). This is a good way to show the differences among the grays and between the light and regular colors.

4. FIRST CHARACTER = "{WHT}*{BLK}"
   SECOND CHARACTER = "*"

Now the heart appears and disappears, since the first character is invisible against the white background. See what happens if you change line 7 to:

```
7 POKE 53281,0: PRINT "{WHT}"
```

and run this combination:

5. FIRST CHARACTER = "*"
   SECOND CHARACTER = "{RVS}*{OFF}"

This shows the difference between a character and its reversal. Try replacing * with Q, or W. Using {K} results in a beating heart similar to the ones we generated earlier.

We can fill the interior of the heart rather than just draw its outline. Replace lines 70 to 190 with the following:

```
65 C5$=C$+C$+C$+C$+C$
70 PRINT TAB(10)C$C5$C$C$ TAB(20)C$C5$C$C$
80 PRINT TAB(9)C$C5$C5$C5$C$C$C$C$
90 PRINT TAB(9)C$C5$C5$C5$C$C$C$C$
100 PRINT TAB(9)C$C5$C5$C5$C$C$C$C$
110 PRINT TAB(9)C$C5$C5$C5$C$C$C$C$
120 PRINT TAB(9)C$C5$C5$C5$C$C$C$C$
130 PRINT TAB(10)C$C5$C5$C5$C$C$
140 PRINT TAB(11)C$C5$C5$C$C$C$C$C$
150 PRINT TAB(12)C$C5$C5$C$C$C$
160 PRINT TAB(14)C$C5$C$C$C$C$
170 PRINT TAB(15)C$C5$C$C$
180 PRINT TAB(16)C$C$C$C$C$C$
190 PRINT TAB(17)C$C$C$C$
```

C5$ simply consists of five consecutive C$ characters; remember that + means "put one after another" (*concatenate*) when applied to strings. Note that lines 80 to 120 are identical; you can also derive the other lines from them by changing the TAB values and adjusting the character strings. You may want to define other multiple characters besides C5$.

RUN the revised program first with * as the character and then with S. Both drawings are unimpressive, since the interiors contain a lot of blank space and the borders are no longer well-defined. You can fill the interior by using "{RVS}^{OFF}" as the character, but this makes the border very irregular. If you do this, don't forget the quotation marks; they make {RVS} take effect in the program, rather than on the data line.

One way to improve the picture is to use different interior and border characters. Let us introduce a border character CB$. Now revise lines 60 to 200 as follows:

```
60 PRINT TAB(12)CB$CB$CB$CB$CB$ TAB(21)CB$CB$CB$CB$CB$
70 PRINT TAB(10)CB$C5$C$CB$ TAB(20)CB$C5$C$CB$
80 PRINT TAB(9)CB$C5$C5$C5$C$C$C$CB$
90 PRINT TAB(9)CB$C5$C5$C5$C$C$C$CB$
100 PRINT TAB(9)CB$C5$C5$C5$C$C$C$CB$
110 PRINT TAB(9)CB$C5$C5$C5$C$C$C$CB$
120 PRINT TAB(9)CB$C5$C5$C5$C$C$C$CB$
130 PRINT TAB(10)CB$C5$C5$C5$C$CB$
140 PRINT TAB(11)CB$C5$C5$C$C$C$C$CB$
150 PRINT TAB(12)CB$C5$C5$C$C$CB$
160 PRINT TAB(14)CB$C5$C$C$C$CB$
170 PRINT TAB(15)CB$C5$C$CB$
180 PRINT TAB(16)CB$C$C$C$C$CB$
190 PRINT TAB(17)CB$C$C$CB$
200 PRINT TAB(18)CB$CB$
```

In this version, most lines (90 to 190) simply start and end with a border character. We must also change the lines that let the user choose a character to:

```
20 PRINT "INTERIOR CHARACTER"
32 PRINT "BORDER CHARACTER"
34 INPUT CB$
```

Run the two-character version using * as the interior character and S as the border character. Now you can see the heart shape more clearly again. Another good combination is . as the interior and * as the border. As before, you can fill the figure by using a reversed space ("{RVS}^{OFF}") as the interior character.

Notes

HEART shows how to draw pictures on computers (and printers) that lack graphics characters. A generation of students and programmers has used this approach to produce cartoons, greeting cards, and even human figures (including pin-ups). Of course, the pictures only look good from a distance. Up close, you can see the individual characters used to draw lines and fill in solid areas.

Clearly, drawings that use only one or two typed characters are primitive at best. You cannot get high precision or fine detail this way. Still, it is amazing (and fun) to see what you can produce with a little persistence and imagination.

Despite this approach's limitations, it has practical uses besides decorating programmers' offices. For example, most computer facilities use a variation of it to put large printed dates, names, and account numbers on output for identification purposes. It is also a quick way to produce rough plots, surface maps, and charts.

When entering programs like HEART, you should take advantage of the fact that the Commodore lets you copy lines. For example, lines 90 through 120 are identical. You can enter them as follows:

1. First type line 90 and press RETURN to enter it into memory.
2. Press {UP} (SHIFT and the CRSR key with up and down arrows) to move the cursor back up to the 9 in line 90.
3. Press {INST} and 1 to insert a 1. Then type 0 to finish changing the line number from 90 to 100. Press RETURN to enter line 100 into memory.
4. Press {UP} and {RIGHT} to move the cursor to the left-most 0 on line 100.
5. Press 1 to change the line number from 100 to 110, and RETURN to enter line 110 into memory.

You can continue this way through line 190. The only difference is that you must change the TAB values in lines 130, 140, and 160 through 190.

Line 150 also requires two additional C$'s. To enter it, first make an extra copy of line 140 by listing it. Then you can convert the copy into line 150 while still retaining the original for use in entering lines 160 through 190.

Note that only the latest line appears on the screen. The others, however, are in memory, and you can see them all by entering LIST. This approach saves a lot of typing when programs have many repetitive or similar lines.

Be sure to press Return after completing each revised line. If you proceed without pressing Return (e.g., move the cursor or clear the screen), the revised line will not be entered into memory. Forgetting Return is an easy error to make when you are revising or renumbering many lines at one time. Remember that you must press Return after every program line, regardless of whether it is a new line, a copy, or a revision.

SPC is often a convenient alternative to TAB. It also moves the cursor right, but by a specified number of columns rather than to a specified column. For example, SPC(2) moves the cursor right two columns, as contrasted to TAB(2), which moves the cursor to column 2. The difference is that SPC is *relative* (that is, it moves the cursor a specified distance from its current position), whereas TAB is *absolute* (that is, it moves the cursor to a specified destination). Despite its name, SPC does not print spaces; it just moves the cursor.

The spacing in most program lines is not significant. We have often typed extra spaces just to make lines more readable. You could, for example, type line 60 as:

```
60 PRINTTAB(7)C$C$TAB(12)C$C$
```

This form uses less computer memory but is difficult to read. Note, however, that you cannot put spaces inside words (e.g., PR INT or T AB), inside names (e.g., C $ or C NUM), or between a function such as TAB and the left parenthesis that encloses its argument.

ADDING GRAPHICS, COLOR, AND REVERSE

The addition of color changes, graphics symbols, and reversed characters lets us produce a much wider variety of pictures than we can draw with one or two typewriter characters. FLAG uses only a few of the Commodore's capabilities to draw a red, white, and blue United States' flag.

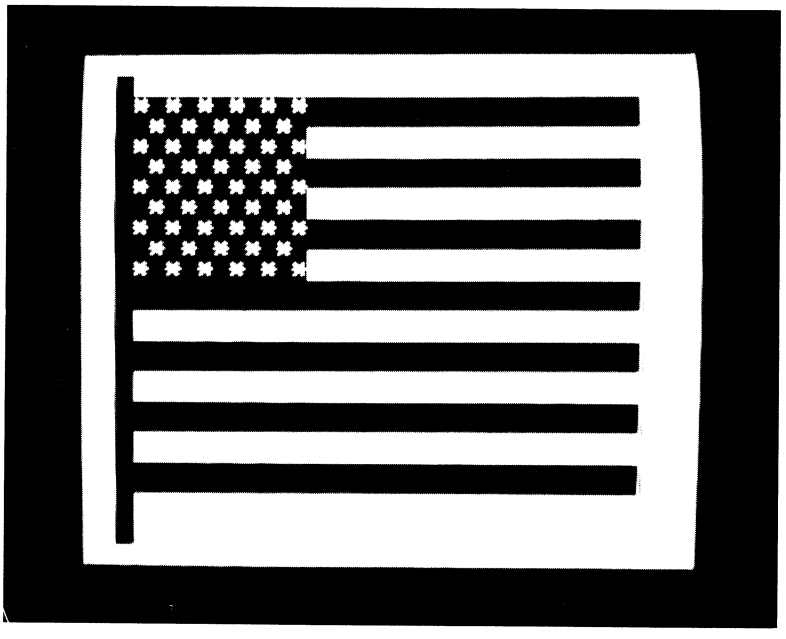


Figure 2–3 Drawing of the United States' flag

Program Name: "FLAG"

Purpose

Draws the United States' flag, as shown in Figure 2–3.

Techniques Demonstrated

Using normal and reversed graphics symbols to draw a figure with repetitive features.

Procedure

LOAD and RUN the program and watch the drawing of the flag.

Variables

F: Fringe counter
K$: Keyboard input used to terminate program
L: Line counter
P: Pole counter

S: Star counter
S$: Partial star field

Brief Description:

Line 10 clears the screen and makes the background white.
Lines 20 to 60 draw the red and white stripes.
Lines 70 to 100 draw the flagpole.
Lines 110 to 150 draw the fringe that sets the flag off from the background.
Line 170 waits for the user to press a key.
Lines 180 to 200 clear the screen, restore the normal screen and character colors, and end the program.
Lines 300 to 360 draw the star field using blue reversed asterisks and solid spaces.
Lines 400 to 460 draw the star field using different segments of a line of asterisks and spaces.

Listing

```
5 REM "FLAG"
7 POKE 53281,1
10 PRINT "{CLR}"
15 REM DRAW STRIPES
20 FOR L=0 TO 18 STEP 3
30 PRINT: REM WHITE STRIPE
40 PRINT"^^^{RED}{RVS}{34 ^}": REM RED STRIPES
50 PRINT"^^^{RED}{RVS}{34 I}": REM RED STRIPE/WHITE STRIPE
60 NEXT L
65 REM DRAW POLE
70 PRINT "{HOME}{YEL}^^Q"
80 FOR P=1 TO 23
90 PRINT "^^{RVS}{BLK}^"
100 NEXT P
105 REM DRAW FRINGE
110 PRINT "{HOME}{DOWN}"
120 FOR F=1 TO 19
130 PRINT TAB(36)"{YEL}{K}"
140 NEXT F
150 PRINT TAB(36)"{V}"
160 GOSUB 300: REM PRINT STARS
170 GET K$: IF K$="" THEN 170
180 POKE 53280,14: POKE 53281,6
190 PRINT "{CLR}{7}"
200 END
210 POKE 53280,14: POKE 53281,6
220 PRINT "{CLR}{7}"
230 END

295 REM STARS VERSION 1
300 PRINT "{HOME}{2 DOWN}" TAB(3)
310 FOR S=1 TO 99
320 IF INT (S/2)*2<>S THEN PRINT "{BLU}{RVS}*";
```

```
330 IF INT (S/2)*2=S THEN PRINT "{BLU}{RVS}^";
340 IF INT (S/11)*11=S THEN PRINT: PRINT TAB(3)
350 NEXT S
360 RETURN

395 REM STARS VERSION 2
400 LET S$="^*^*^*^*^*^*^*"
405 PRINT "{HOME}{DOWN}{3 RIGHT}"
410 PRINT "{HOME}{DOWN}{3 RIGHT}"
420 FOR L=1 TO 9
430 LET B=1+L-2*INT(L/2)
440 PRINT TAB(3)"{BLU}{RVS}"MID$(S$,B,11)
450 NEXT L
460 RETURN
```

Modifications

Change line 160 from GOSUB 300 to GOSUB 400 to demonstrate the line-at-a-time method for drawing the star field. Note how much faster this method is.

Notes

Neither the printed nor the screen listing of FLAG gives the reader much help in visualizing the actual drawing. In fact, it is even difficult to tell how long the lines will be when they contain screen controls (such as [CLR]), color changes, and reversal commands. The effects of color changes and reversals on characters are also difficult to determine. All this points out the importance of sketching a picture on graph paper before attempting to draw it on the computer.

Lines 320 and 330 determine when to print a star and when to print a solid blue space. The star field is a rectangle 11 columns wide and 9 lines high, containing a total of 99 characters. Thus, the program draws stars using a FOR . . . NEXT loop that steps S from 1 to 99. Examining the field from left to right and top to bottom shows that every other character position contains a star. If we number characters starting with 1 in the top left-hand corner, we see that all odd numbers should be stars (*). We can determine if S is even or odd by calculating INT(S/2)*2. The result is equal to S only if S is even. Remember that INT produces the largest whole number (integer) less than its argument. Line 340 uses the same approach to determine when the computer has reached the end of a line in the star field. This occurs when S is divisible by 11, the field's width.

In the second method of drawing the star field, line 440 prints the stars. Examining the field again shows that it consists of nine lines with two alternating patterns:

" ■*■*■*■*■*■ " and " *■*■*■*■*■*■*"

These patterns correspond to characters 1 through 11 and 2 through 12 of S$, respectively. Line 430 sets B (beginning character) to 1 when the line number is even and to 2 when it is odd. This selects the appropriate part of S$ in line 440.

References

The Hammond World Atlas, International Edition (Hammond, 1975), contains pictures of the flags of many countries.

HIGHER-RESOLUTION PICTURES

All pictures have resolution (that is, the size of the smallest feature you can clearly distinguish). In a high-resolution drawing or photograph, you can see tiny details of a figure, object, or scene. In a low-resolution version, you can see only general shapes; details are represented as dots or shaded areas. Note the difference, for example, in what you see in a photograph of a person, animal, or landscape (high resolution), an artist's rendition (medium resolution), and a rough pencil sketch (low resolution).

Drawing with a typewriter character as we did in program HEART results in a picture with very low resolution. You can make out the general shape of a figure, but it does not look smooth and regular. The lines are broad and jagged, and the curves are obviously just connected straight segments. To see the resolution clearly, use an inverted space as the character in HEART. The result looks like a primitive carving.

The problem is that the character spaces are much larger than the smallest features a person can distinguish at close range. Note that human abilities and expectations are the key here. Ultimately, of course, all pictures are irregular. Even photographs and drawings made with fine-point pens will look ragged and uneven under a magnifying glass or microscope. Drawings look crude to us, however, if their resolution is lower than what we can normally see.

One way to improve a picture's resolution is to reduce the size of the individual picture elements. Thus, in our case we must work with less than a character space at a time. For example, say we could divide each character space into four parts as shown in Figure 2–4. By considering every possible combination of light and dark quarter-spaces, we come up with the 16 patterns shown in Figure 2–5. Four of these have one quarter dark, six have two quarters dark, four have three quarters dark, one is all light, and one is all dark.

Some of these patterns are left-hand graphics symbols on the Commodore keyboard. For example, the D key has the lower right-hand quarter space dark and all others light. The other patterns with one quarter space dark are on the F, C, and V keys. The B key has the upper left-hand and

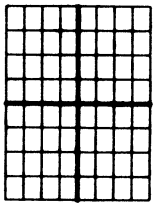


Figure 2–4 Character space eight-by-eight grid divided into four parts

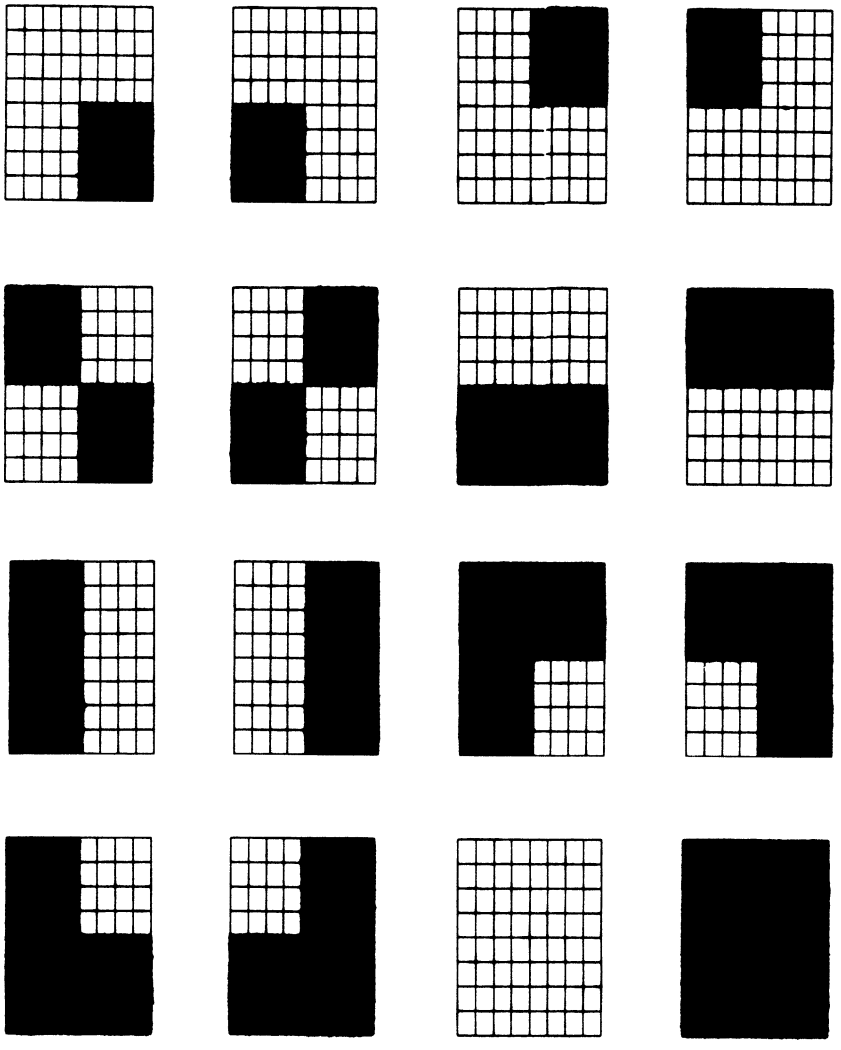


Figure 2–5 16 combinations of quarter spaces

lower right-hand quarter spaces dark, while the I and K keys have the bottom and left halves dark, respectively. Of course, all quarters light is just a space character. Thus 8 of the 16 patterns are readily available.

How do we obtain the others? We can form them by reversing the keyboard characters. Commodore-D reversed, for example, has the lower right-hand quarter space light and the others dark. Reversing Commodore-F, C, and V provides the other patterns with three quarter spaces dark, while a reversed space has all quarters dark, and reversed Commodore-B has dark upper right-hand and lower left-hand quarter spaces. Similarly, reversing Commodore-I makes the top half of the character space dark, and reversing Commodore-K makes the right half dark. Figure 2–6 summarizes how to obtain the patterns required to draw with quarter-space resolution.

Program CARTOON illustrates the drawing of pictures with quarter-space resolution. As you can see in Figure 2–7 and 2–9, we first trace the cartoon figure on the Character Position Chart. A grid divides each character space into four parts. We then decide which graphics symbol best represents each character space.

Program Name: "CARTOON"

Purpose

Draws a cartoon figure (see Figure 2–8).

Techniques Demonstrated

Drawing complex pictures with quarter-space resolution using graphics symbols. This program draws in white, so be sure the background color provides a contrast. If in doubt, press RUN/STOP and RESTORE.

Procedure

LOAD and RUN the program. Press the space bar to exit.

Variables

K$: Keyboard input used to terminate program

Brief Description

Line 10 clears the screen.

Line 20 moves the cursor down three lines and makes the printing color white.

Lines 30 to 180 draw the cartoon.

Line 190 waits for the user to press a key.

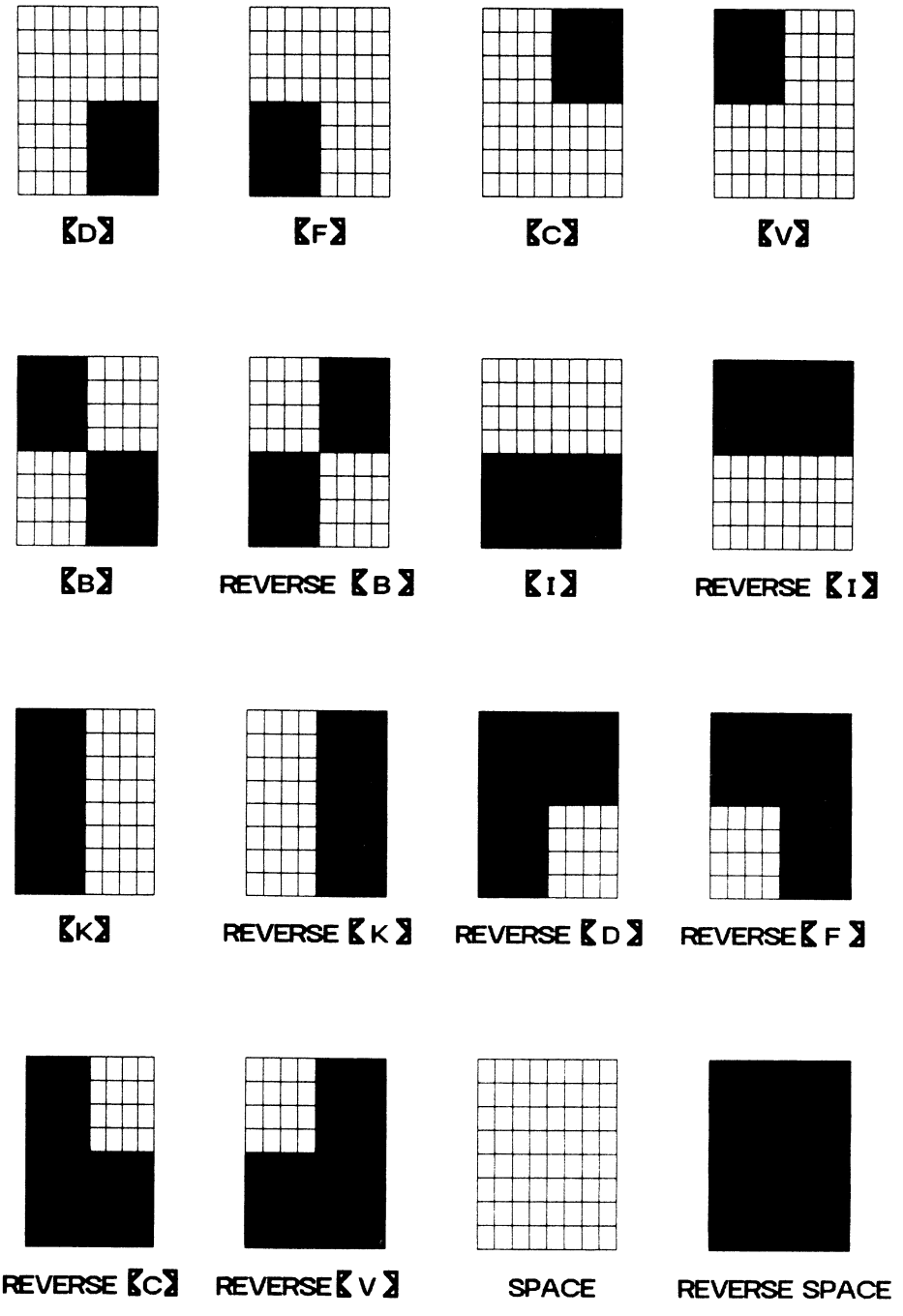


Figure 2–6 Graphics symbols required to draw with quarter-space resolution

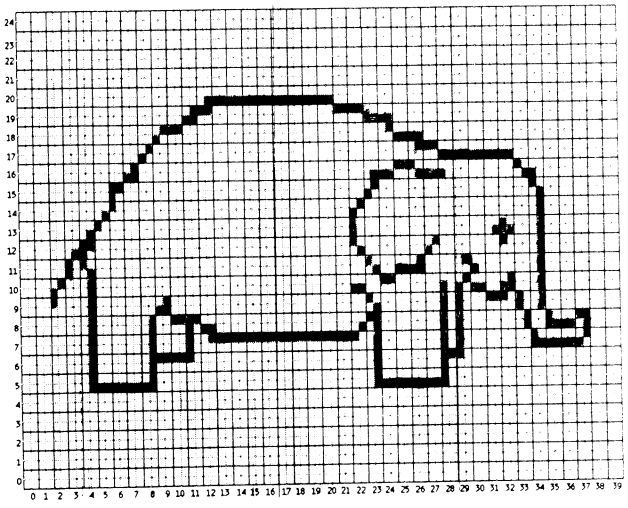


Figure 2–7 Cartoon figure of an elephant drawn on the character position chart

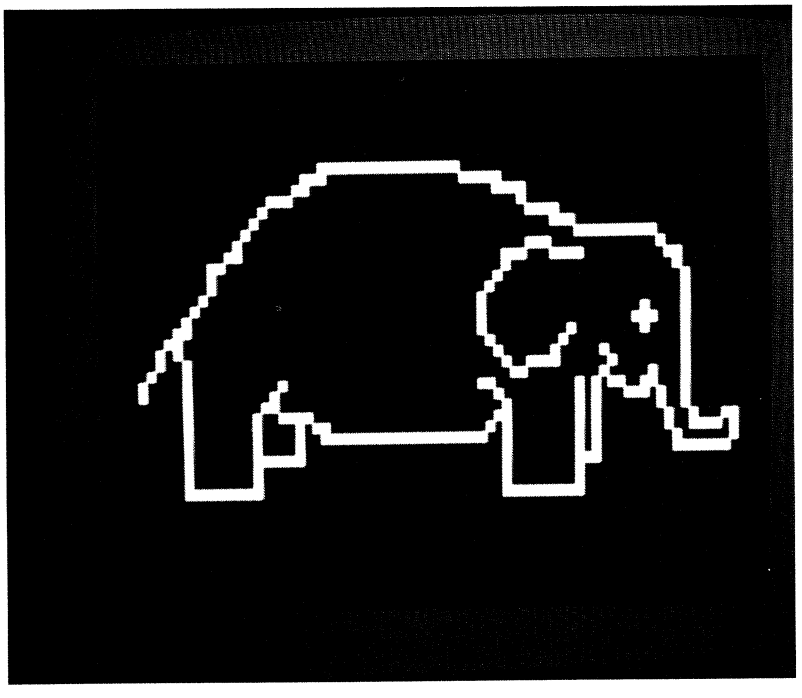


Figure 2–8 Cartoon elephant drawn by program CARTOON

Listing

```
5 REM "CARTOON"
10 PRINT "{CLR}"
15 REM MOVE CURSOR DOWN 3 LINES
20 PRINT "{3 DOWN}{WHT}";
25 REM DRAW CARTOON
30 PRINT TAB(12)"{D}{8 I}"
40 PRINT TAB(11)"{RVS}{V}{I}" TAB(21)"{2 I}{C}{OFF}{I}"
50 PRINT TAB(9)"{RVS}{B}{I}" TAB(24)"{OFF}{C}{2 I}"
60 PRINT TAB(8)"{RVS}{B}" TAB(26)"{OFF}{C}{RVS}{I}{OFF}{5 I}"
70 PRINT TAB(7)"{RVS}{V}" TAB(23)"{OFF}{D}{I}{RVS}{I}{OFF}{B}{I}{F}"
          TAB(33)"{B}{F}"
80 PRINT TAB(6)"{RVS}{D}" TAB(23)"{B}" TAB(34)"{OFF}{B}"
90 PRINT TAB(5)"{D}{V}" TAB(22)"{RVS}{B}" TAB(34)"{K}"
100 PRINT TAB(4)"{D}{V}" TAB(22)"{K}" TAB(31)"{D}{RVS}{C}{OFF}^{RVS}{K}"
110 PRINT TAB(3)"{D}{RVS}{D}" TAB(22)"{OFF}{B}^^^^{RVS}{B}"
          TAB(32)"{OFF}{V}^{RVS}{K}"
120 PRINT TAB(3)"{K}{B}" TAB(23)"{B}^{I}{RVS}{V}{OFF}^^{C}{F}"
          TAB(34)"{RVS}{K}"
130 PRINT TAB(2)"{RVS}{B}{OFF}^{RVS}{K}" TAB(10)"{OFF}{F}"
          TAB(22)"{I}^{RVS}{I}"
          TAB(28)"{OFF}{F}{RVS}{B}{OFF}{I}^{RVS}{V}{OFF}^{RVS}{K}"
140 PRINT TAB(2)"{V}^{RVS}{K}" TAB(9)"{V}" TAB(23)"{OFF}{B}"
          TAB(28)"{2 K}^{RVS}{I}{OFF}{V}{K}{RVS}{K}"
150 PRINT TAB(4)"{RVS}{K}" TAB(8)"{K}{OFF}^{RVS}{I}{D}{OFF}{I}"
          TAB(22)"{D}{RVS}{F}" TAB(28)"{OFF}{2 K}";
155 PRINT TAB(33)"{RVS}{K}{OFF}^{RVS}{C}{OFF}{I}{RVS}{F}"
160 PRINT TAB(4)"{RVS}{K}"
          TAB(8)"{RVS}{K}{OFF}^^{K}{C}{RVS}{9 I}{OFF}{V}{RVS}{K}";
165 PRINT TAB(28)"{OFF}{2 K}" TAB(34)"{RVS}{C}{OFF}{2 I}{RVS}{B}"
170 PRINT TAB(4)"{RVS}{K}" TAB(8)"{RVS}{K}{2 I}{OFF}{V}"
          TAB(23)"{RVS}{K}" TAB(28)"{D}{OFF}{V}"
180 PRINT TAB(4)"{RVS}{K}{OFF}{3 I}{RVS}{V}" TAB(23)"{K}{OFF}{4 I}{K}"
190 GET K$: IF K$="" THEN 190
200 POKE 53280,14: POKE 53281,6
210 PRINT "{CLR}{7}"
220 END
```

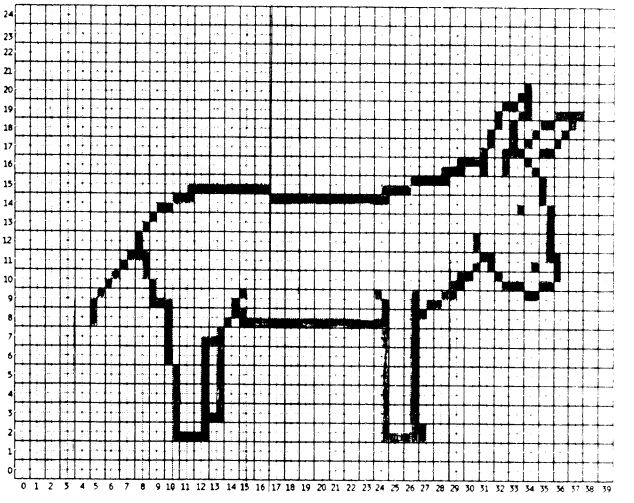


Figure 2–9 Cartoon figure of a donkey on the character position chart

Modifications

Now that we have drawn an elephant, it is only fair to give a donkey equal time. Use Figure 2–9 to create a cartoon of the donkey.

DRAWING WITH RECTANGLES AND TRIANGLES

The program AFRICA uses TABs and a variety of graphics symbols to draw a map of Africa.

Program Name: "AFRICA"

Purpose

Draws a map of Africa (see Figure 2–10).

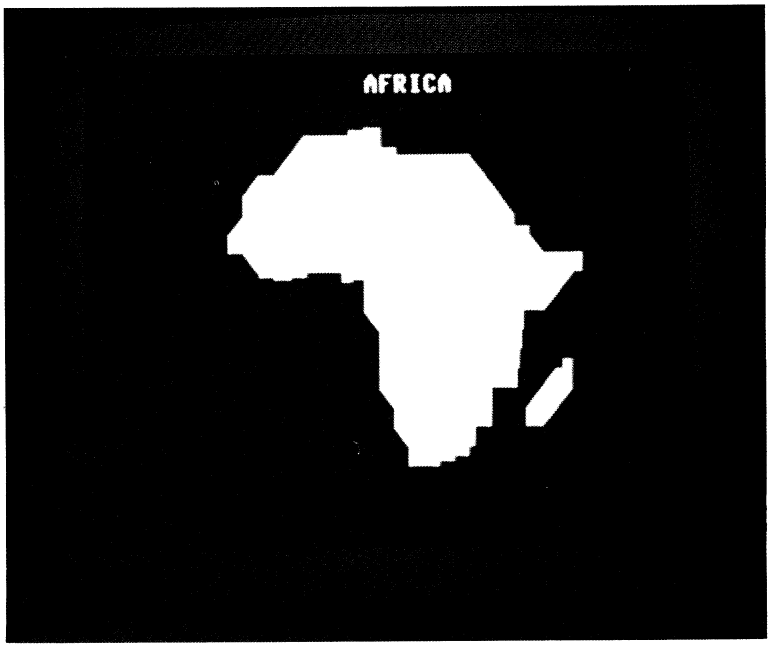


Figure 2–10 Drawing of Africa by Program AFRICA

Techniques Demonstrated

Using rectangular and triangular graphics characters to draw complex figures.

Procedure

LOAD and RUN the program. Press the space bar to exit.

Variables

K$: Keyboard input used to terminate program.

Brief Description

Line 10 clears the screen and prints AFRICA in white.
Line 20 makes the printing color yellow.
Lines 30 to 190 draw a map of Africa.
Line 200 waits for the user to press a key.
Lines 210 to 220 restore the screen and border colors, clear the screen, and set the printing color back to light blue.

Listing

```
5 REM "AFRICA"
10 PRINT "{CLR}{DOWN}"TAB(18)"{WHT}AFRICA"
20 PRINT "{YEL}"
30 PRINT TAB(17);"{P}{O}"
40 PRINT TAB(13);"{RVS}£{5 ^}{OFF}{O}"
50 PRINT TAB(12);"{RVS}£{12 ^}{*}"
60 PRINT TAB(10);"{RVS}£{15 ^}{*}"
70 PRINT TAB(10);"{RVS}{17 ^}{*}"
80 PRINT TAB(9);"{RVS}£{18 ^}{OFF}{O}"
90 PRINT TAB(9);"{RVS}{20 ^}{*}"
100 PRINT TAB(10);"{*}{RVS}{21 ^}{L}"
110 PRINT TAB(11);"{Y}{U}{Y}^^{C}{U}{RVS}{13 ^}{OFF}_"
120 PRINT TAB(18);"{RVS}{12 ^}{OFF}£"
130 PRINT TAB(18);"{*}{RVS}{9 ^}{L}"
140 PRINT TAB(19);"{RVS}{9 ^}{OFF}{K}"
150 PRINT TAB(19);"{RVS}{9 ^}{OFF}{J}^^{D}"
160 PRINT TAB(19);"{RVS}{9 ^}{OFF}{H}^{RVS}£^"
170 PRINT TAB(19);"{*}{RVS}{6 ^}{OFF}{K}^^{RVS}£^{OFF}£"
180 PRINT TAB(20);"{RVS}{6 ^}{OFF}{K}^^{RVS}^{OFF}_"
190 PRINT TAB(20);"{*}{RVS}^^^^{OFF}{J}"
200 PRINT TAB(21);"{RVS}^^{P}{I}"
210 GET K$: IF K$="" THEN 210
220 POKE 53280,14: POKE 53281,6
230 PRINT "{CLR}{7}"
240 END
```

Modifications

To draw the Equator across the map of Africa, add:

205 PRINT "{HOME}{12 DOWN}" TAB(18)"{RVS}**********" (*14 shifted asterisks*).

Note that the characters printed by line 195 overwrite ones PRINTed earlier in the program. You may want to add other features such as locations of major cities and rivers, or label the oceans.

Be careful when adding features to a complicated picture. If you print in or beyond the right-most column or on the bottom line, you may make the screen scroll. You cannot make it scroll back! The safest approach is to keep away from the right and bottom boundaries. Another problem is that you may accidentally erase part of the picture. This is particularly likely to happen with spaces that may not appear to be occupied; that is, the characters in them are quarter spaces, horizontal or vertical lines near the boundaries, or small rectangles. One way to limit the damage is to save the original picture-drawing program on disk or tape; then you can always reload it if you make a series of errors.

Notes

We form most of the map of Africa with different characters than the quarter spaces used in CARTOON. We have resolved some parts of the picture down to an eighth of a space. We do this by representing the dark part of a character space as a rectangle extending inward from one border. For example, Commodore-G darkens the left-most eighth of a space, Commodore-H the left-most quarter, Commodore-J the left-most three eighths, Commodore-K the left half, Commodore-L the right-most three eighths, Commodore-N the right-most quarter, and Commodore-M the right-most eighth. Similarly, Commodore-@ darkens the bottom eighth, Commodore-P the bottom quarter, Commodore-O the bottom three eighths, Commodore-I the bottom half, Commodore-U the top three eighths, Commodore-Y the top quarter, and Commodore-T the top eighth.

To darken the remaining sections of a character space, we must use reversed characters. For example, reversed Commodore-N darkens the left-most three quarters of a space, reversed Commodore-K darkens the right half, and reversed Commodore-O darkens the top five eighths. This is a strange resolution, since it is one eighth of a character space in one dimension, but an entire space in the other. Figure 2–11 lists the characters that provide one-eighth resolution horizontally, while Figure 2–12 lists the ones that provide one-eighth resolution vertically.

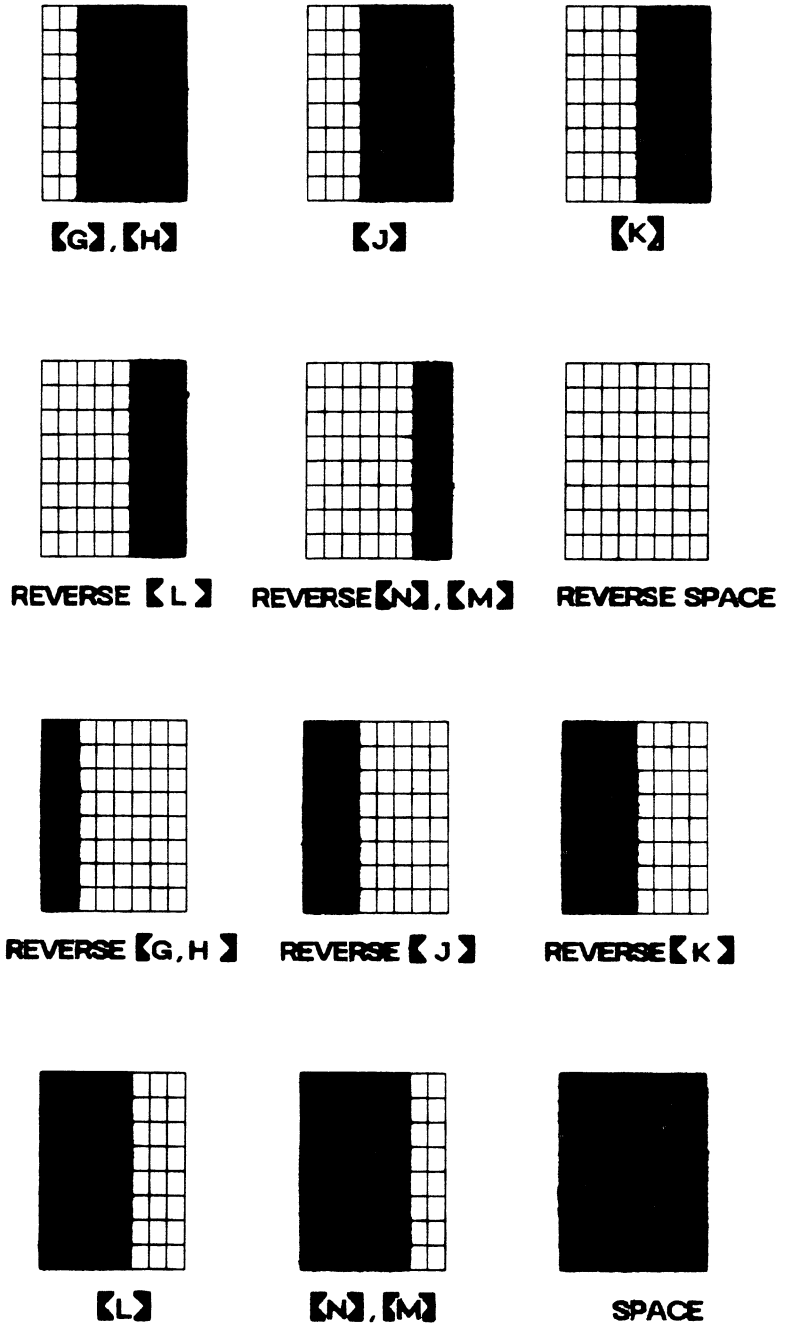


Figure 2–11 Graphics symbols required to draw with one-eighth space resolution horizontally

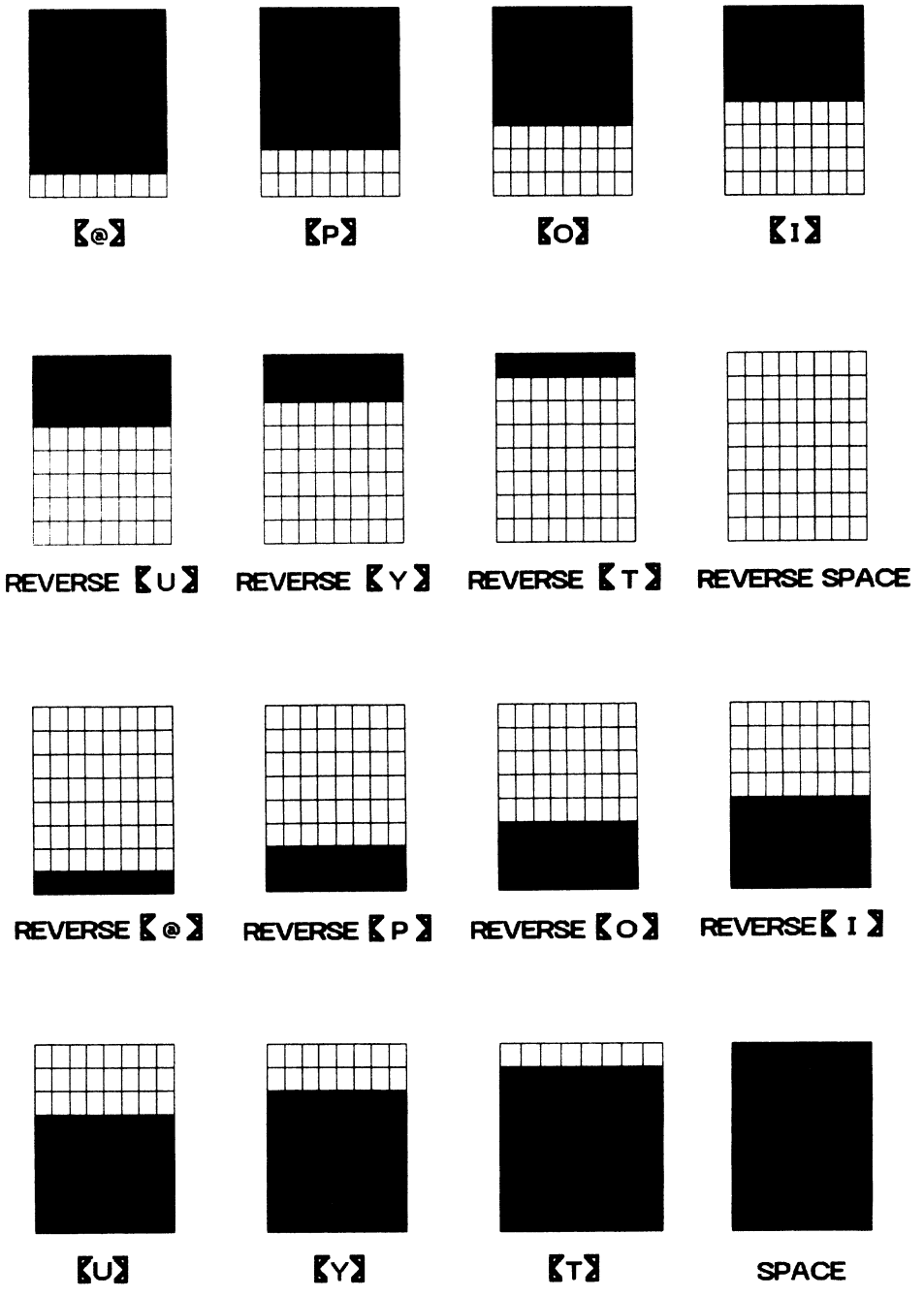


Figure 2–12 Graphics symbols required to draw with one-eighth space resolution vertically

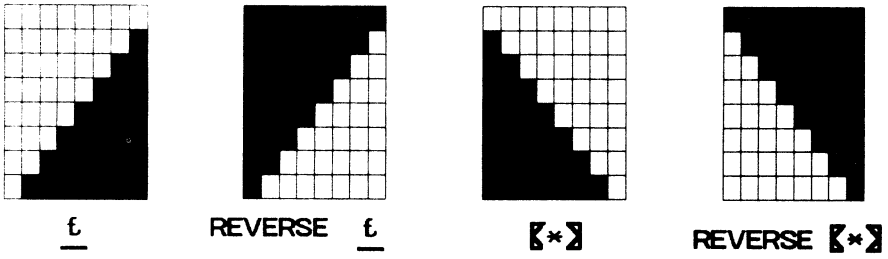


Figure 2–13 Graphics symbols required to draw with half-space triangles

We can also darken any of the four triangular half-spaces created by drawing diagonals across the space. The right triangle defined by the left and top borders is the right-hand graphics symbol on the British pound sign key. The one defined by the right and top borders is the left-hand graphics symbol on the asterisk key. We can obtain the other two triangles by reversing these two. Figure 2–13 summarizes the production of half-space triangles. Lines 40 through 130 and 170 through 190 use these symbols.

References

The *Hammond World Atlas, International Edition* (Hammond, 1975), contains maps showing outlines of countries and locations of cities.

HIGHER-RESOLUTION OUTLINES

The program "TEXAS" uses horizontal, vertical, and diagonal lines to draw an outline of the state of Texas.

Program Name: "TEXAS"

Purpose

Draws the outline of Texas (see Figure 2–14).

Techniques Demonstrated

Using horizontal, vertical, and diagonal lines to draw an outline map.

Procedure

LOAD and RUN the program.

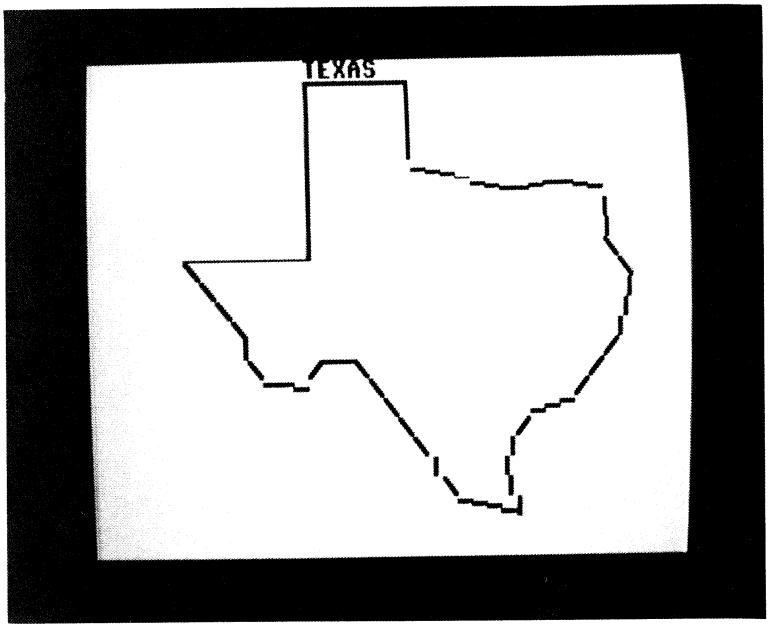


Figure 2–14 Outline of Texas by Program TEXAS

Variables

K$: Keyboard input used to terminate the program.

Brief Description

Line 10 clears the screen.
Line 20 makes the border black and the background white.
Lines 30 to 240 draw the outline of Texas in black.
Line 250 waits for the user to press a key.
Lines 260 and 270 restore the standard colors and clear the screen.

Listing

```
5 REM "TEXAS"
10 PRINT "{CLR}";
20 POKE 53280,0: POKE 53281,1
30 PRINT TAB(14)"{BLK}TEXAS"
40 PRINT TAB(14)"O{5 Y}P"
50 PRINT TAB(14)"{G}{5 ^}{N}"
55 PRINT TAB(14)"{G}{5 ^}{N}"
60 PRINT TAB(14)"{G}{5 ^}{N}"
70 PRINT TAB(14)"{G}{6 ^}CFR{@}"
80 PRINT TAB(14)"{G}"SPC(10)"EDCCDEEDC"
```

```
90 PRINT TAB(14)"{G}"SPC(19)"{G}"
100 PRINT TAB(14)"{G}" SPC(19)"T"
110 PRINT TAB(6)"{8 @}{G}" SPC(19)"M"
120 PRINT TAB(6)"M" SPC(28)"M"
130 PRINT TAB(7)"M" SPC(27)"Y"
140 PRINT TAB(8)"M" SPC(26)"B"
150 PRINT TAB(9)"M" SPC(25)"{G}"
160 PRINT TAB(9)"{M}" SPC(24)"N"
170 PRINT TAB(10)"M^^^N{2 Y}M" SPC(15)"N"
180 PRINT TAB(11)"EE*^^^^M" SPC(13)"N"
190 PRINT TAB(19)"M" SPC(9)"RCE"
200 PRINT TAB(20)"M" SPC(7)"N"
210 PRINT TAB(21)"M" SPC(5)"Y"
220 PRINT TAB(22)"-" SPC(4)"G"
230 PRINT TAB(23)"M^^^H"
240 PRINT TAB(24)"EDCRT"
250 GET K$: IF K$="" THEN 250
260 POKE 53280,14: POKE 53281,6
270 PRINT "{CLR}{7}"
280 END
```

Modifications

To put some major cities on the map, insert the following lines:

```
250 PRINT "SPACE^BAR^FOR^CITIES"
260 GET K$: IF K$="" THEN 260
265 REM CHANGE MESSAGE
270 PRINT "{UP}SPACE^BAR^TO^EXIT^^^"
280 PRINT "{HOME}{4 DOWN}" TAB(16) "{YEL}QAMARILLO"
290 PRINT "{HOME}{13 DOWN}" TAB(26)"{RED}*AUSTIN"
300 PRINT "{HOME}{14 DOWN}" TAB(25)"{CYN}HOUSTONQ"
310 PRINT "{HOME}{8 DOWN}" TAB(27)"{GRN}Q^DALLAS"
320 PRINT "{HOME}{10 DOWN}" TAB(7)"{PUR}QEL^PASO"
330 GET K$: IF K$="" THEN 330
340 POKE 53280,14: POKE 53281,6
350 PRINT "{CLR}{7}"
360 END
```

The asterisk for Austin indicates that it is the state capitol.

If you don't mind a crowded map, you can add even more cities with:

```
321 REM PRINT ADDITIONAL CITIES
322 PRINT "{HOME}{7 DOWN}" TAB(16)"{BLU}QLUBBOCK"
323 PRINT "{HOME}{10 DOWN}" TAB(27)"{1}QWACO"
324 PRINT "{HOME}{15 DOWN}" TAB(24)"{3}QSAN^ANTONIO"
325 PRINT "{HOME}{22 DOWN}" TAB(27)"{2}QBROWNSVILLE"
326 PRINT "{HOME}{8 DOWN}" TAB(16)"{4}FT.^WORTHQ"
```

We could use the same approach to show the locations of oil fields, recreational areas, battles, or historical sites.

We can even use the map as a background for an interactive quiz. The following additions test whether you know the name of the capitol city of Texas.

```
245 REM MARK LOCATION OF STATE CAPITOL
250 PRINT "{HOME}{13 DOWN}" TAB(26)"{RED}{RVS}*{BLK}"
260 PRINT "{HOME}{24 DOWN}";
265 REM BLANK PROMPT LINE
270 PRINT "{UP}{38 ^}"
275 REM ASK FOR ANSWER
280 INPUT "{UP}CAPITOL";CN$
285 REM CHECK ANSWER
290 IF CN$="AUSTIN" THEN 320
295 REM WRONG ANSWER - TRY AGAIN
300 PRINT "{UP}WRONG-TRY^AGAIN{24 ^}"
310 FOR D=1 TO 1000: NEXT D: GOTO 270
320 PRINT "{UP}CORRECT-^SPACE^TO^EXIT"
330 PRINT "{HOME}{13 DOWN}" TAB(26)"{RED}*AUSTIN"
340 GET K$: IF K$="" THEN 340
350 POKE 53280,14: POKE 53281,6
360 PRINT "{CLR}{7}"
370 END
```

Line 250 prints a reversed asterisk at Austin's location. Line 270 clears the prompt line before asking for the answer; it also erases the computer's comment before repeating the question. The delay in line 310 gives you time to see the computer's WRONG–TRY AGAIN response before the program proceeds to line 270.

Enter your answer carefully. This program does not tolerate spelling or typing errors. It will reject entries such as AUSTEN (with Pride and Prejudice), AURSTIN, or AWSTIN. You may want to add a limit on the number of guesses. This will provide an escape route for those who can't spell or have no background in American geography. Of course, you can always exit with the RUN/STOP key.

Notes

TEXAS draws the state's outline using graphics characters that consist only of lines. The available characters are:

1. Eight vertical lines (see Figure 2–15).
2. Eight horizontal lines (see Figure 2–16).
3. Three diagonals (see Figure 2–17).
4. Four half-space corners—that is, characters consisting of a vertical line from the center to an edge and a horizontal line from the center to an edge (see Figure 2–17).
5. Four full-space corners. That is, characters consisting of a vertical line along one edge, connected to a horizontal line along another edge (see Figure 2–17).
6. Four rounded corners. That is, characters consisting of quarter-circles centered at a corner of the character space (see Figure 2–17).

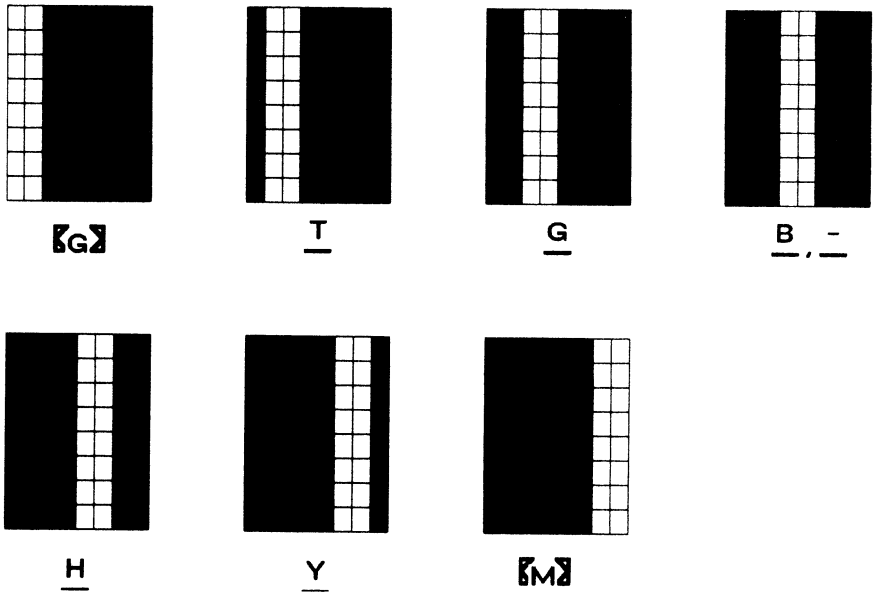


Figure 2–15 Graphics symbols that draw vertical lines at one-eighth space resolution

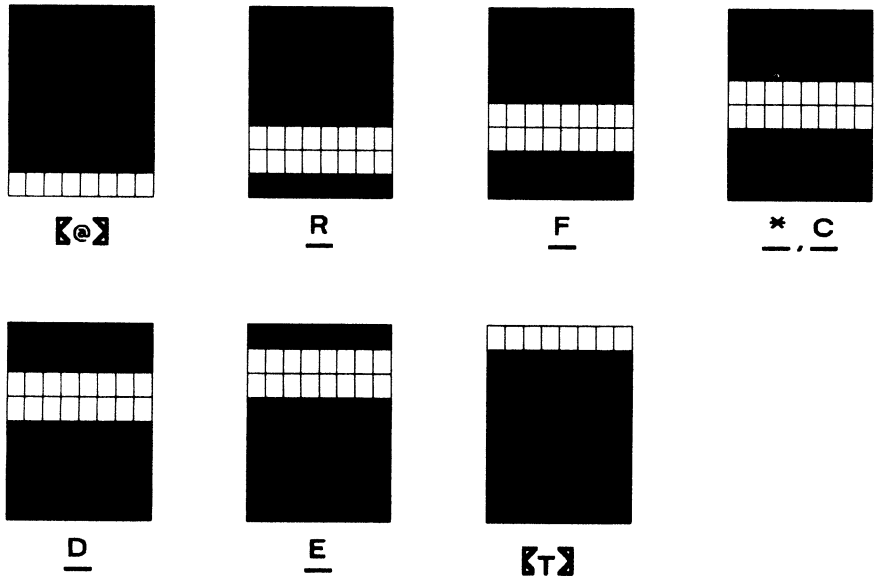


Figure 2–16 Graphics symbols that draw horizontal lines at one-eighth space resolution

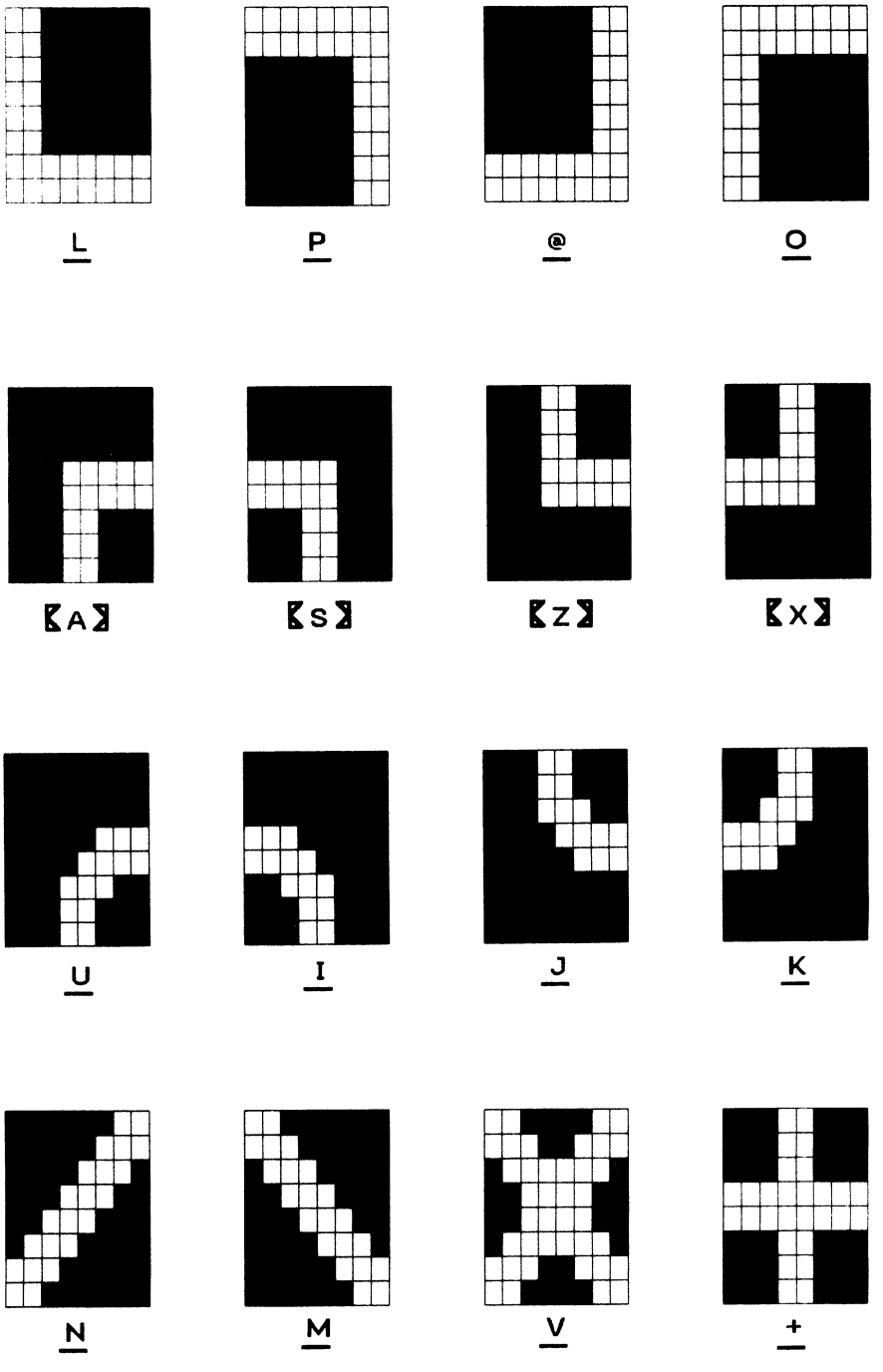


Figure 2–17 Graphics symbols that draw corners and diagonal lines

Figure 2–15 summarizes the vertical line characters. Commodore-G places the line at the left edge and Commodore-M at the right edge. Shift-T places the line two eighths of a space right of the left edge; Shift-G three eighths of a space right; Shift-B four eighths of a space right; Shift-minus five eighths of a space right (matching up with corners and T connectors); Shift-H six eighths of a space right; and Shift-Y seven eighths of a space right of the left edge. Note that each line is two eighths of a space wide.

Figure 2–16 summarizes the horizontal line characters. Commodore-@ places a horizontal line at the bottom and Commodore-T at the top. Shift-R places the line two eighths of a space up from the bottom; Shift-F three eighths of a space up; Shift-* four eighths of a space up (matching up with T connectors and corners); Shift-C five eighths of a space up; Shift-D six eighths of a space up; and Shift-E seven eighths of a space up.

Figure 2–17 summarizes the corners and diagonal lines. The diagonal lines are Shift-M (top left-hand corner to bottom right-hand corner), Shift-N (bottom left-hand corner to top right-hand corner), and Shift-V (both diagonals). The half-space corners are Commodore-X (upper left-hand corner), Commodore-Z (upper right-hand corner), Commodore-A (lower left-hand corner), and Commodore-S (lower right-hand corner). The full space corners are Shift-L (bottom left), Shift-@ (bottom right), Shift-O (top left), and Shift-P (top right). The rounded corners are Shift-I (lower left), Shift-U (lower right), Shift-K (upper left), and Shift-J (upper right).

Figure 2–18 summarizes the rest of the graphics symbols. The playing card symbols are: Shift-A (spade), Shift-X (club), Shift-S (heart), and Shift-Z (diamond). Shift-W produces an ellipse while Shift-Q prints a filled-in, elliptical disk. Shaded areas are produced by Commodore-plus (full space shaded), Commodore-minus (left half shaded), and Commodore-British pound sign (bottom shaded). T connectors are used with corner symbols to create ruler lines for business forms.

PRINTED FORMS

You can use the TAB function in PRINT statements to create a variety of forms containing columns of text, such as invoices or purchase orders. A calendar is a common example of such a form. Figure 2–19 shows a calendar for May 1993, drawn by our next program.

Program Name: "CALENDAR"

Purpose

Prints a calendar for any month of any year.

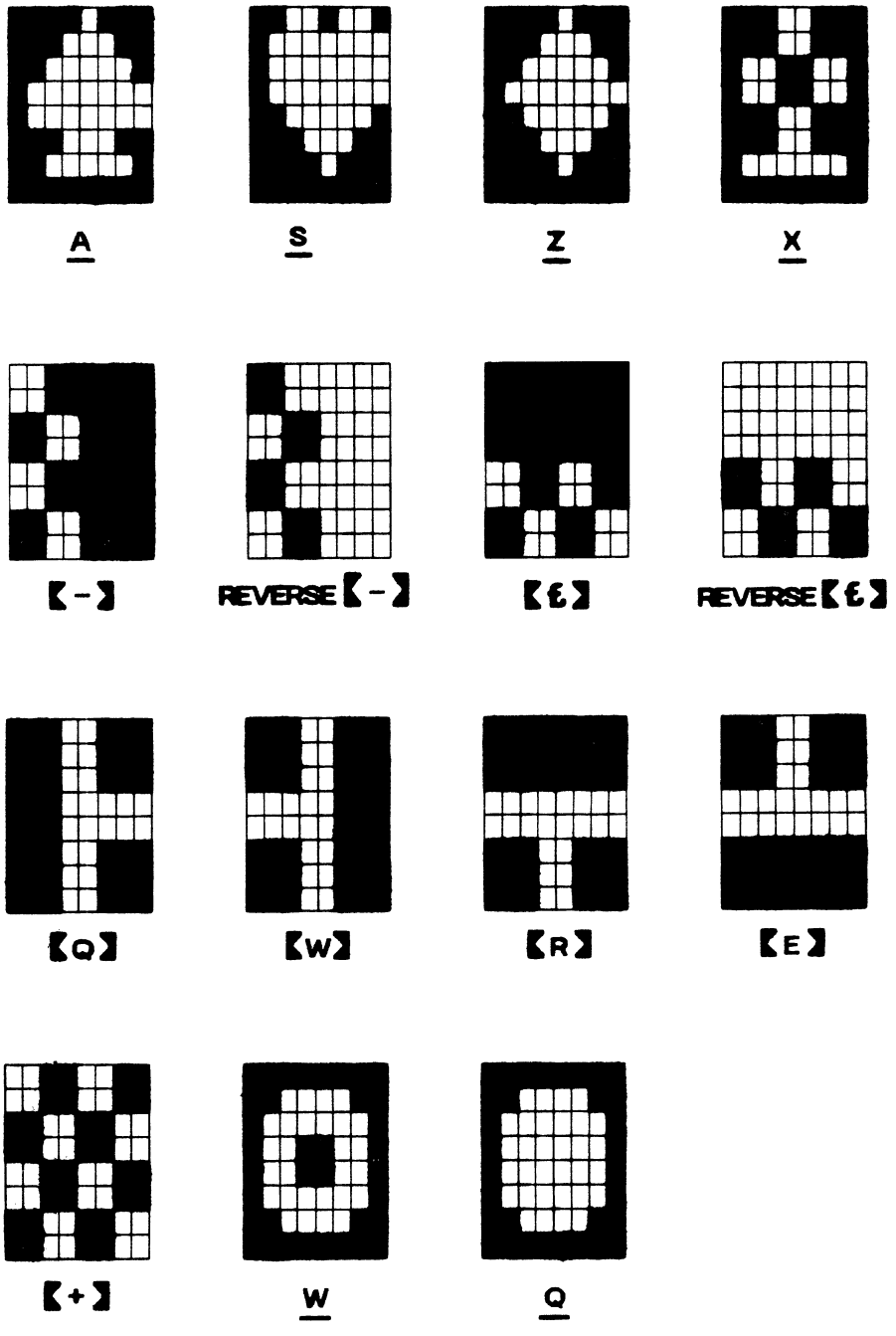


Figure 2–18 Graphics symbols that draw card symbols, shaded areas, and T connectors

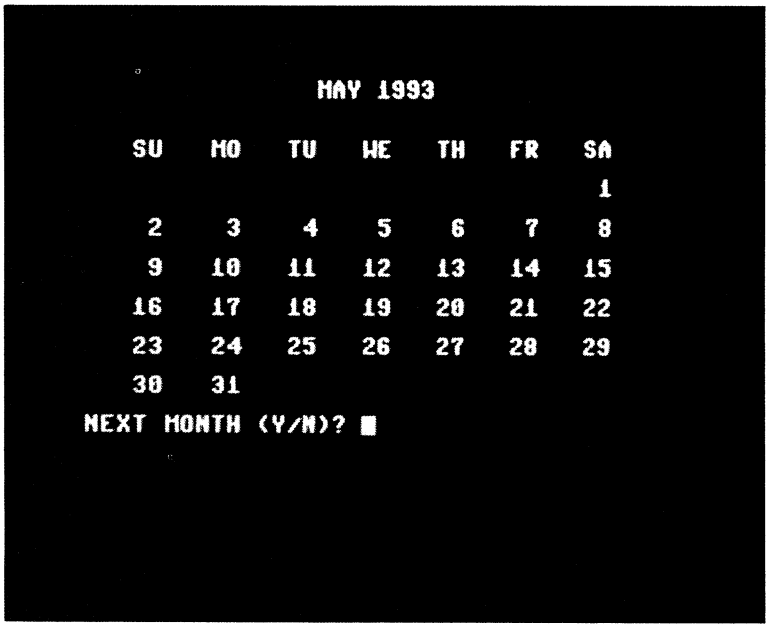


Figure 2–19 Calendar for May 1993

Techniques Demonstrated

Use of TAB to print items in columns. Also demonstrates the use of RIGHT$ and STR$ to align numbers. A modification illustrates the use of corners, T connections, and line characters to outline a form.

Procedure

The program asks for the year and month of the calendar. It clears the screen and displays the calendar for that month. The program then draws calendars on request for succeeding months.

Variables

A$: User response of Y or N
C: Column at which to start printing the date
D: Day of the week (1 to 7) of the first day of the month
DAY: Date
DAYS: Array containing the number of days in each month
M: Month (between 1 and 12)

MM: Month user asked to see
MNTH: Month index
NAMEM$: String array containing the names of all months
NDAYS: Number of days in month M
PY: Previous year, year minus one
Y: Year (must be greater than zero)

Special Cases

Obviously, February is special since it may have 28 or 29 days. To see the variations in the modern (Gregorian) calendar, run CALENDAR for February 1900, February 1987, February 1988, and February 2000. 1988 and 2000 are leap years, whereas 1900 and 1987 are not.

Brief Description

Lines 20 to 50 form lists of the names of the months and the usual numbers of days in them.

Lines 60 to 80 obtain the month and year to show. Line 80 rejects improper month entries.

Lines 90 to 100 compute the day of the week on which January 1st falls for the specified year. At this point, the day of the week is the remainder left when D is divided by 7; a remainder of 0 = Saturday, 1 = Sunday, . . ., 6 = Friday.

Lines 110 to 160 update the count D to the first day in the specified month by adding the number of days in each month preceding it.

Lines 170 to 180 calculate the day of the week on which the first day of the specified month falls. Line 180 adjusts the value so that Saturday is day 7 instead of day 0. This is necessary because calendars start their weeks with Sunday rather than Saturday at the far left.

Lines 190 to 210 print a heading for the calendar. The first line has the month and year. The next lines have the names of the days of the week above the columns in which the dates appear.

Line 230 calculates the column in which to start printing the first day of the month.

Line 240 determines how many days there are in the current month.

Lines 245 to 290 print the dates.

Line 260 prints a date starting in column C.

Line 270 increases C by 5 so the next date will start 5 columns farther right.

Line 280 checks whether the program has reached the end of a week. If it has, it moves the cursor down a line and back near the left edge, thus starting the next week's dates two lines down at the left.

Lines 300 to 320 ask the user whether he or she wants to see next month's calendar.

Lines 330 to 340 update the month number and starting day number to the next month.

Lines 350 to 380 update the year and month number to January of the next year if necessary.

Lines 1000 to 1030 determine the number of days in month M. Lines 1010 and 1020 make the number of days 29 if the month is February of a leap year.

Listing

```
5 REM "CALENDAR"
10 PRINT "{CLR}"
15 REM NAMES AND USUAL LENGTHS OF MONTHS
20 DATA JAN,31,FEB,28,MAR,31,APR,30,MAY,31,JUN,30
30 DATA JUL,31,AUG,31,SEP,30,OCT,31,NOV,30,DEC,31
40 DIM NAMEM$(12),DAYS(12)
50 FOR MNTH=1 TO 12: READ NAMEM$(MNTH), DAYS(MNTH): NEXT MNTH
55 REM ASK USER FOR YEAR AND MONTH
60 INPUT "WHICH^YEAR";Y
70 INPUT "MONTH^(1-12)";MM
80 IF MM<1 OR MM>12 THEN 70
85 REM CALCULATE DAY OF WEEK OF JANUARY 1
90 PY=Y-1
100 D=2+PY+INT(PY/4)-INT(PY/100)+INT(PY/400)
105 REM ADD DAYS BEFORE SPECIFIED MONTH
110 M=1
120 IF M=MM THEN 170
130 GOSUB 1000
140 D=D+NDAYS
150 M=M+1
160 GOTO 120
165 REM DETERMINE FIRST DAY OF FIRST OF MONTH
170 D=D-7*INT(D/7)
180 IF D=0 THEN D=7
185 REM PRINT CALENDAR HEADING
190 PRINT "{CLR}"
200 PRINT TAB(15) NAMEM$(M);Y
210 PRINT "{2 DOWN}^^^SU^^^MO^^^TU^^^WE^^^TH^^^FR^^^SA"
215 REM STARTING COLUMN OF DAY 1
220 PRINT
230 C=(D-1)*5+3
240 GOSUB 1000
245 REM PRINT DAYS OF MONTH
250 FOR DAY=1 TO NDAYS
260 PRINT TAB(C) RIGHT$(STR$(DAY),2);
270 C=C+5
275 REM START EACH WEEK ON A NEW LINE
280 IF C>=7*5 THEN C=3: PRINT "{DOWN}"
290 NEXT DAY
295 REM ASK USER WHETHER TO DO NEXT MONTH
```

```
300 PRINT: INPUT "{DOWN}NEXT^MONTH^(Y/N)"; A$
310 IF A$<>"Y" AND A$<>"N" THEN
        PRINT "PLEASE ANSWER Y OR N": GOTO 300
320 IF A$="N" THEN END
325 REM NEXT MONTH
330 D=D+NDAYS
340 M=M+1
350 IF M<=12 THEN 170
355 REM NEXT YEAR
360 Y=Y+1
370 M=1
380 GOTO 170

995 REM DETERMINE NUMBER OF DAYS IN MONTH M
1000 NDAYS=DAYS(M)
1005 REM FEBRUARY (MONTH 2) HAS 29 DAYS IN LEAP YEARS
1010 IF M<>2 THEN RETURN
1020 IF (Y/4=INT(Y/4) AND Y/100<>INT(Y/100))
        OR Y/400=INT(Y/400) THEN NDAYS=29
1030 RETURN
```

Modifications

Try marking weekends on the calendar by coloring them differently than the rest of the days. You can do this by inserting the lines:

```
252 PRINT "{7}";
254 IF C=3 OR C=33 THEN PRINT "{WHT}";
292 PRINT "{7}";
```

You can mark holidays similarly. Simply add lines that change the printing color on the holiday dates. For example, the following line makes Christmas Day appear in yellow:

```
256 IF M=12 AND DAY=25 THEN PRINT "{YEL}";
```

By using vertical and horizontal center lines, T-connections, and half-space corners, we can put ruler lines on the calendar. Enter the following:

```
293 REM ADD RULER LINES
294 GOSUB 2000

1995 REM ADD OUTLINES TO CALENDAR
2000 PRINT "{HOME}^{A}{34 *}{S}"   (34 shifted asterisks)
2010 PRINT "^-" SPC(34) "-"
2020 PRINT "^{Q}{6 ****{R}}****{W}"
                    (6 units of 4 shifted asterisks and Commodore-R)
2030 PRINT "^-{4 RIGHT}-{4 RIGHT}-{4 RIGHT}-{4 RIGHT}-
        {4 RIGHT}-{4 RIGHT}-{4 RIGHT}-"
2040 PRINT "^-{4 RIGHT}-{4 RIGHT}-{4 RIGHT}-{4 RIGHT}-
        {4 RIGHT}-{4 RIGHT}-{4 RIGHT}-"
2050 PRINT "^{Q}{6 ****+}****{W}"
                    (6 units of 4 shifted asterisks and shifted +)
2060 FOR J=1 TO 12
2070 PRINT "^-{4 RIGHT}-{4 RIGHT}-{4 RIGHT}-{4 RIGHT}-
        {4 RIGHT}-{4 RIGHT}-{4 RIGHT}-"
```

```
2080 NEXT J
2090 PRINT "^{Z}{6 ****{E}}****{X}"
                    (6 units of 4 shifted asterisks and Commodore-E)
2100 RETURN
```

Note that lines 2030, 2040, and 2070 are identical. The calendar looks more professional now. We can also round the corners by changing lines 2000 and 2090 to:

```
2000 PRINT "{HOME}^U{34 *}I"        (34 shifted *)
2090 PRINT "^J{6 ****{E}}****K"
                    (6 units of 4 shifted asterisks and Commodore-E)
```

To make the ruler lines black, insert

```
293 PRINT "{BLK}";
295 PRINT "{7}";
```

Notes

This program shows how to determine whether one number is divisible by another. M is divisible by N only if M/N is an integer (whole number); that is, if M/N = INT (M/N). This test occurs several times in the subroutine that identifies leap years. Line 1020 shows that a year is a leap year in the modern (Gregorian) calendar if either of the following is true:

1. The year is divisible by 4, but not by 100.
2. The year is divisible by 400.

Thus, 2000 is a leap year, whereas 1900 and 2100 are not.

Line 260 allows the computer to print seven neatly spaced dates on a line. When PRINTing a number, the computer puts one space on each side automatically to separate it from other numbers on the same line. Thus a PRINTed two-digit number occupies four columns.

To see how the computer prints numbers, RUN the line:

```
PRINT 7;11;17;23 (the semicolons ensure no extra spaces between numbers)
```

The result looks like:

```
 7  11  17  23
READY.
```

since the computer puts a space before and after each number.

We can reduce the number of columns each date occupies by using the STR$ function. STR$(N) produces a string version of the number N pre-

ceded by a single space. Thus, printing STR$(N) rather than N itself makes two-digit dates occupy three columns rather than four.

The problem here is that STR$(N) can consist of either two or three characters, depending on whether N has one or two digits. This results in misalignment, since the lengths vary and printing STR$(N) always produces a space in front. To see what happens, try RUNning

```
PRINT STR$(7);STR$(8);STR$(9)
```

and

```
PRINT STR$(21);STR$(22);STR$(23)
```

The 7 ends up above the 2 in 21, instead of above the 1 as it should be in a calendar. Furthermore, the alignment becomes worse to the right, since each one-digit date occupies two columns, whereas each two-digit date occupies three columns.

We can correct the misalignment by using RIGHT$(STR$(N),2) to select the right-most two characters of STR$(N). These characters are a space and the date if the date is a single digit, and just the date if it is two digits. That is, we have gotten rid of the extra space ahead of the two-digit dates. To see this, RUN the following lines:

```
PRINT RIGHT$(STR$(7),2);RIGHT$(STR$(8),2);RIGHT$(STR$(9),2)
PRINT RIGHT$(STR$(21),2);RIGHT$(STR$(22),2);RIGHT$(STR$(23),2)
```

Now the numbers line up the way they should.

References

The book *TIME* (Life Science Library, Time Incorporated, 1966) contains an article entitled "Subdividing the Year," which discusses the development of the modern calendar.

SIMPLE ANIMATION

Program TREE uses simple arithmetic to draw a geometrical figure, a triangular-shaped Christmas tree. It then changes parts of the picture to produce animation. The succession of slightly different pictures makes the tree appear to have twinkling lights hung on it.

Program Name: "TREE"

Purpose

Draws a Christmas tree with twinkling colored lights (see Figure 2–20 and Plate 4).

Techniques Demonstrated

Illustrates simple animation based on a succession of slightly different pictures. Demonstrates the use of random numbers to create irregular sequences.

Procedure

RUN the program. It asks you to enter the number of lights. The program then draws a tree, selects light positions randomly, and changes the colors of the lights.

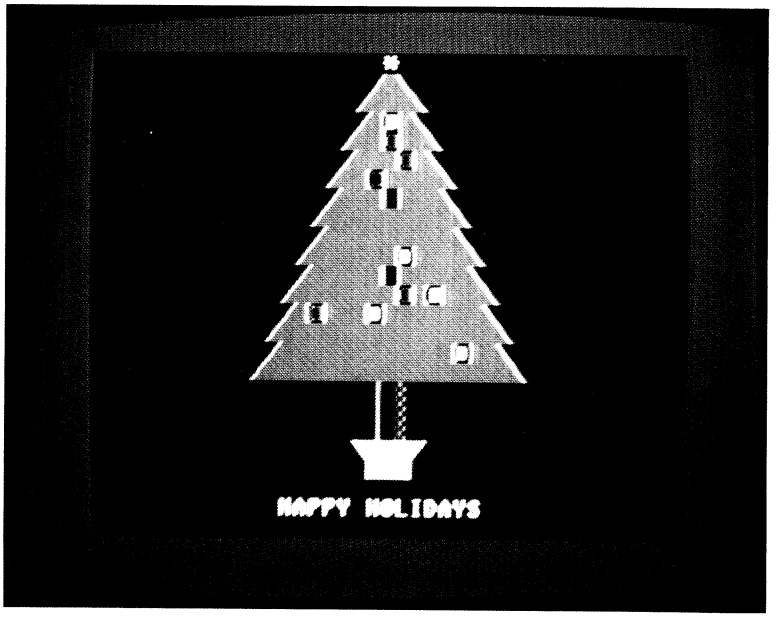


Figure 2–20 Christmas tree drawn by Program TREE

Variables

C: Horizontal (column) position of a light
C$: String containing color commands
CENTER: Number of the middle column on the screen (19)
COL: Horizontal positions of lights
J: Number of light being placed on tree
LEFTEDGE: Starting column for each row of the tree
LIGHT: Number of a randomly chosen light
LT$: String containing a colored light
MAXWIDTH$: String representing the maximum width of the tree; it contains 17 space characters
N: Number of lights
R: Vertical (row) position of a light
ROW: Vertical positions of lights
VERT$: Vertical positioning string
WIDTH: Width of the bulk of the tree on each row (not including the triangular edges) in character spaces

Brief Description

Line 10 sets up a string of vertical positioning commands for use in placing lights randomly.
Line 13 sets up a string of spaces for use in creating the solid bulk of the tree.
Line 20 asks the user to enter a value for N, the number of lights. Numbers 5 to 15 are reasonable values.
Line 40 sets up a string of color commands for use in flashing lights in random colors.
Line 50 sets the tree's central column.
Line 80 puts a white star at the top of the tree.
Lines 90 to 130 draw the body of the tree, using triangles for the edges and a string of inverted spaces for the bulk.
Lines 140 to 180 draw the tree's base and trunk.
Lines 190 to 230 hang lights at random positions on the tree.
Line 240 prints HAPPY HOLIDAYS below the tree.
Lines 250 and 270 make the star at the top of the tree flash by alternating it with a space character.

Lines 260 and 280 each display 16 randomly chosen lights in random colors.

Line 510 randomly chooses one of the N lights to display.

Line 520 obtains the randomly chosen light's row and column positions.

Line 530 chooses a color randomly for the light.

Line 540 displays the randomly chosen light in its random color.

Listing

```
5 REM "TREE" AN ANIMATED CHRISTMAS TREE
10 VERT$="{HOME}{24 DOWN}"
13 MAXWIDTH$="{17 ^}"
15 REM ASK FOR NUMBER OF LIGHTS TO BE HUNG
20 INPUT "{CLR}{DOWN}NUMBER^OF^LIGHTS"; N
30 DIM COL(N), ROW(N)
35 REM COLORS OF DECORATIONS
40 C$="{WHT}{RED}{CYN}{PUR}{BLU}{YEL}{1}{2}{3}{6}{7}"
45 REM SET CENTRAL COLUMN
50 CENTER=19
55 REM SET COLUMN OF LEFT EDGE AT TOP OF TREE
60 LEFTEDGE=CENTER-1
65 REM RANDOMIZE
70 J=RND(0)
75 REM PRINT TREE
80 PRINT "{CLR}" TAB(CENTER) "{WHT}*{GRN}"
90 FOR WIDTH=1 TO 15 STEP 2
100 PRINT TAB(LEFTEDGE) "{RVS}£" LEFT$(MAXWIDTH$,WIDTH) "{*}"
110 LEFTEDGE=LEFTEDGE-1
120 PRINT TAB(LEFTEDGE) "{RVS}£" LEFT$(MAXWIDTH$,WIDTH+2) "{*}"
130 NEXT WIDTH
140 PRINT TAB(CENTER-1) "{2}{N}{RVS}^{OFF}{-}"
150 PRINT TAB(CENTER-1) "{2}{N}{RVS}^{OFF}{-}"
160 PRINT TAB(CENTER-1) "{2}{N}{RVS}^{OFF}{-}"
170 PRINT TAB(CENTER-2) "{YEL}{*}{RVS}^^^{OFF}£"
180 PRINT TAB(CENTER-1) "{RVS}^^^"
185 REM CHOOSE POSITIONS OF LIGHTS RANDOMLY
190 FOR J=1 TO N
200 R=INT(14*RND(1))-1
210 ROW(J)=21-R
220 COL(J)=CENTER-INT(R/2)+INT(RND(1)*(2*INT(R/2)+1))
230 NEXT J
235 REM ANIMATE SCENE WITH FLASHING STAR AND LIGHTS
240 PRINT TAB(CENTER-7) "{DOWN}{WHT}HAPPY^HOLIDAYS"
250 PRINT "{HOME}"; TAB(CENTER) "^";
260 GOSUB 500
270 PRINT "{HOME}"; TAB(CENTER) "{WHT}*";
280 GOSUB 500
290 GOTO 250
495 REM DISPLAY 16 LIGHTS IN RANDOM COLORS
500 FOR J=1 TO 16
510 LIGHT=INT(RND(1)*N)+1
520 C=COL(LIGHT): R=ROW(LIGHT)
530 LT$=MID$(C$,INT(RND(1)*LEN(C$))+1,1)+"Q"
540 PRINT LEFT$(VERT$,25-R);SPC(C) LT$;
550 NEXT J
560 RETURN
```

Modifications

Trim the tree with snow by inserting these lines:

```
100 PRINT TAB(LEFTEDGE) "{WHT}{RVS}£{GRN}"
          LEFT$(MAXWIDTH$,WIDTH) "£*£"
120 PRINT TAB(LEFTEDGE) "{WHT}{RVS}£{GRN}"
          LEFT$(MAXWIDTH$,WIDTH+2) "£*£"
```

You can obtain line 120 easily from line 100 (insert "+2" after WIDTH). To trim both edges with snow, replace £*£ with {WHT}{RVS} £*£ {GRN} at the ends of lines 100 and 120.

Notes

To draw a triangular tree in the center of the screen, we first sketch it on the character position chart (Figure 1-2). We want to leave room for a star at the top, and a trunk, a base, and a greeting at the bottom. Thus the triangular shape extends from line 23 (at the top) through line 8 (at the bottom). On line 23 the bulk of the tree is 1 column wide; this does not include the edges. On lines 21 and 22 the bulk is 3 columns wide. On lines 7 and 8 the bulk is 15 columns wide. On line 9 the bulk is 17 columns wide. We increase the number of columns by two for every two lines. The tree must be symmetric around the trunk—that is, extend equally far on both sides of its center.

Note that the bulk of the tree's branches extends one column farther from the center horizontally every second row (line) down. Thus the TAB value for the left edge must decrease by one for every two lines we move down. This explains program lines 90 through 130.

The computer can choose a random number between LOW and HIGH with INT(RND(1) * (HIGH − LOW + 1)) + LOW. RND picks a random number between 0 and 1 that is never exactly 1. The multiplication extends the range to between 0 and HIGH − LOW + 1. Then INT drops the fractional part of the number, making it into a whole number (integer) between 0 and HIGH − LOW. Line 510 uses this approach to choose a light randomly. Here, LOW is 1 and HIGH is N.

If you choose LOW to be zero and HIGH one, this formula simulates the tossing of a coin. In fact, if you have trouble making decisions, try this program:

```
10 R=RND(0): REM RANDOMIZE
20 R=INT(RND(1)*2)
30 IF R=0 THEN PRINT "YES"
40 IF R=1 THEN PRINT "NO"
```

This short program and TREE both use RND(0) to start the random number sequence in different places each time. While RND does produce random numbers, its sequences will always be the same if it starts at the same place. You could compare this to showing a film of a person throwing dice. Even though the dice might be honest and the tosses random, the sequence of values would always be the same if we started the film at the same point. RND (0) starts the sequence at a value determined by the Commodore 64's clock. This is like choosing a starting point on the film by running it forward until your watch reaches the next minute.

In lines 100 and 120, LEFT$ chooses a number of spaces from MAXWIDTH$ equal to the width of the bulk of the tree on that line. The inverted space characters then form the bulk, while the triangular graphics symbols form the edges.

Line 220 chooses a random column for a light. It must, however, choose that column from among those occupied by the bulk of the tree on the randomly chosen row.

Line 530 picks a color randomly for a light. MID$(C$,CFIRST,NUM) selects NUM characters from C$, starting with the CFIRST character. Thus here MID$ picks 1 color command from a random position in the string, since INT(RND(1)*LEN(C$))+1 has a random value between 1 and the length of the string. When applied to strings, + means "put together" or "concatenate," not "add." LT$ thus ends up being a two-character string consisting of a color change command followed by Q (shifted Q).

The random numbers move the flashing lights around the tree, thus producing a scene that is constantly changing. This is essential to hold the viewer's attention; people will not keep looking at a scene if nothing is happening or if the pattern is simple and repetitive. Video games, in particular, must produce a variety of constantly changing, interesting scenes.

If, when you press STOP to end this program, you don't see a cursor on the screen, press {WHT} (CTRL and 1). The cursor is there, but you cannot see it because it is blue on a blue background.

When you enter the {HOME}s on lines 250 and 270, don't press SHIFT. SHIFTed {HOME} is {CLR}, which will make the tree vanish, and leave the lights flashing in thin air!

3

DRAWING LINES AND CIRCLES

By specifying which character goes in each position, as we did in the programs in Chapter 2, you can draw any picture you want. However, this is a very time-consuming process. You must sketch the picture, select all the characters, write the program, and then enter each line into the computer. Finding and correcting errors (incorrect characters, typing mistakes, or programming errors) makes the entire process take longer than you ever expected.

Fortunately, the computer can do more of the work in most situations. After all, no matter how abstract they are, most pictures do not consist of random characters in random positions. Instead, they contain lines, circles, and other familiar geometrical forms. Since these forms are governed by the rules of geometry and trigonometry, the computer can generate them from a small amount of information. You can then use them as building blocks, rather than using single characters.

This chapter develops programs that produce geometrical forms. It starts with two useful routines, PRINTCHR and LIMIT, that place characters at specified coordinates and demonstrate alternative ways of keeping figures on the screen, respectively. These then serve as a basis for programs RECTANGLE, LINES, TURTLE, and CIRCLE, which actually generate geometrical forms.

PLOTTING POINTS IN CARTESIAN COORDINATES

We generally describe geometrical forms in terms of a Cartesian coordinate system. In this system, the horizontal direction is X and the vertical direction Y. We can consider the Commodore 64's screen as a Cartesian coordinate system; it is equivalent to the upper-right quarter (or *quadrant*) of a typical piece of graph paper with the origin at its center. To remain consistent with our earlier description of the screen as a grid, we will make the X coordinate be the column number (0 to 39) and the Y coordinate the line number (0 to 24). Thus our Cartesian system (see Figure 3–1) has:

1. Origin (X = 0, Y = 0) in the lower left-hand corner.
2. X axis (Y = 0) in the bottom row.
3. Y axis (X = 0) along the left edge.

We can then describe any point on the screen by means of its X - and Y - coordinates. For example:

1. The top left-hand corner has coordinates X = 0 and Y = 24.
2. The point just left of the center of the screen has coordinates X = 19 and Y = 12.
3. The bottom right-hand corner has coordinates X = 39 and Y = 0.

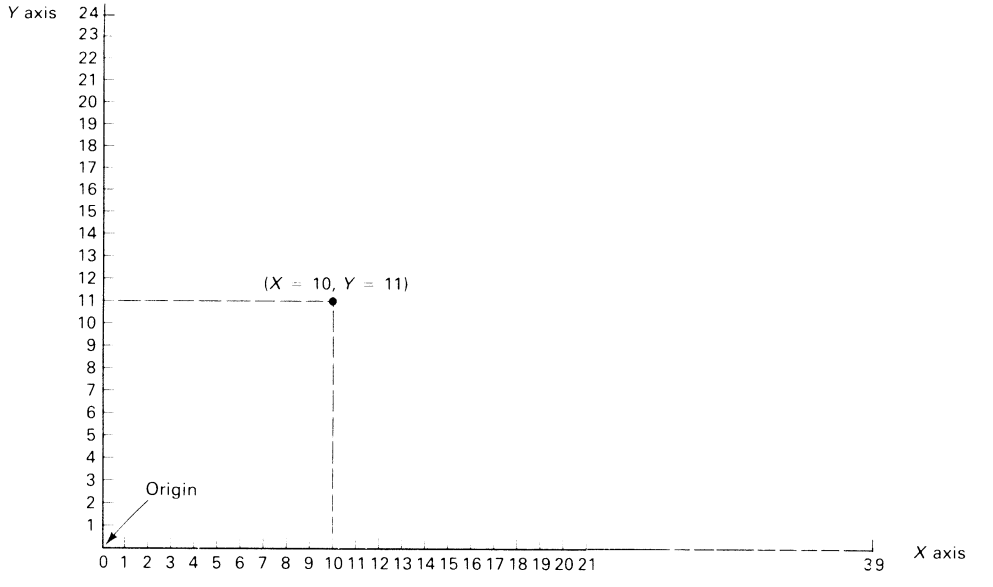


Figure 3–1 Cartesian coordinate system for screen

The following sequence in a PRINT statement moves the cursor to an arbitrary position (X,Y):

1. {HOME} moves it to the upper left-hand corner.
2. 24-Y {DOWN}s move it down to line (row) Y.
3. SPC(X) or TAB(X) moves it right to column X.

Program PRINTCHR contains a subroutine that implements this sequence. We will use the routine as a standard approach throughout the rest of this book.

PLACING CHARACTERS AT SPECIFIC COORDINATES

Program Name: PRINTCHR

Purpose

Places characters on the screen, starting in a particular X (horizontal) and Y (vertical) position. Figure 3–2 shows the output from a typical run.

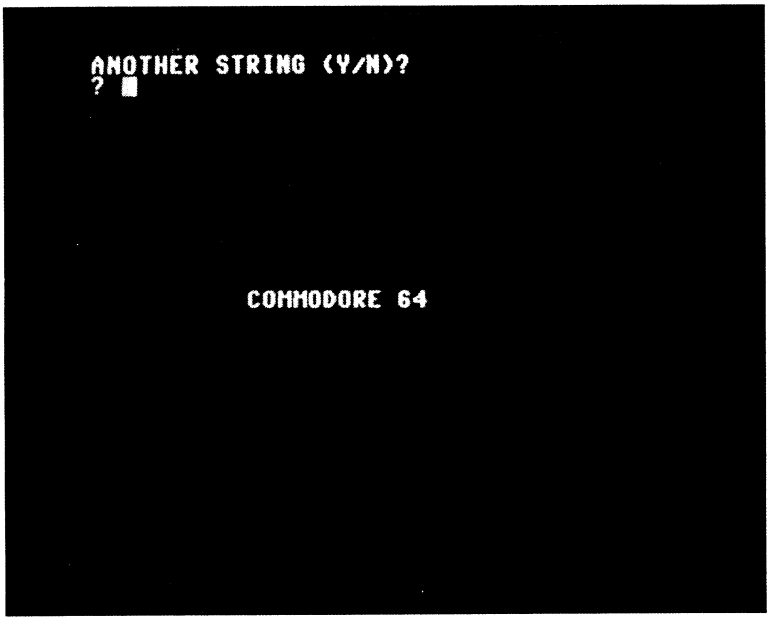


Figure 3–2 Output of PRINTCHR

Techniques Demonstrated

Using {DOWN} and SPC to move the cursor to a point with coordinates X and Y.

Procedure

The program asks for the line number, column number, and characters to print. It then prints the characters starting at the specified location.

Variables

A$: Answer (Yes or No) to question of whether to continue

CHARS$: Character(s) to print

VERT$: String containing {HOME} and 24 {DOWN} commands, used to position the cursor vertically

X: Column number

Y: Line number

Special Cases

The program does not check whether the coordinates are on the screen. If the line number exceeds 25 or the column number exceeds 255, the program will terminate with an ILLEGAL QUANTITY ERROR IN LINE 1000. Other values, while not illegal, will produce results that may not be what you expected:

1. A column number between 40 and 255 makes the computer move down a line after each 40 columns. For example, column 40 is equivalent to column 0 of the next line, column 60 is near the center of that line, and column 80 is equivalent to column 0 two lines down. This is like saying that three days after March 30th is April 2nd, not March 33rd. Moving down entire lines may make the computer move (scroll) up the entire display if it reaches the end of line 0.

2. Characters printed on lines 23 and 24 will interfere with PRINTCHR's entry prompts. Program line 100, in fact, will immediately print over part of line 24.

3. Line 25 is the same as line 24, since LEFT$(VERT$,0) takes nothing from the VERT$ string. However, program line 80 starts the cursor in the top, left-hand corner anyway.

Brief Description

Line 10 sets up the VERT$ string for use in the positioning subroutine.

Lines 20 to 80 ask the user for the line number, column number, and characters. Each entry must end with a RETURN.

Line 90 calls the subroutine that prints the characters.

Lines 100 to 130 ask the user whether to continue. These lines repeat the program if the answer is Y(es), stop it if the answer is N(o), and ask again if the answer is neither Y nor N.

Line 10000 first uses Y (line number) to obtain the correct number of {DOWN} commands from VERT$. LEFT$(VERT$,25-Y) selects the left-most 25-Y characters of VERT$, thus ending up with {HOME} and 24-Y {DOWN}s. SPC(X) then moves the cursor right to column X. Thus the computer starts printing CHARS$ in column X of line Y.

Listing

```
5 REM "PRINTCHR"
10 VERT$="{HOME}{24 DOWN}"
15 REM REQUEST POSITION, CHARACTERS
20 PRINT "{CLR}WHAT^LINE?^(0-24)"
30 INPUT Y
40 PRINT "{CLR}WHAT^COLUMN?^(0-39)"
50 INPUT X
60 PRINT "{CLR}WHAT^CHARACTERS?"
70 INPUT CHARS$
80 PRINT "{CLR}";
85 REM PRINT LINE
90 GOSUB 10000
95 REM DECIDE WHETHER TO CONTINUE
100 PRINT "{HOME}ANOTHER^STRING^(Y/N)?"
110 INPUT A$
120 IF A$="Y" THEN 20
130 IF A$<>"N" THEN 100
140 END

9995 REM PRINT A STRING AT X,Y
10000 PRINT LEFT$(VERT$,25-Y); SPC(X) CHARS$;
10010 RETURN
```

Modifications

See what happens if the character string is too long to fit on a line. Try starting "FIRST QUARTER SALES" in column 30 of line 15. What happens if you start it in column 30 of line 0?

If you want to use color commands, {RVS},{OFF}, cursor controls, or commands such as {CLR} in the string, you must put quotation marks around them. For example, you can enter a string consisting of three reversed spaces (solid squares) by entering "{RVS}^^^{OFF}". This will form a light blue line three columns long, starting at the specified location. To make the line red but restore the normal light blue printing color afterward, type "{RED}{RVS}^^^{OFF}{7}".

Note the difference between {RVS} and a color change. A color change stays in effect until the next color change or until RUN/STOP-RESTORE returns the computer to its normal dark blue screen with light blue characters. {RVS}, on the other hand, stays in effect only until the next RETURN or {OFF}.

KEEPING PICTURES ON THE SCREEN

Keeping pictures on the screen is a continual problem in computer graphics. The computer cannot do much to help without making a nuisance of itself. If, for example, it stops and displays an error message (as the Commodore 64 does when a program TABs to a negative column), that message will distort or destroy the picture. Furthermore, users may have trouble determining the causes of generalized messages such as ILLEGAL QUANTITY IN LINE 100.

When the computer takes automatic action, the results are often worse. If, for example, a program tells the computer to print to the right of column 39, it will proceed to column 0 of the next line. Similarly, if a program tells it to print below line 0, it will simply move the entire screen display up a line to make room. Both actions distort pictures and produce strange-looking results.

How can a program help? There are three reasonable options:

1. Stop the drawing and report a problem whenever a coordinate is off the screen. This destroys the current picture, but can tell the user exactly what error occurred. It is a natural approach when the edges of the screen represent actual physical limits, as in creating computer message boards or in designing posters, signs, cards, artwork, or visual aids.

2. Stop at the edge. We call this approach *clipping*, since it cuts figures off as if you had clipped them with a pair of scissors. It is natural when the edges are just the limits of the viewing area, as in drawing maps, landscapes, or playing fields.

3. Continue on the other side of the screen. We call this approach *wraparound*, since it works as if the two edges of the screen were connected around the back. It is natural in drawing projections of spheres (e.g., the Earth) and cylinders (e.g., a pipeline or rocket) where the edges really are connected. It also adds an extra dimension to video games; for example, it makes a spacecraft or monster reenter from the opposite edge after it disappears off the screen. This obviously makes maneuvering and avoiding danger much more difficult.

Clipping and wraparound both produce a completed picture, although it may not look the way the user intended.

Another way to keep pictures on the screen is by checking the input data. The following sequence, for example, rejects vertical coordinate (Y) values that are not between 0 and 38.

```
5 REM INPUT ERROR TEST
10 INPUT "LINE NUMBER (0-38)"; Y
20 IF Y<0 OR Y>38 THEN PRINT "INCORRECT VALUE": GOTO 10
```

This keeps the user from accidentally entering an improper value, but does not keep the program from generating them. For example, a line-

drawing program might start at a valid point but continue beyond the edge of the screen.

Program LIMIT includes three limit-checking subroutines as well as input error checking (or *data validation*).

Program Name: "LIMIT"

Purpose

Draws a horizontal line starting at a specified point and continuing for a specified distance in the specified direction. Illustrates three ways of avoiding off-screen errors. Figure 3–3 shows the output of a typical run.

Techniques Demonstrated

Drawing of horizontal lines, validation of coordinates, reporting of off-screen errors, clipping, and wraparound.

Procedure

The program asks for the horizontal and vertical starting positions, the line length (in character positions), and the horizontal direction. It then plots

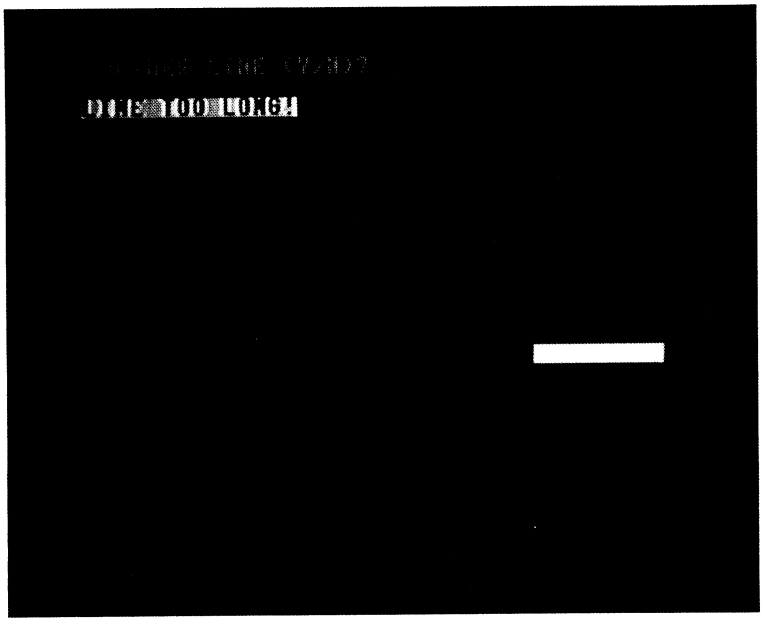


Figure 3–3 Output of Program LIMIT (X = 30, Y = 9, L = 15, D$ = "R")

the line. If the line goes off the screen, the limit-checking subroutine in line 5000 stops and reports the problem, the one in 6000 clips the line, while the one in 7000 continues it on the other side of the screen.

Variables

A$: Answer Y (yes) or N (no) to question of whether to repeat the program

D: Direction value (1 for right, −1 for left)

D$: Line direction, R (right) or L (left)

H: Length counter

L: Line length

OK: Indicator from limit-checking subroutine of whether coordinates are valid (0=invalid, 1=valid)

PLOT$: Character to be PRINTed

VERT$: Vertical positioning string with [HOME] and 24 [DOWN] commands

X: Horizontal plotting position

Y: Vertical plotting position

Brief Description

Line 10 sets up VERT$ as a string for determining vertical position.

Line 20 defines the plotting character as a solid square.

Line 30 clears the screen.

Lines 40 to 60 make the printing color cyan and ask for the horizontal (X) position, rejecting off-screen values.

Lines 70 to 90 ask for the vertical (Y) position, rejecting off-screen values.

Line 100 asks for line length.

Lines 120 to 160 ask for the direction, and set the step value from it. The step is 1 for a line heading right, −1 for one heading left.

Line 170 calls the line-drawing subroutine.

Lines 180 to 210 repeat the program at the user's request.

Lines 400 to 410 clear the prompt line.

Lines 4000 to 4050 are the line-drawing subroutine.

Line 4010 calls the limit-checking subroutine.

Line 4020 extends the line by one column if deemed proper by the limit-checking subroutine.

Line 4030 adds the step value to the horizontal coordinate X.

Lines 5000 to 5030 stop the program if the X value is off the screen and print an error message.

Lines 6000 to 6020 keep the plot on the screen by restricting the X value to 0 through 38.

Lines 7000 to 7020 provide wraparound to the opposite side of the screen.

Lines 10000 to 10010 PRINT a solid square at the specified coordinates (X,Y).

Listing

```
5 REM "LIMIT"
10 VERT$="{HOME}{24 DOWN}"
20 PLOT$="{RVS}^{OFF}": REM USE A REVERSED SPACE TO FORM LINE
30 PRINT "{CLR}"
40 INPUT "{CYN}{HOME}X^POSITION^0-38";X
50 GOSUB 400: REM CLEAR PROMPT LINE
60 IF X<0 OR X>38 THEN 40
70 INPUT "{HOME}Y^POSITION^0-22";Y
80 GOSUB 400
90 IF Y<0 OR Y>22 THEN 70
100 INPUT "{HOME}LINE^LENGTH";L
110 GOSUB 400
120 INPUT "{HOME}DIRECTION^(R/L)";D$
130 GOSUB 400
140 IF D$<>"R" AND D$<>"L" THEN 110
150 D=1
160 IF D$="L" THEN D=-1
170 GOSUB 4000: REM DRAW LINE
180 INPUT "{HOME}ANOTHER^LINE^(Y/N)";A$
190 GOSUB 400
200 IF A$="Y" THEN 30
210 IF A$<>"N" THEN 180
220 PRINT "{CLR}{7}"
230 END

395 REM CLEAR PROMPT LINE
400 PRINT "{HOME}{39 ^}"
410 RETURN

3995 DRAW LINE
4000 FOR H=1 TO L
4010 GOSUB 5000: REM CHECK FOR OFF-SCREEN VALUE
4020 IF OK=1 THEN GOSUB 10000
4030 X=X+D
4040 NEXT H
4050 RETURN

4995 REM STOP DRAWING AND REPORT IF OFF-SCREEN
5000 IF X>=0 AND X<=38 THEN OK=1: GOTO 5040
5005 REM PRINT WARNING MESSAGE
5010 PRINT "{HOME}{2 DOWN}{RVS}{6}LINE^TOO^LONG!{CYN}"
5020 H=L
5030 OK=0
5040 RETURN

5995 REM RESTRICT DRAWING TO ON-SCREEN VALUES (CLIPPING)
6000 OK=1
```

```
6010 IF X<0 OR X>38 THEN OK=0
6020 RETURN

6995 REM PROVIDE WRAPAROUND TO OTHER SIDE OF SCREEN
7000 IF X>38 THEN X=X-39
7010 IF X<0 THEN X=X+39
7020 OK=1
7030 RETURN

9995 REM PLOT SOLID SQUARE AT X,Y
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN
```

Modifications

By changing the GOSUB line number to 5000, 6000, or 7000 in line 4010, you can experiment with the three different ways of handling off-screen conditions.

You can change the line's color or appearance by changing PLOT$ in line 20. For example:

```
20 PLOT$="{RVS}{RED}^{OFF}{CYN}"
```

produces a solid red line,

```
20 PLOT$="{+}"
```

results in what looks like a mesh fence,

```
20 PLOT$="{RVS}{*}{OFF}"
```

creates a serrated edge,

```
20 PLOT$="Q"
```

produces a row of solid circles, and

```
20 PLOT$="{@}"
```

draws a thin line along the bottom of the character spaces.

DRAWING RECTANGLES

Suppose you want to draw a city skyline. The drawing should show the boxy profile of tall buildings typical of 1950s and '60s architecture. You could specify each character in each building's profile. A simpler approach, how-

ever, would be to determine each building's upper left-hand and lower right-hand coordinates, then let RECTANGLE generate connecting lines.

Program Name: "RECTANGLE"

Purpose

Draws a rectangle. Figure 3–4 shows a typical output.

Techniques Demonstrated

Drawing horizontal and vertical lines. Generation of simple geometric figures.

Procedure

Enter the corner coordinates and watch the rectangle appear.

Variables

A$: Answer Y (yes) or N (no) to question of whether to repeat the program

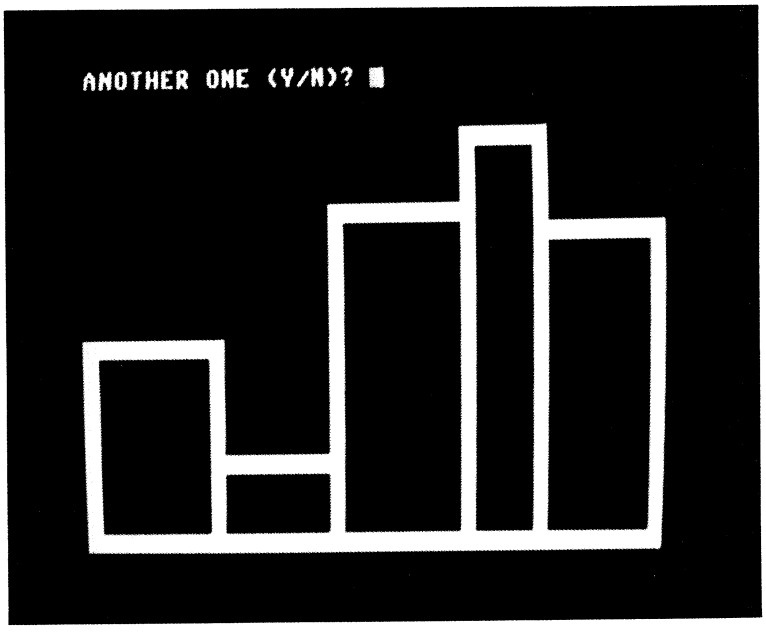


Figure 3–4 Skyline drawn by Program RECTANGLE

PLOT$: Solid square

VERT$: Vertical positioning string with {HOME} and 24 {DOWN} commands

X, Y: Coordinates for plotting solid square

X1, Y1: Coordinates of upper left-hand corner

X2, Y2: Coordinates of lower right-hand corner

Brief Description

Lines 10 to 30 clear the screen and initialize VERT$ and PLOT$.

Lines 40 to 90 ask for the coordinates of the upper left-hand corner, rejecting values that are off the screen.

Line 100 plots the specified upper left-hand corner. This lets you see it while you are entering the other corner's coordinates.

Lines 110 to 160 ask for the coordinates of the lower right-hand corner. These lines reject values that are off the screen or down from and right of the upper left-hand corner.

Line 170 plots the specified lower right-hand corner.

Line 180 calls the rectangle-drawing subroutine.

Lines 190 to 230 repeat the drawing process on request.

Lines 400 to 410 clear the prompt line.

Lines 4000 to 4030 calculate the coordinate values to draw the rectangle's top and bottom.

Lines 4040 to 4080 calculate the coordinate values to draw the rectangle's sides.

Lines 10000 to 10010 move the cursor to the X,Y position and PRINT the characters in PLOT$.

Listing

```
5 REM "RECTANGLE"
10 PRINT "{CLR}"
20 VERT$="{HOME}{24 DOWN}"
30 PLOT$="{RVS}^{OFF}": REM SOLID SQUARE
40 INPUT "{HOME}UPPER^LEFT^X^(0-38)";X1
50 GOSUB 400: REM CLEAR PROMPT LINE
60 IF X1<0 OR X1>38 THEN 40
70 INPUT "{HOME}UPPER^LEFT^Y^(0-22)";Y1
80 GOSUB 400
90 IF Y1<0 OR Y1>22 THEN 70
95 REM PLOT UPPER LEFT CORNER POINT
100 X=X1: Y=Y1: GOSUB 10000
110 PRINT "{HOME}LOWER^RT^X^(";X1;"-38)";: INPUT X2
120 GOSUB 400
130 IF X2<X1 OR X2>38 THEN 110
```

```
140 PRINT "{HOME}LOWER^RT^Y^(Ø-";Y1;")";: INPUT Y2
150 GOSUB 400
160 IF Y2>Y1 OR Y2<Ø THEN 140
165 REM PLOT LOWER RIGHT CORNER POINT
170 X=X2: Y=Y2: GOSUB 10000
180 GOSUB 4000
190 INPUT "{HOME}ANOTHER^ONE^(Y/N)";A$
200 GOSUB 400
210 IF A$="Y" THEN 40
220 IF A$<>"N" THEN 190
230 PRINT "{CLR}"
240 END

395 REM CLEAR PROMPT LINE
400 PRINT "{HOME}{39 ^}"
410 RETURN

3985 REM DRAW RECTANGLE
3995 REM DRAW TOP AND BOTTOM
4000 FOR X=X1 TO X2
4010 Y=Y1: GOSUB 10000
4020 Y=Y2: GOSUB 10000
4030 NEXT X
4040 FOR Y=Y2 TO Y1
4050 X=X1: GOSUB 10000
4060 X=X2: GOSUB 10000
4070 NEXT Y
4080 RETURN

9995 REM PLOT CHARACTER$ STARTING AT X,Y
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN
```

Modifications

The following variation on lines 4000–4080 draws a solid rectangle:

```
3995 REM DRAW SOLID RECTANGLE (1=UPPER LEFT, 2=LOWER RIGHT)
4000 FOR X=X1 TO X2
4010 FOR Y=Y2 TO Y1
4020 GOSUB 10000
4030 NEXT Y
4040 NEXT X
4050 RETURN
```

We can use this routine to create bar graphs. The next program draws up to ten solid vertical bars. The user enters the number of bars, and the program determines their width automatically. It then draws each bar as soon as the user enters its height and color. We have coded the first eight colors just as on the top row of keys, but you do not have to press CTRL (e.g., 1 is black, 2 is white, 3 is red, etc.). Add eight to the key's number to obtain the Commodore-shifted color (9 is orange, 10 is brown, 11 is light red, 12 is dark gray, 13 is medium gray, 14 is light green, 15 is light blue, and 16 is light gray).

```
10 REM "BAR^GRAPH"
15 REM SET UP POSITIONING, COLOR STRINGS
20 VERT$="{HOME}{24 DOWN}"
30 CS$="{BLK}{WHT}{RED}{CYN}{PUR}{GRN}{BLU}{YEL}
        {1}{2}{3}{4}{5}{6}{7}{8}"
35 REM SOLID SQUARE FOR PRODUCING BARS
40 SQ$="{RVS}^{OFF}"
50 PRINT "{CLR}"
55 REM GET NUMBER OF BARS
60 PRINT "{HOME}{WHT}NUMBER^OF^BARS^(1-16)"
70 INPUT NB
80 IF NB<1 OR NB>16 THEN 50
85 REM START AT ORIGIN, SET BAR WIDTH TO FILL SCREEN
90 X1=0: Y2=0: WIDTH=INT(38/NB)
95 REM DRAW BARS
100 FOR J=1 TO NB
105 REM CLEAR PROMPT LINE
110 GOSUB 400
115 REM ASK FOR BAR HEIGHT
120 PRINT "{HOME}{WHT}BAR"; J; "HEIGHT^(0-22)"
130 INPUT Y1
140 IF Y1<0 OR Y1>22 THEN 110
145 REM CLEAR PROMPT LINE
150 GOSUB 400
155 REM ASK FOR BAR COLOR
160 PRINT "{HOME}BAR"; J; "COLOR(1-16)"
170 INPUT CC
180 IF CC<1 OR CC>16 THEN 150
185 REM CONVERT COLOR CODE TO COLOR
190 BC$=MID$(CS$,CC,1)
195 REM DETERMINE ENDING ROW, COLUMN
200 Y1=Y1-1: X2=X1+WIDTH-1
205 REM CHARACTER IS SOLID SQUARE IN SPECIFIED COLOR
210 PLOT$=BC$+SQ$
215 REM DRAW BAR IF HEIGHT>0
220 IF Y1>=0 THEN GOSUB 4000
225 REM START NEXT BAR WHERE CURRENT BAR ENDS
230 X1=X2+1
240 NEXT J
245 REM CLEAR PROMPT LINE
250 GOSUB 400
255 REM EXIT MESSAGE
260 PRINT "{HOME}{WHT}SPACE^BAR^TO^EXIT{7}"
270 GET K$: IF K$="" THEN 270
280 END

395 REM CLEAR PROMPT AND INPUT LINES
400 PRINT "{HOME}{39 ^}"
410 PRINT "{39 ^}"
420 RETURN

3995 REM DRAW SOLID RECTANGLE (1=UPPER LEFT, 2=LOWER RIGHT)
4000 FOR X=X1 TO X2
4010 FOR Y=Y2 TO Y1
4020 GOSUB 10000
4030 NEXT Y
4040 NEXT X
4050 RETURN

9995 REM PLOT CHARACTERS STARTING AT X,Y
10000 PRINT LEFT$(VERT$,25-Y); SPC(X) PLOT$;
10010 RETURN
```

RUN this program for the following example. Be sure to enter the color codes without pressing CTRL.

```
NUMBER OF BARS = 5
BAR 1 HEIGHT = 20, BAR 1 COLOR =  3 (RED)
BAR 2 HEIGHT = 10, BAR 2 COLOR = 14 (LIGHT GREEN)
BAR 3 HEIGHT =  4, BAR 3 COLOR = 13 (MEDIUM GRAY)
BAR 4 HEIGHT = 16, BAR 4 COLOR =  4 (CYAN)
BAR 5 HEIGHT =  8, BAR 5 COLOR =  8 (YELLOW)
```

You will see some color distortion along the edges of the bars, no matter how carefully you adjust your television set. Note particularly the boundary between the cyan and yellow bars. This picture is reproduced in the color section of this book as Plate 5.

DRAWING LINES BETWEEN ENDPOINTS

LINES, the next program, draws a series of lines between randomly selected points. You can use the subroutine in lines 995 through 1160 to draw a straight line between any two points. This subroutine thus lets the computer do the work of filling in the lines; all the user need do is specify the endpoints.

Program Name: "LINES"

Purpose

Draws random connecting lines. Figure 3–5 shows a typical output.

Techniques Demonstrated

Line drawing given endpoints. Random selection of points.

Procedure

The program first chooses two random points and draws a line between them. It then chooses another random point, and draws a line connecting it to the last endpoint. This continues until the user presses the RUN/STOP key.

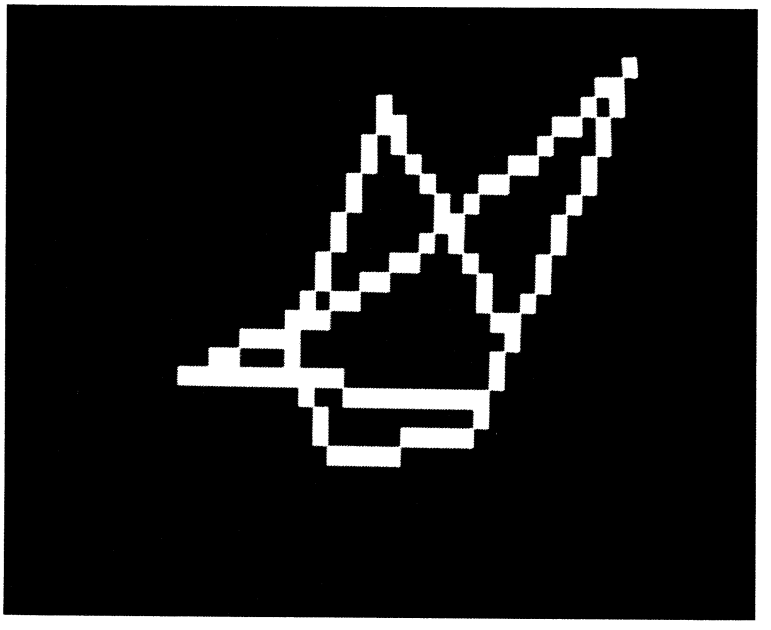


Figure 3–5 Typical output from LINES

Variables

A: X-axis intercept
B: Y-axis intercept
M: Slope of the line
PLOT$: Solid square (reversed space) character
S: Step from point 1 to point 2
VERT$: Vertical positioning string
X, Y: Coordinates of next solid square to print
X1, Y1: Horizontal and vertical positions of old point
X2, Y2: Horizontal and vertical positions of new point

Special Cases

The point-to-point line drawing routine in lines 995 through 1160 clips the line to keep it on the screen.

Brief Description

Lines 20 to 30 set up the vertical positioning string and the solid square character.

Lines 45 to 60 select a random point.

Lines 70 to 120 repeatedly select another random point and draw a line between it and the last point.

Lines 995 to 1160 draw a line from X1,Y1 to X2,Y2.

- Line 1000 chooses a large slope to handle a vertical line (the horizontal coordinate does not change).
- Line 1010 computes the line's slope if it is not vertical.
- Lines 1015 to 1090 plot the line using horizontal steps if it rises or falls gradually.
- Lines 1095 to 1160 plot the line using vertical steps if it rises or falls steeply.

Lines 9995 to 10010 put a solid square at coordinates X, Y.

Listing

```
5 REM "LINES" DRAW LINES TO RANDOM POINTS
20 VERT$="{HOME}{24 DOWN}"
25 REM REVERSED SPACE
30 PLOT$="{RVS}^{OFF}"
40 PRINT "{CLR}";
45 REM SELECT TWO RANDOM POINTS
50 X1=INT(RND(0)*39)
60 Y1=INT(RND(1)*25)
70 X2=INT(RND(1)*39)
80 Y2=INT(RND(1)*25)
85 REM DRAW LINE BETWEEN TWO POINTS
90 GOSUB 1000
95 REM NEW STARTING POINT=OLD ENDPOINT
100 X1=X2
110 Y1=Y2
120 GOTO 70

995 REM DRAW LINE FROM X1,Y1 TO X2,Y2
1000 M=1000
1005 REM COMPUTE M=SLOPE OF LINE
1010 IF X1<>X2 THEN M=(Y1-Y2)/(X1-X2)
1015 REM IF LINE IS STEEP, USE VERTICAL STEPS
1020 IF ABS(M)>1 THEN 1100
1025 REM NOT STEEP, SO STEP HORIZONTALLY
1035 REM COMPUTE Y-AXIS INTERCEPT
1040 B=Y1-M*X1
1045 REM PLOT LINE
1050 S=SGN(X2-X1)
1060 FOR X=X1 TO X2 STEP S
1070 Y=INT(M*X+B+0.5)
1075 IF X>=0 AND X<=38 AND Y>=0 AND Y<=24 THEN GOSUB 10000
1080 NEXT X
1090 RETURN
1095 REM STEEP, SO STEP VERTICALLY
1100 IF Y1=Y2 THEN RETURN
1105 REM COMPUTE X-AXIS INTERCEPT
1110 A=X1-Y1/M
1115 REM PLOT LINE
1120 S=SGN(Y2-Y1)
1130 FOR Y=Y1 TO Y2 STEP S
1140 X=INT(Y/M+A+0.5)
```

```
1145 IF X>=0 AND X<=38 AND Y>=0 AND Y<=24 THEN GOSUB 10000
1150 NEXT Y
1160 RETURN

9995 REM PLOT REVERSED SPACE AT X,Y
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN
```

Modifications

The screen fills rapidly, making it difficult to see individual lines being drawn. One way to distinguish the current line and to produce a more interesting picture is by choosing a random color for each new line. Add the following statements to LINES to make the output more colorful:

```
10 C$="{BLK}{WHT}{RED}{CYN}{PUR}{GRN}{BLU}{YEL}
     {1}{2}{3}{4}{5}{6}{7}{8}"
115 PRINT MID$(C$,INT(RND(1)*LEN(C$))+1,1);
```

The result is like an abstract painting, perhaps from someone's Cubist period. Unlike a painting, however, the screen is alive; its contents change constantly as new lines appear.

Notes

The low resolution of the screen produces the odd-looking staircase effect. Lines that are not exactly horizontal or vertical consist of straight sections with breaks between them. This is the best we can do with the 40-by-25 resolution. While the lines are not accurate geometrically, they look nice and demonstrate the principles involved. As we noted when discussing cartoon drawing, all pictures are ultimately irregular. What you see on the screen here is comparable to what you would see on a television set under a magnifying glass.

One interesting feature of Commodore 64 graphics is that the vertical and horizontal resolutions differ. Note that horizontal lines are broader than vertical ones. This is not surprising when you think about it. A television screen is roughly square, so dividing it into 25 spaces vertically and 40 horizontally should result in a space being about twice as tall as it is wide.

This difference creates problems if you are drawing rotating objects. Say, for example, you want to draw a spinning two-atom molecule (two circles connected by a straight line). If you make the distance between the two atoms constant (in terms of spaces), they will be approximately twice as far apart when the line is vertical as when the line is horizontal. To avoid this expansion and contraction as the molecule rotates, you must change the length of the line in spaces to keep its physical length constant.

SGN (the Sign function) indicates whether the line is going left (-1) or

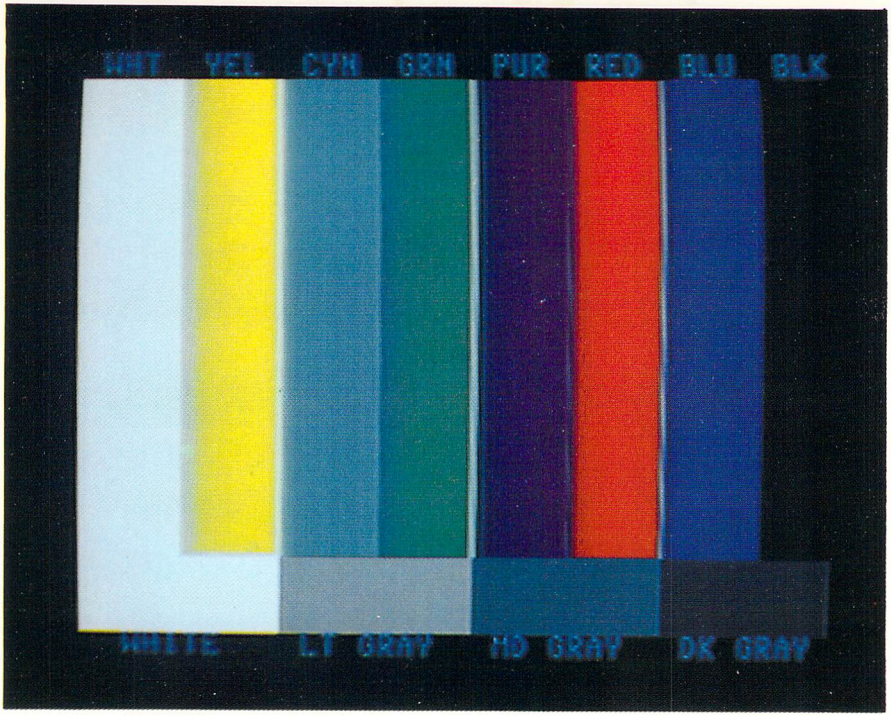
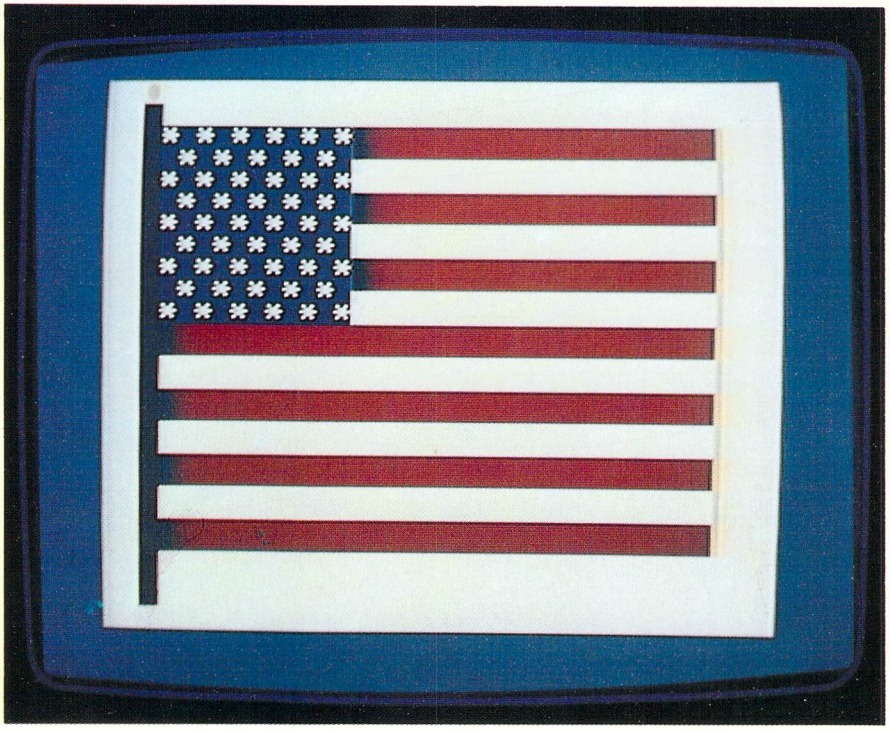


Plate 1 Color test pattern drawn by COLORBARS

Plate 2 United States flag by FLAG



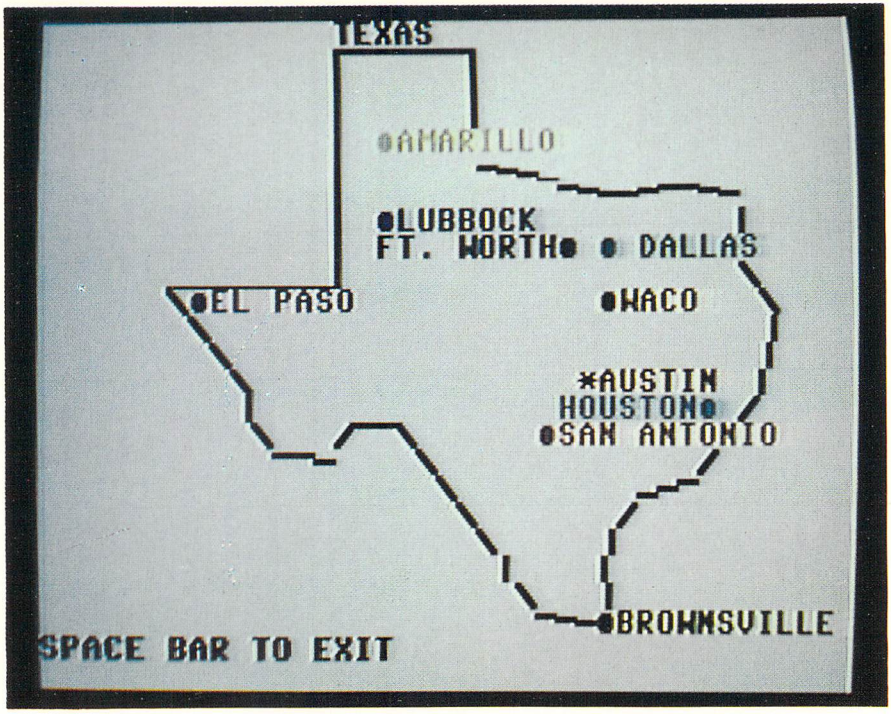
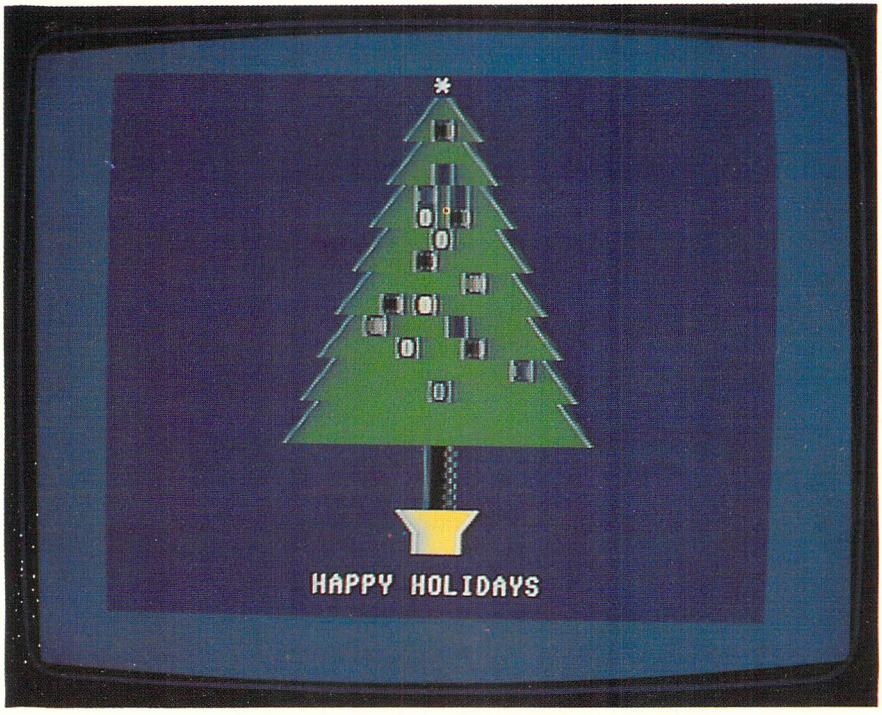


Plate 3 Outline of Texas with cities by TEXAS

Plate 4 Christmas tree by TREE



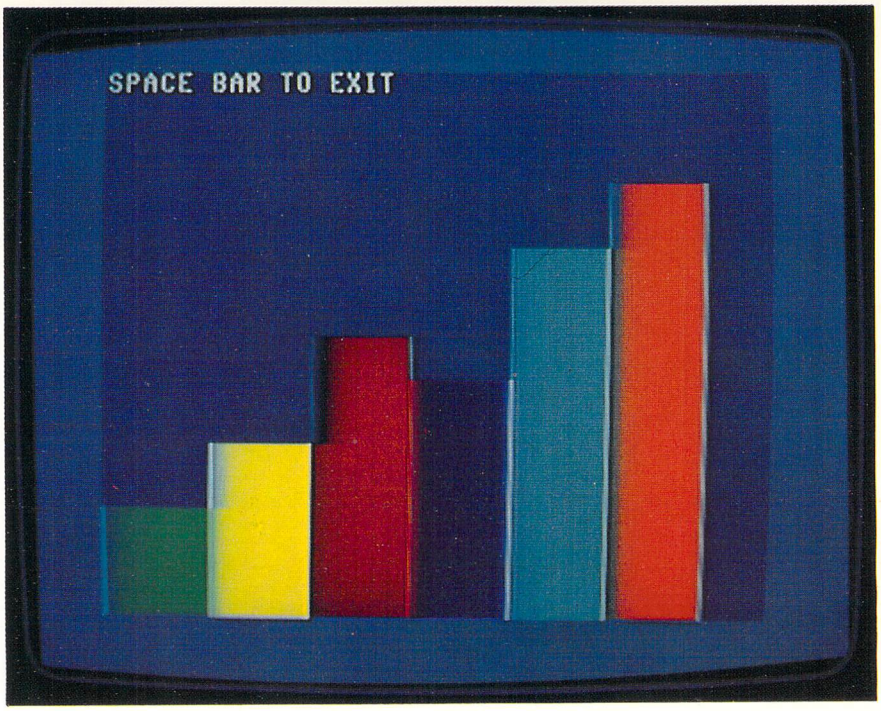
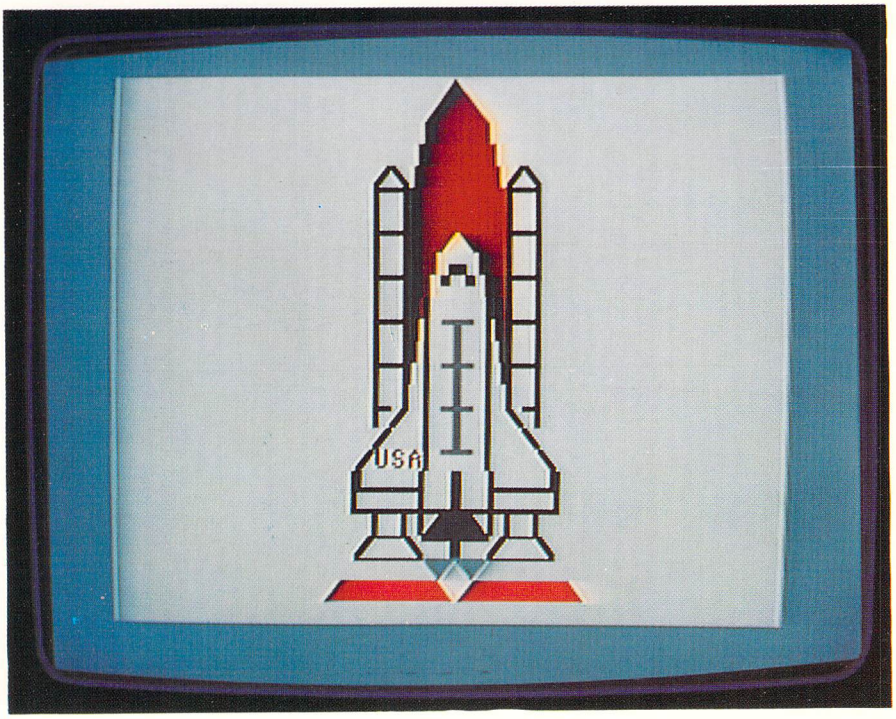


Plate 5 Bar graph modification to RECTANGLE

Plate 6 Space shuttle during LAUNCH



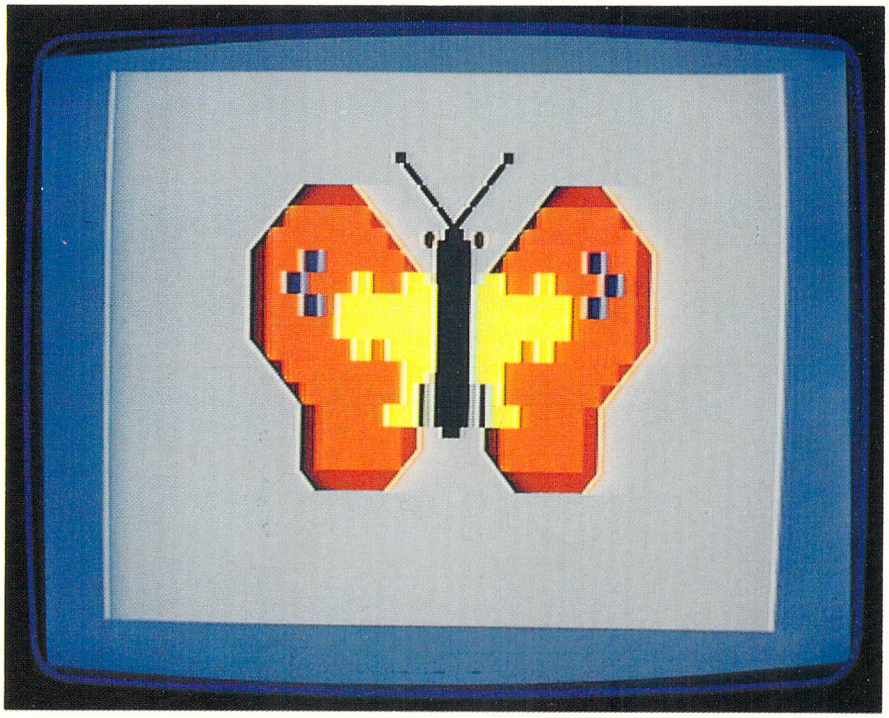
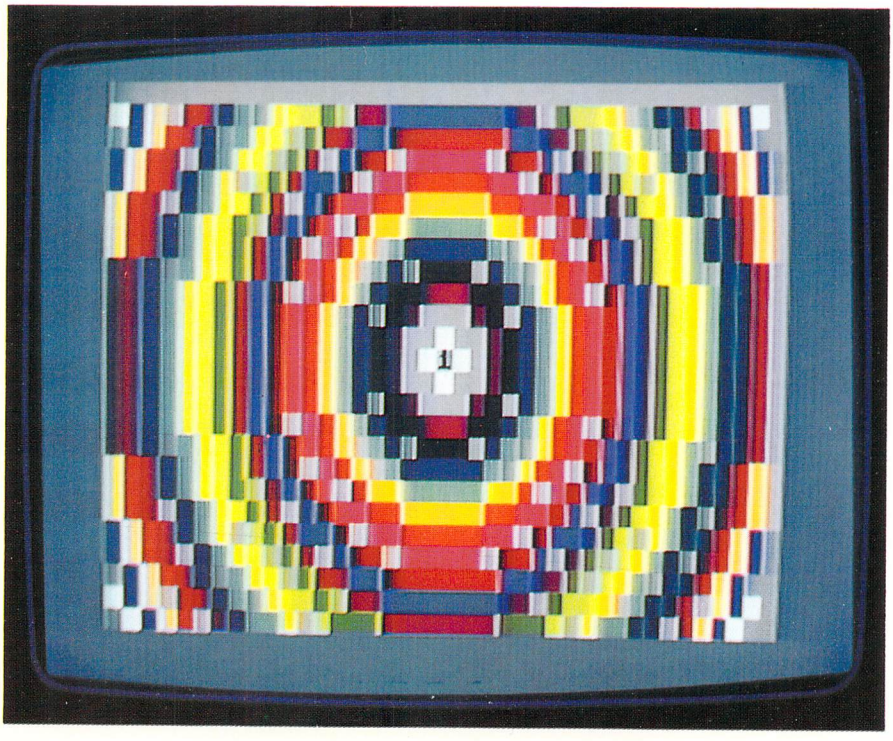


Plate 7 Butterfly drawn with PIC–EDIT

Plate 8 Bull's-eye drawn with SKETCH



right (1) in line 1050, and whether it is going down (−1) or up (1) in line 1120. SGN's value is −1 if its argument is negative, 0 if its argument is zero, and +1 if its argument is positive.

Occasionally, computer graphics involves a little mathematics, particularly geometry. You need not understand the mathematics in detail to use graphics routines, but it helps if you are at least familiar with the terminology.

For example, geometry tells us that two features identify a line (see Figure 3–6):

1. Where it would meet the vertical or horizontal axis (its *intercept*).
2. How fast it rises or falls (its *slope*).

You may think of the intercept as where the line starts; the axes just provide a standard reference. The slope describes the line's direction. A slope of 0 means the line does not rise at all; that is, it is horizontal. A large positive (or negative) slope means that the line rises (or falls) steeply; that is, it is close to vertical.

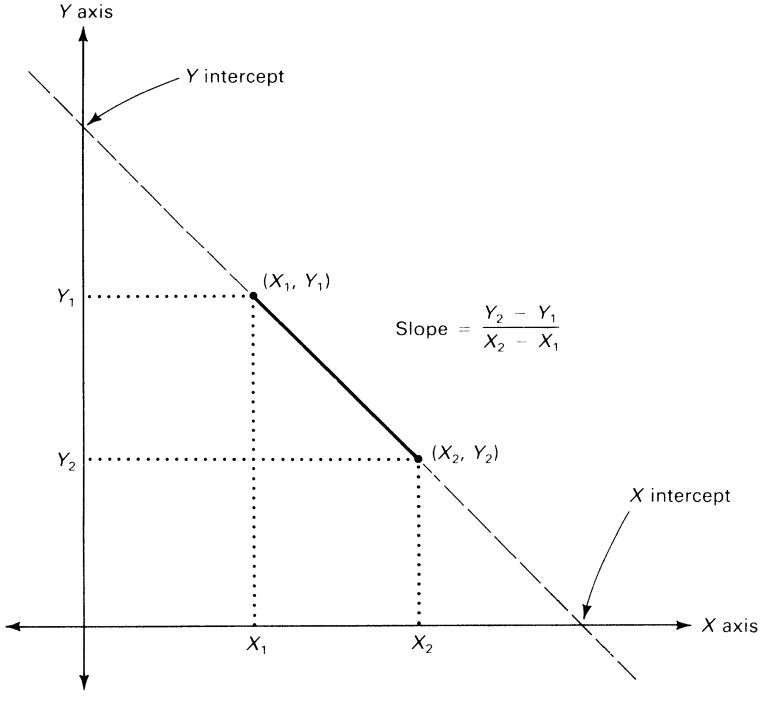


Figure 3–6 Slope and intercepts of a line

We can describe a line algebraically with the equation:

$$Y = M*X+B$$

(Note that the * means multiply.) The slope M is how far the line rises or falls in each horizontal step, while B—the Y intercept—is where the line meets the Y axis (that is, what its Y value is when X = 0).

We can also invert the equation algebraically to obtain:

$$X = Y/M-A$$

This is like turning a piece of graph paper on its side. M is the same as before. If a line rises M steps vertically for each horizontal step, it moves 1/M steps horizontally for each vertical step. A, the X intercept, describes where the line meets the X axis (that is, what its X value is when Y = 0).

The subroutine in lines 1000 through 1160 draws a line using either X=Y/M−A or Y=M*X+B. The intercepts (A and B) mean that, if the line were extended past its endpoints (X1,Y1 and X2,Y2), it would pass through the point A,0 on the X axis and 0,B on the Y axis.

When you start delving into geometry and algebra, remember to relate them to how a figure actually looks on paper or on the screen. For example, draw some lines with different slopes and intercepts to see how they look. You can quickly get lost in the X's and Y's if you do not relate them to their physical meanings.

References

Chapter 13 of the book *Modern Elementary Geometry* by James M. Moser (Prentice-Hall, 1971) discusses the basics of coordinate geometry and the equations of a line.

DRAWING LINES USING STARTING POINT, ANGLE, AND LENGTH

Another way to describe a line is by specifying its starting point, angle, and length. This method is useful when you want one line to be at a particular angle from another, such as 45° or 90°. For example, to form one side of a triangular mountain, we would need a line rising at an angle of 45° from the horizontal; 45° is halfway between horizontal and vertical. Program TURTLE contains a subroutine that draws lines this way.

Program Name: "TURTLE"

Purpose

Lets you draw lines by simulating a turtle with a pen attached to its belly. The turtle can be told to bring the pen up or down, turn, go forward,

or change color. If the turtle moves forward with the pen down, it draws a line. Figure 3–7 shows a typical picture drawn with a sequence of commands.

Techniques Demonstrated

Drawing a line given a starting point, an angle, and a length. Use of SIN and COS functions.

Procedure

Run the program. The dark circle in the center of the screen represents the turtle. The program asks you to press T for Turn, G for Go forward, U for pen Up, or D for pen Down. You may also press {CLR} to erase the screen, or C to change the color of the line to be drawn. When you press T or G, the program displays the turtle's current angle. An angle of 0° means the turtle is heading right, 90° up, 180° left, and 270° down.

Press a letter (C, D, G, T, or U). If you press T, the program asks you to enter a turn angle. Positive angles produce counterclockwise (left) turns, negative angles produce clockwise (right) turns. For example, an angle of 90° turns the turtle one quarter-circle left. If you press G, the program asks you how far the turtle should move. If you ask the turtle to go off the screen,

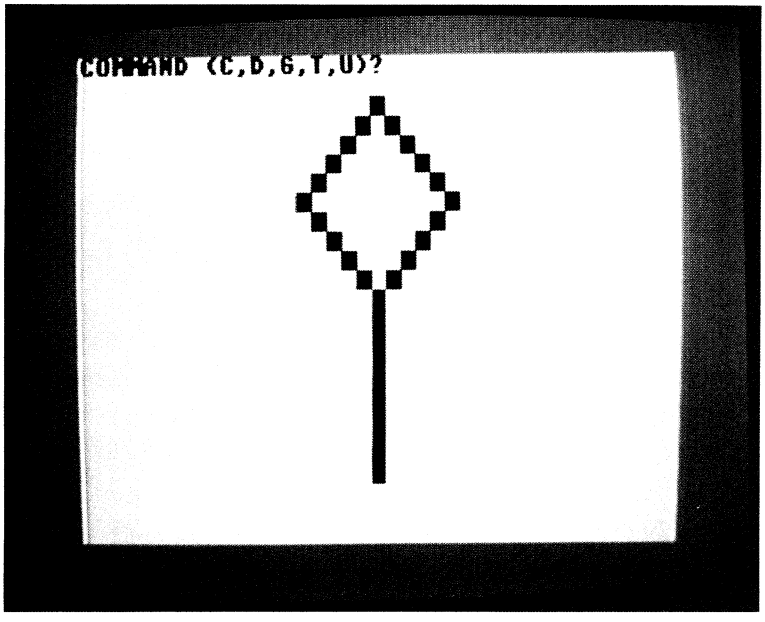


Figure 3–7 TURTLE draws a tree

it complains and refuses to move. If you press C, you will be asked to enter a color command (e.g., {RED} for red or {1} for orange). Be sure to press CTRL or Commodore when entering a color change; also press RETURN afterward.

Variables

ANGLE: Direction, in degrees, in which the turtle is facing
C$: Current command (C, D, G, T, or U) to the turtle
DIST: Distance to move the turtle
K: Counter
PC$: Turtle's color when it reached the current square
PD: Indicator of whether the pen was down when the turtle reached the current square (0 if it wasn't, 1 if it was)
PEN: Indicates whether pen is up (0) or down (1)
PLOT$: Characters to print
T: Angle to turn the turtle by
TC$: Turtle color
TURTLE$: Turtle character (open circle if pen up, solid disk if pen down)
VERT$: Vertical positioning string
X, Y: Coordinates at which to start printing turtle
X1, Y1: Current coordinates of turtle
X2, Y2: Coordinates where turtle will stop

Brief Description

Line 10 initializes the vertical positioning string.
Lines 20 to 60 start the turtle in the center of a cleared screen, facing up with its pen down. The turtle is initially black against a white background.
Lines 70 to 80 clear the prompt line and ask for a command.
Lines 90 to 130 flash the turtle and wait for a command.
Line 140 displays the command entered.
Lines 150 to 160 move the pen up or down in response to a U or D command.
Line 170 GOSUBs to the turn routine in response to a T command.
Line 180 GOSUBs to the forward movement routine in response to a G command.

Line 190 GOSUBs to the color change routine in response to a C command.

Lines 300 to 330 ask for and obtain a color change command.

Line 400 erases the prompt (top-most) line.

Lines 500 to 560 ask for the turn angle and turn the turtle.

Lines 600 to 620 ask for the distance to move the turtle.

Lines 630 to 1160 draw a line from X1, Y1 at angle ANGLE for distance DIST if the pen is down.

Lines 640 and 660 calculate the endpoint of the line.

Lines 650 and 670 detect when the endpoint is off the screen.

Line 680 draws the line if the pen is down.

Line 690 erases the old turtle if the pen is up.

Lines 700 to 710 fill in the old turtle's position if the pen was down.

Line 720 sets the new starting point to the old endpoint.

Lines 1000 to 1160 are the point-to-point line-drawing routine from LINES.

Listing

```
5 REM "TURTLE" DRAWING LINES AT ANGLES
10 VERT$="{HOME}{24 DOWN}"
15 REM CLEAR SCREEN
20 PRINT "{CLR}{BLK}": POKE 53281,15
25 REM START TURTLE IN MIDDLE, POINTING UP, PEN DOWN
30 ANGLE=90
40 X1=19: Y1=12
50 PD=0: PC$="{BLK}": TC$="{BLK}"
60 PEN=1: TURTLE$="Q{LEFT}"
65 REM CLEAR PROMPT LINE AND REQUEST COMMAND
70 GOSUB 400
80 PRINT "{HOME}{BLK}COMMAND^(C,D,G,T,U)?^";
85 REM FLASH TURTLE CURSOR
90 X=X1: Y=Y1: PLOT$=TC$+TURTLE$: GOSUB 10000
100 FOR K=1 TO 100: NEXT K
115 REM READ KEYBOARD
120 GET C$
130 IF C$="" THEN PRINT "^{LEFT}": FOR K=1 TO 100: NEXT K: GOTO 90
135 REM DISPLAY COMMAND ENTERED
140 PRINT "{HOME}{21 RIGHT}"; C$;
145 REM PUT PEN UP OR DOWN
150 IF C$="U" THEN PEN=0: TURTLE$="W{LEFT}"
160 IF C$="D" THEN PEN=1: TURTLE$="Q{LEFT}"
165 REM TURN TURTLE
170 IF C$="T" THEN GOSUB 500
175 REM MAKE TURTLE GO FORWARD
180 IF C$="G" THEN GOSUB 600
185 REM CHANGE TURTLE'S COLOR
190 IF C$="C" THEN GOSUB 300
200 GOTO 70
```

```
295 REM CHANGE PRINTING COLOR
300 GOSUB 400
310 PRINT "{HOME}PRESS^A^COLOR^COMMAND^KEY";
320 GET TC$: IF TC$="" THEN 320
330 RETURN

395 REM CLEAR PROMPT LINE AND SET PRINTING COLOR TO BLACK
400 PRINT "{HOME}{BLK}{39 ^}";
410 RETURN

495 REM TURN TURTLE BY T DEGREES
497 REM ERASE PROMPT LINE AND DISPLAY CURRENT DIRECTION
500 GOSUB 400
510 PRINT "{HOME}ANGLE="; ANGLE;
515 REM REQUEST TURN ANGLE
520 INPUT "{LEFT},TURN^BY"; T
530 ANGLE=ANGLE+T
540 IF ANGLE>=360 THEN ANGLE=ANGLE-360: GOTO 540
550 IF ANGLE <0 THEN ANGLE=ANGLE+360: GOTO 550
560 RETURN

595 REM MOVE TURTLE
597 REM ERASE PROMPT LINE AND DISPLAY CURRENT DIRECTION
600 GOSUB 400
610 PRINT "{HOME}ANGLE="; ANGLE;
615 REM REQUEST DISTANCE TO MOVE TURTLE
620 INPUT "{LEFT},^HOW^FAR"; DIST
625 REM RETURN IF NO MOTION
630 IF DIST=0 THEN RETURN
635 REM CALCULATE ENDPOINT AND CHECK LIMITS
640 X2=INT(X1+DIST*COS(π*ANGLE/180)+0.5)
650 IF X2<0 OR X2>38 THEN 740
660 Y2=INT(Y1+DIST*SIN(π*ANGLE/180)+0.5)
670 IF Y2<0 OR Y2>24 THEN 740
675 REM DRAW LINE IF PEN DOWN
680 IF PEN=1 THEN PLOT$=TC$+"{RVS}^{OFF}": GOSUB 1000
685 REM ERASE OLD TURTLE IF PEN UP
690 IF PEN=1 THEN 720
700 PLOT$="^": IF PD=1 THEN PLOT$=PC$+"{RVS}^{OFF}"
710 GOSUB 10000
715 REM MOVE TURTLE, REMEMBER COLOR, PEN UP OR DOWN
720 X1=X2: Y1=Y2: PD=PEN: PC$=TC$
730 RETURN
735 REM REPORT OFF-SCREEN ERROR
740 PRINT "{HOME}{RVS}LINE^GOES^OFF-SCREEN{OFF}^^"
745 REM PAUSE
750 FOR K=1 TO 1000: NEXT K
760 RETURN

995 REM DRAW LINE FROM X1,Y1 TO X2,Y2
1000 M=1000
1005 REM COMPUTE M=SLOPE OF LINE
1010 IF X1<>X2 THEN M=(Y1-Y2)/(X1-X2)
1015 REM IF LINE IS STEEP, USE VERTICAL STEPS
1020 IF ABS(M)>1 THEN 1100
1025 REM NOT STEEP, SO STEP HORIZONTALLY
1035 REM COMPUTE Y-AXIS INTERCEPT
1040 B=Y1-M*X1
1045 REM PLOT LINE
1050 S=SGN(X2-X1)
1060 FOR X=X1 TO X2 STEP S
1070 Y=INT(M*X+B+0.5)
1075 IF X>=0 AND X<=38 AND Y>=0 AND Y<=24 THEN GOSUB 10000
1080 NEXT X
1090 RETURN
```

```
1095 REM STEEP, SO STEP VERTICALLY
1100 IF Y1=Y2 THEN RETURN
1105 REM COMPUTE X-AXIS INTERCEPT
1110 A=X1-Y1/M
1115 REM PLOT LINE
1120 S=SGN(Y2-Y1)
1130 FOR Y=Y1 TO Y2 STEP S
1140 X=INT(Y/M+A+0.5)
1145 IF X>=0 AND X<=38 AND Y>=0 AND Y<=24 THEN GOSUB 10000
1150 NEXT Y
1160 RETURN

9995 REM PRINT CHARACTERS AT X,Y
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN
```

Example

The following sequence of TURTLE commands draws a tree with grass underneath it. Remember that the turtle starts in the center of the screen with its pen down:

C	{GRN}	Make printing color green.
T	−45	Turn right 45°.
G	7	Go forward 7 units (draw bottom, right-hand border of tree).
T	90	Turn left 90°.
G	7	Go forward 7 units (draw top, right-hand border of tree).
T	90	Turn left 90°.
G	7	Go forward 7 units (draw top, left-hand border of tree).
T	90	Turn left 90°.
G	7	Go forward 7 units (draw bottom, left-hand border of tree).
C	{2}	Make printing color brown.
T	−45	Turn right 45°.
G	10	Go forward 10 units (draw trunk).
C	{6}	Make printing color light green.
U		Pen up.
T	90	Turn left 90°.
G	15	Go forward 15 units (move to far right edge of grass).
D		Pen down.
T	180	Turn left 180° (reverse).
G	30	Go forward 30 units (draw grass).
T	90	Turn left 90°.
G	1	Go forward 1 unit (make grass thicker).
T	90	Turn left 90°.
G	30	Go forward 30 units (draw more grass).

If you misdirect the turtle, remember the following:

1. You can exit from an erroneous G or T command by entering 0 for the distance or angle, respectively.

2. You can always turn the poor turtle around (don't make him too dizzy) with a T of 180 degrees. You can then send him back where he came from, but be sure to turn him around again before continuing.

Modifications

You can easily add more commands to TURTLE. For example, let us make R (Reverse) equivalent to a turn by 180 degrees. The required changes are:

```
80 PRINT "{HOME}{WHT}COMMAND (C,D,G,R,T,U)?";
195 IF C$="R" THEN GOSUB 800
795 REM REVERSE TURTLE
797 REM ERASE PROMPT LINE AND DISPLAY CURRENT DIRECTION
800 GOSUB 400
810 PRINT "{HOME}ANGLE=";ANGLE;
820 PRINT " SPACE BAR TO REVERSE"
825 REM WAIT FOR USER TO CONFIRM COMMAND
830 GET K$: IF K$="" THEN 830
840 ANGLE=ANGLE+180
850 IF ANGLE>=360 THEN ANGLE=ANGLE-360
860 RETURN
```

Lines 810 to 830 simply show the user the current angle before continuing. In fact, pressing any key (not just the space bar) makes the program continue.

Lines 640 through 1160 form a useful routine that draws a line given its angle, length, and starting point. To convert these lines to a subroutine, delete lines 673 and 690, and enter:

```
680 PLOT$="{RVS}^{OFF}": GOSUB 1000
```

The line-drawing routine, together with a little mathematics, can produce interesting pictures (see Figure 3–8 for an example). Make changes suggested above, and replace lines 20 through 630 with the following main program:

```
5 REM "CARDIOID"
10 VERT$="{HOME}{24 DOWN}"
20 PRINT "{CLR}{RED}"
30 X1=19: Y1=18
40 FOR ANGLE=0 TO 360 STEP 6
50 DIST=ABS(14*SIN(π*ANGLE/360-π/4))
60 GOSUB 640
70 NEXT ANGLE
80 GET C$: IF C$="" THEN 80
90 PRINT "{CLR}{WHT}"
100 END
```

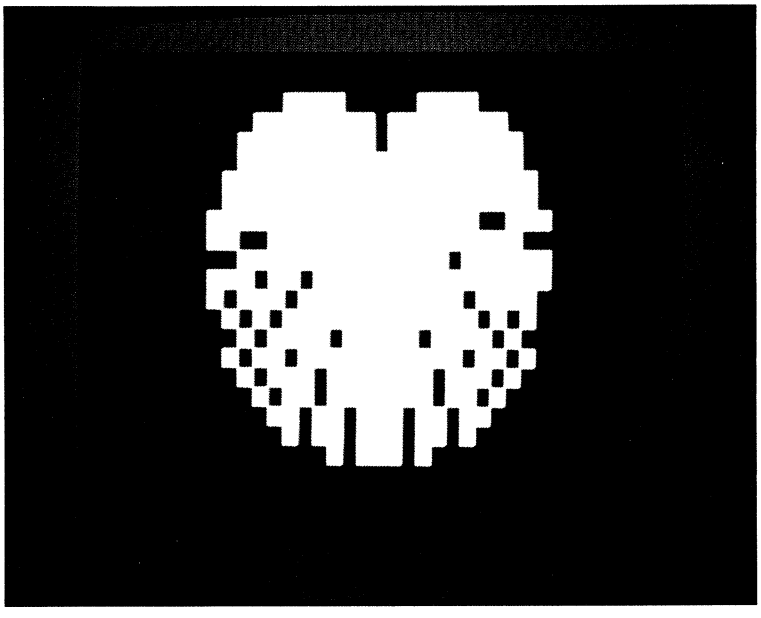


Figure 3–8 Heart drawn using starting position, angle, and distance

Notes

The difference between the vertical and horizontal resolutions is quite noticeable here. Make the turtle move 5 steps at its initial 90° heading, then turn it 90° and make it move another 5 steps. The vertical line is about twice as long and half as wide as the horizontal line.

You can also see the staircase effect clearly in TURTLE. Draw a line at a 45° angle. It looks like a nice, even staircase, since it rises at the same rate vertically and horizontally. Try some other angles. The resulting lines will look like stairs that are elongated in one direction; they may even have irregular-sized steps like the ones you see in rock formations or on hiking trails.

Be careful of the different kinds of input requests in TURTLE. When entering a command, do not press RETURN. It is not necessary because line 120 uses GET$ to read the keyboard. When entering color changes, angles, or distances, do press RETURN. It is necessary because lines 320, 520, and 620 use INPUT to read your response.

Lines 640 and 660 use the COS (cosine) and SIN (sine) functions. Trigonometry tells us that the distance between the starting point and the endpoint of a line of length D and angle A is:

D × sin(A) vertically
D × cos(A) horizontally

These are the line's vertical and horizontal *projections*, respectively. The Commodore 64's SIN and COS functions take angles in a measure known as radians, rather than the more familiar degrees. To convert degrees into radians, use:

angle in radians = (angle in degrees) × pi/180

Remember that you can enter the constant pi (3.14159) by pressing the up-arrow and SHIFT keys simultaneously.

"Why," you may ask, "is this program called 'TURTLE'?" The name is derived from LOGO, a popular computer language developed specifically for children. A standard LOGO activity is to control the motion of a small robot. Since this robot is usually squat and circular, it is called a turtle. The turtle can draw pictures with an attached ballpoint pen that can be moved up or down. Other languages, such as Apple II Pascal, use this simple way of generating pictures and refer to it as *turtle graphics*. Fortunately, no turtle organizations have yet objected to the typically crude representations of a turtle.

The heart-shaped figure we drew in the modification (Figure 3–8) is called a *cardioid*. You may have seen it before if you have drawn graphs using polar coordinates. The function we plotted to obtain it is:

distance = ABS(max radius × sin(angle/2 − 45))

References

There are many books on LOGO, including H. Abelson, *Apple LOGO* (BYTE/McGraw-Hill, New York, 1982), and D. H. Watt, *Learning with LOGO* (BYTE/McGraw-Hill, New York, 1983). Many LOGO-oriented articles appear in educational magazines such as *The Computing Teacher* (International Council for Computers in Education, University of Oregon, Eugene, Oregon 97403).

Chapter 2 of *Trigonometry for the Practical Man* (Van Nostrand Reinhold Company, New York, 1962), by J. E. Thompson, contains an adequate discussion of basic trigonometry.

DRAWING A CIRCLE USING PYTHAGORAS' THEOREM

Another familiar geometric form is the circle. The next program uses Pythagoras' theorem to draw one.

Program Name: "CIRCLE"

Purpose

Draws a circle with a specified center and radius. Figure 3–9 shows a typical output.

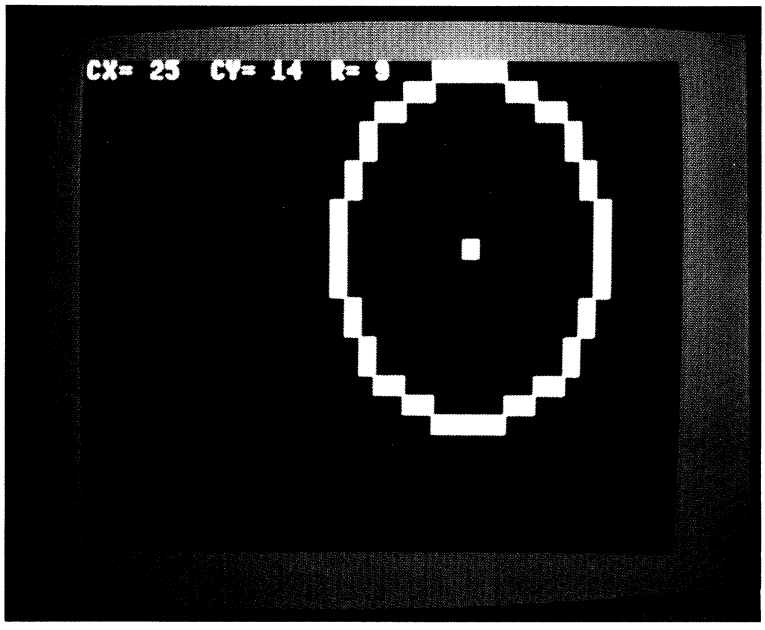


Figure 3–9 Sample output of Program CIRCLE

Techniques Demonstrated

Drawing of curves using algebraic formulas. Use of the SQR (square root) function. Clipping of drawings to keep them on the screen.

Procedure

The program first asks for the coordinates of the center, rejecting values that are off the screen, and next asks for the radius. It then draws the circle, clipping any part that would go off the screen.

Variables

CX, CY: Horizontal and vertical coordinates of center

K$: Keyboard input used to terminate the program

PLOT$: Solid square character

PX, PY: Horizontal and vertical coordinates of point on circle.

R: Radius of circle

RX, RY: Octant reflections of X, Y. In octants 2, 3, 6, and 7, RX equals X and RY equals Y. In the remaining octants, RX equals Y and RY equals X. An octant is one eighth of a circle (half of a quadrant).

VERT$: Vertical positioning string
X, Y: Coordinates at which to put a solid square
XP, YP: Horizontal and vertical distances from center to point on circle

Brief Description

Lines 35 to 70 ask for the coordinates of the center, rejecting values that are off the screen.
Lines 80 to 90 clear the screen and print the coordinates of the center.
Lines 95 to 120 ask for the radius of the circle and print it.
Line 130 plots the center of the circle.
Line 140 executes the circle-drawing routine.
Lines 150 to 160 wait for the user to press a key before exiting.
Lines 1595 to 1800 draw a circle.
Lines 1600 to 1680 calculate the coordinates of one octant (one eighth of the circle, the portion from 12 o'clock to 1:30 on a clock) and reflect it eight ways to form the complete circle.
Line 1600 steps X through one octant.
Line 1610 calculates the vertical distance from the center to the point on the circle.
Lines 1620 to 1640 draw the first four reflections of the octant.
Lines 1650 to 1670 draw the remaining four reflections.
Lines 1700 to 1790 draw four reflections of RX,RY around CX,CY.
Lines 9995 to 10010 draw a character at coordinates X, Y.

Listing

```
10 REM "CIRCLE" DRAW A CIRCLE USING PYTHAGORAS' THEOREM
20 VERT$="{HOME}{24 DOWN}"
30 PLOT$="{RVS}^{OFF}"
35 REM ASK FOR CENTER COORDINATES
40 INPUT "{CLR}CENTER^X^(0-38)"; CX
50 IF CX<0 OR CX>38 THEN 40
60 INPUT "CENTER^Y^(0-24)"; CY
70 IF CY<0 OR CY>24 THEN 60
80 PRINT "{CLR}";: REM CLEAR SCREEN
90 PRINT "CX="; CX; "^CY="; CY
95 REM ASK FOR RADIUS
100 INPUT "RADIUS"; R
110 PRINT "{HOME}{DOWN}{13 ^}"
120 PRINT "{HOME}" TAB(15) "^R=";R
125 REM DRAW CENTER
130 X=CX+0.5: Y=CY+0.5: GOSUB 10000
135 REM DRAW THE CIRCLE
140 GOSUB 1600
150 GET K$: IF K$="" THEN 150
```

```
160 PRINT "{CLR}"
170 END

1595 REM DRAW CIRCLE STEPPING HORIZONTALLY
1600 FOR XP=0 TO R/SQR(2)+0.5
1605 REM CALCULATE VERTICAL DISTANCE FROM CENTER
1610 YP=SQR(R*R-XP*XP)
1615 REM DRAW OCTANTS 2, 3, 6, AND 7
1620 RX=XP
1630 RY=YP
1640 GOSUB 1700
1645 REM DRAW OCTANTS 1, 4, 5, AND 8
1650 RX=YP
1660 RY=XP
1670 GOSUB 1700
1680 NEXT XP
1690 RETURN

1695 REM DRAW FOUR REFLECTIONS OF RX, RY AROUND CX,CY
1697 REM DRAW UPPER RIGHT QUADRANT
1700 X=CX+RX+0.5
1710 Y=CY+RY+0.5
1720 IF X>=0 AND X<=38 AND Y>=0 AND Y<=24 THEN GOSUB 10000
1725 REM DRAW LOWER RIGHT QUADRANT
1730 Y=CY-RY+0.5
1740 IF X>=0 AND X<=38 AND Y>=0 AND Y<=24 THEN GOSUB 10000
1745 REM DRAW LOWER LEFT QUADRANT
1750 X=CX-RX+0.5
1760 IF X>=0 AND X<=38 AND Y>=0 AND Y<=24 THEN GOSUB 10000
1765 REM DRAW UPPER LEFT QUADRANT
1770 Y=CY+RY+0.5
1780 IF X>=0 AND X<=38 AND Y>=0 AND Y<=24 THEN GOSUB 10000
1790 RETURN

9995 REM PUT A SOLID SQUARE AT X,Y
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN
```

Modifications

Lines 1595 through 1690 and 1695 through 1790 form a circle-drawing subroutine that any program can use.

Since the subroutine clips the circle anyway, lines 50 and 70 are unnecessary. With those lines (and line 130) removed, we could allow the center of the circle to lie off the screen. CIRCLE would then draw the part (an *arc*) that is on the screen. For example, setting CX = 45, CY = 12, and R = 10 draws roughly the left quarter of a circle, producing an effect like one edge of a crescent moon.

Notes

Watch how a circle appears. Try, for example, CX = 20, CY = 12, and R = 10. First, we see four straight segments along the axes' directions. Then the curved parts appear at the end of the straight segments and converge toward each other. It seems like a strange way to form a circle, but, of course, all the lines come together correctly.

At the same time, however, the result is clearly unimpressive. The circle isn't smooth at all. The low resolution makes it obvious that the curves consist of connected solid blocks. Furthermore, the difference between the horizontal and vertical resolutions produces an elongated figure that is much taller than it is wide. As usual, you should remember that the process would be the same if the resolution were better; what you see here is what you would see under a magnifying glass at higher resolution.

Line 1610 uses Pythagoras' theorem to calculate the vertical distance from the center for each horizontal step. A radius, a horizontal line from the center to its horizontal position (i.e., its horizontal projection), and a vertical line from there to its endpoint (i.e., its vertical projection) always form a right triangle (see Figure 3–10). Since the radius is the hypotenuse, the square of its length is equal to the sum of the squares of its projections' lengths. That is, we have:

$$X^2 + Y^2 = R^2$$

Subtracting X^2 from both sides gives us:

$$Y^2 = R^2 - X^2$$

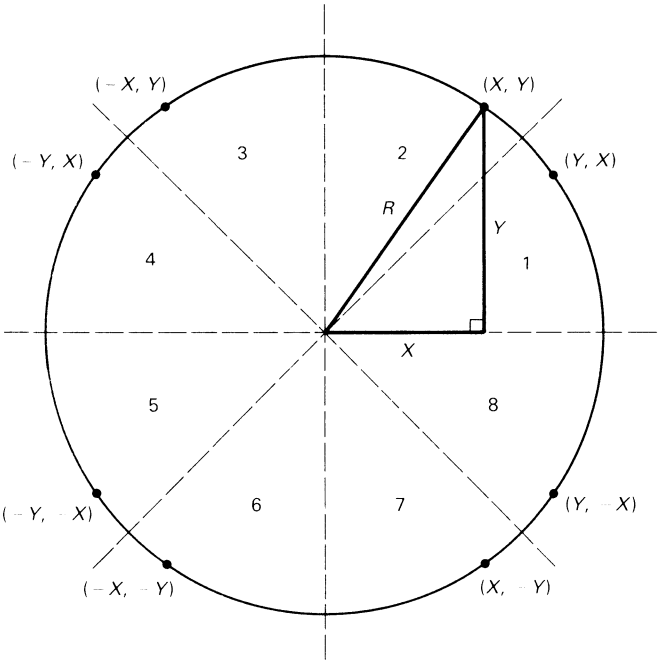


Figure 3–10 Plotting a circle using Pythagoras' theorem

Finally, taking the square root of both sides yields:

$$Y = \text{square root}(R^2 - X^2)$$

The square root function is SQR in BASIC.

Each calculation gives us eight points on the circle. This is because a circle is symmetrical around any line passing through its center. We can therefore divide it into eight equal wedges, known as *octants,* by passing four lines through its center as shown in Figure 3–10. Two lines are the horizontal and vertical axes, while the other two are diagonals. We have numbered the octants counterclockwise, beginning at the 3 o'clock position. The program calculates the coordinates of points in octant 2. The symmetrical reflections of these points about the lines are:

Octant	Coordinate	Octant	Coordinate
1	(Y, X)	2	(X, Y)
3	(−X, Y)	4	(−Y, X)
5	(−Y, −X)	6	(−X, −Y)
7	(X, −Y)	8	(Y, −X)

Note that line 1600 also uses Pythagoras' theorem to calculate how far to step X to obtain an eighth of a circle.

References

Plane Geometry, by Rolland R. Smith and James F. Ulrich (World Book Company, 1956, pp. 378–380), discusses Pythagoras' theorem.

4

GAMES

Video games are surely the most obvious application of computer graphics. Imagine how dull a space war, adventure, or auto racing game would be if we couldn't see the action. While engaged in an intergalactic battle, for example, the player would observe only a printed list of the coordinates of enemy ships, base stations, obstacles, and allies. An accurate shot with a laser gun would result in the message "EXPLOSION. ENEMY SHIP DESTROYED." Such a game would not attract much of a crowd at the local arcade or toy store.

This chapter explores techniques involved in computer games. It contains programs that draw game boards and pieces, throw dice, shuffle cards, construct mazes, bounce a ball, guide a lunar lander, and launch a rocket. It describes how to let players control game pieces, how to introduce random events, how to compute scores, and how to animate countdowns and launches. It also includes a brief discussion of the principles of game design.

DRAWING A GAME BOARD

Program TTT translates an old, simple, but always popular board game to the Commodore 64.

Program Name: "TTT"

Purpose

Lets two players play tic-tac-toe. Figure 4–1 shows a typical game in progress.

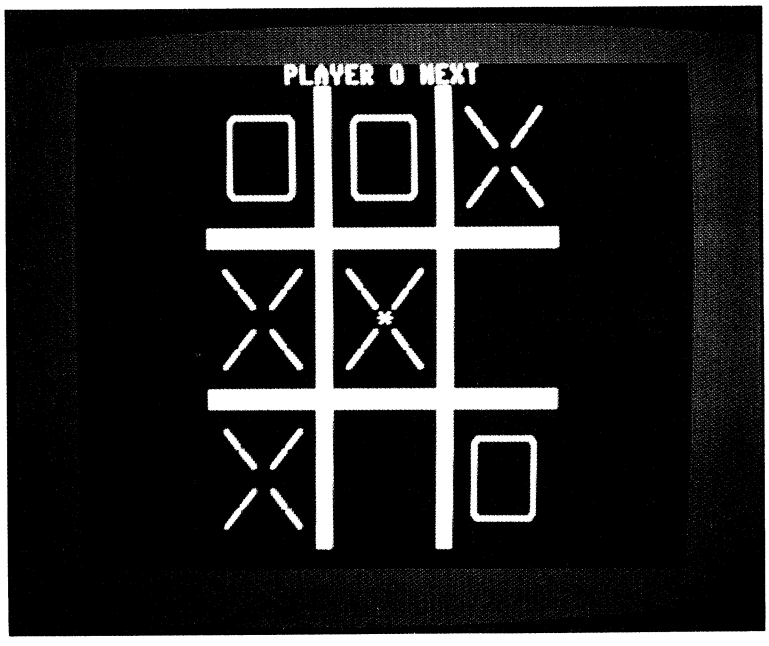


Figure 4–1 Tic-tac-toe board, X wins

Techniques Demonstrated

Controlling game pieces from the keyboard with GET, drawing game pieces, and making a character blink.

Procedure

The program draws a tic-tac-toe board and indicates it is player X's turn. A blinking asterisk, initially placed in the center, indicates the active box. The player can move the asterisk by pressing a CRSR key {UP}, {DOWN}, {LEFT}, or {RIGHT}. Pressing the X or O key draws the corresponding letter in the active box and indicates the other player's turn. Pressing the R key restarts the game. Pressing the RUN/STOP key terminates the program.

Variables

C$: Characters (space and *) required to produce a blinking asterisk

CN: Character number in the sequence that produces a blinking asterisk

J: Counter used to draw the board

KEY$: Key pressed by player

P$: Names of player pieces (X and O)
PN: Player number (1=X, 2=O)
VERT$: Vertical positioning string
X, Y: Coordinates of blinking asterisk

Special Cases

The initial program cannot tell when the game is over; modifications describe how to identify a winning move or a completely occupied board. The initial version simply continues until a player presses R to start over, or RUN/STOP to stop. The initial program also lets the player put a game piece wherever the blinking asterisk is, regardless of whether the square is already occupied. That is, a player could accidentally put an O on top of an X, or vice versa. Another modification restricts the players' moves to empty squares.

Brief Description

Lines 20 to 60 set up the strings and make it X's move initially.
Lines 65 to 120 clear the screen and draw the tic-tac-toe board.
Lines 160 to 180 blink the asterisk.
Lines 190 to 200 check whether the player has pressed a key.
Lines 215 to 250 move the asterisk if the player presses a CRSR key and the move is reasonable (i.e., on the game board).
Line 260 starts the game over if the player presses the R key.
Line 270 ignores keystrokes other than one indicating the playing of a piece (X or O).
Lines 280 to 290 print the player piece in the box containing the blinking asterisk.
Line 300 makes it the other player's turn.
Lines 9995 to 10010 put a character at coordinates X,Y.

Listing

```
5 REM "TTT" 2 PLAYER TIC-TAC-TOE, PLAYER X GOES FIRST
10 REM INITIALIZE POSITIONING, PLAYER, AND CURSOR STRINGS
20 VERT$="{HOME}{24 DOWN}"
30 PN=1: P$(1)="X": P$(2)="O"
40 CN=1: C$(1)="^": C$(2)="*"
45 REM INITIALIZE PLAYER PIECE STRINGS
50 EX$="{2 UP}{2 LEFT}M^^^N{DOWN}{5 LEFT}^M^N^{DOWN}
          {5 LEFT}^^V^^{DOWN}{5 LEFT}^N^M^{DOWN}{5 LEFT}N^^^M"
60 OH$="{2 UP}{2 LEFT}U***I{DOWN}{5 LEFT}-^^^-{DOWN}
          {5 LEFT}-^^^-{DOWN}{5 LEFT}-^^^-{DOWN}{5 LEFT}J***K"
65 REM DRAW BOARD
```

```
70 PRINT "{CLR}{8 DOWN}{8 RIGHT}{RVS}{23 ^}"
80 PRINT "{7 DOWN}{8 RIGHT}{RVS}{23 ^}"
90 PRINT "{HOME}{DOWN}";
100 FOR J=1 TO 23
110 PRINT TAB(15) "{RVS}^" TAB(23) "^"
120 NEXT J
125 REM CURSOR (BLINKING *) IN CENTER INITIALLY
130 Y=12
140 X=19
150 PRINT "{HOME}{13 ^}PLAYER^"; P$(PN); "^NEXT";
155 REM WAIT FOR PLAYER TO PRESS A KEY
157 REM BLINK ASTERISK
160 PLOT$=C$(CN)+"{LEFT}": GOSUB 10000
170 CN=3-CN
175 REM WAIT TO MAKE ASTERISK BLINK
180 FOR K=1 TO 100: NEXT K
185 REM LOOK FOR KEY COMMAND
190 GET KEY$
200 IF KEY$="" THEN 160
205 REM ERASE ASTERISK
210 PRINT "^{LEFT}";
215 REM TEST FOR CURSOR KEYS AND VALID MOVES
220 IF KEY$="{LEFT}" AND X>11 THEN X=X-8
230 IF KEY$="{RIGHT}" AND X<27 THEN X=X+8
240 IF KEY$="{UP}" AND Y<20 THEN Y=Y+8
250 IF KEY$="{DOWN}" AND Y>4 THEN Y=Y-8
255 REM TEST FOR END OF GAME
260 IF KEY$="R" THEN 30
265 REM IGNORE ALL OTHER KEYS
270 IF KEY$<>P$(PN) THEN 160
275 REM PUT PLAYER PIECE IN ACTIVE SPACE
280 IF KEY$="X" THEN PRINT EX$
290 IF KEY$="O" THEN PRINT OH$
295 REM SWITCH PLAYERS (X<-->O)
300 PN=3-PN
310 GOTO 150

9995 REM PRINT CHARACTERS STARTING AT X,Y
10000 PRINT LEFT$(VERT$,25-Y); SPC(X) PLOT$;
10010 RETURN
```

Modifications

You may change the current position indicator (the blinking asterisk) by changing the C$ character in line 40. For example, you might try:

1. C$(2)="Z" (a diamond instead of the asterisk).
2. C$(2)="A" (a spade instead of the asterisk).
3. C$(1)="+": C$(2)="V" (produces a rotating effect).
4. C$(1)="M": C$(2)="N" (produces a vibrating effect).
5. C$(1)="{RED}{RVS}^{OFF}{7}": C$(2)="{GRN}{RVS}^{OFF}{7}" (produces a square that blinks red and green).
6. C$(1)="{RED}S{7}": C$(2)="{RED}Q{7}" (produces a red beating heart).

Another way to flash the indicator is by reversing the asterisk; that is, make C$(1)="{RVS}*{OFF}".

Lines 220 through 250 move the piece if the player presses a CRSR key and the new position is on the board. While these keys are easy to remember (e.g., [UP] moves the asterisk up), they are awkward because [UP] and [LEFT] are uppercase. One alternative is to use L, R, U, and D for left, right, up, and down, respectively. Now you don't have to SHIFT, but you do have to search for the scattered control keys. A further complication is that L is actually on the right side of the keyboard, and R on the left. Another alternative is to pick a cluster such as W (up), A (left), S (right), and Z (down). The letters don't mean anything, but the keys are close together and positioned correctly relative to each other; that is, W is above the others, A is to the left, and so forth.

Lines 50 and 60 form large X's and O's using several graphics characters. You can change the way the pieces look by experimenting with these lines. One problem is that the program puts the blinking asterisk in the center of an X after drawing it. Then, when the next player moves the asterisk, the X is left with a hole in the middle. The solution to this problem is for line 210 to replace the asterisk with V if the square contains an X. We will describe how to make that replacement later.

To have the program restrict players to putting an X or O only in an unoccupied square, we need indicators of whether squares are occupied. We let each element of a 3 by 3 array OC be 0 if the corresponding square is empty and 1 if it is occupied. That is, OC(R,C) indicates whether the square in row R and column C is occupied. R and C both run from 1 to 3. The changes in the program are:

```
10   DIM OC(3,3)
52   REM ALL SQUARES INITIALLY UNOCCUPIED
54   FOR R=1 TO 3: FOR C=1 TO 3: OC(R,C)=0: NEXT C: NEXT R
56   REM START ASTERISK IN CENTER (ROW 2, COLUMN 2)
58   R=2: C=2

218 REM KEEP TRACK OF ASTERISK'S ROW AND COLUMN
220 IF KEY$="{LEFT}" AND X>11 THEN X=X-8: C=C-1
230 IF KEY$="{RIGHT}" AND X<27 THEN X=X+8: C=C+1
240 IF KEY$="{UP}" AND Y<20 THEN Y=Y+8: R=R+1
250 IF KEY$="{DOWN}" AND Y>4 THEN Y=Y-8: R=R-1
271  REM CANNOT PUT PIECE DOWN IF SPACE ALREADY OCCUPIED
272  IF OC(R,C)<>0 THEN 160
273  REM IF UNOCCUPIED, MARK SPACE AS OCCUPIED
274  OC(R,C)=1
```

A slight modification solves the problem of holes in X's. We make OC(R,C)=0 if the square is unoccupied, 1 if it contains an X, and 2 if it contains an O. The changes are:

```
273  REM IF UNOCCUPIED, MARK SPACE OCCUPIED BY X OR O
274  OC(R,C)=PN
211  REM PUT BACK CENTER OF X IF ONE IS IN SQUARE
212  IF OC(R,C)=1 THEN PRINT "V{LEFT}";
```

Now that we know which pieces are where, we can determine if we have a winner. The following lines simply check all possible combinations for a winning move:

```
5 REM "TTT GAME"
301  REM CHECK ROWS FOR WINNER
302  FOR RR=1 TO 3
305  REM NO WINNER IF SQUARE UNOCCUPIED
310  IF OC(RR,1)=0 THEN 330
315  REM WINNER IF ENTIRE ROW IS THE SAME
320  IF OC(RR,1)=OC(RR,2) AND OC(RR,2)=OC(RR,3) THEN 430
330  NEXT RR
335  REM CHECK COLUMNS FOR WINNER
340  FOR CC=1 TO 3
345  REM NO WINNER IF SQUARE UNOCCUPIED
350  IF OC(1,CC)=0 THEN 370
355  REM WINNER IF ENTIRE COLUMN IS THE SAME
360  IF OC(1,CC)=OC(2,CC) AND OC(2,CC)=OC(3,CC) THEN 430
370  NEXT CC
375  REM CHECK DIAGONALS FOR WINNER
378  REM NO WINNER IF CENTER SQUARE EMPTY
380  IF OC(2,2)=0 THEN 410
385  REM WINNER IF ENTIRE DIAGONAL IS THE SAME
390  IF OC(1,1)=OC(2,2) AND OC(2,2)=OC(3,3) THEN 430
400  IF OC(1,3)=OC(2,2) AND OC(2,2)=OC(3,1) THEN 430
405  REM NO WINNER, SWITCH PLAYERS (X<-->O)
410  PN=3-PN
420  GOTO 150
425  REM WE HAVE A WINNER
430  PRINT "{HOME}^^^PLAYER^"; P$(3-PN); "^WINS!^"
440  INPUT "E-EXIT,R-REPLAY"; K$
450  IF K$="R" THEN 30
460  IF K$<>"E" THEN 440
470  PRINT "{CLR}"
480  END
```

We could also add the following lines to end the game automatically if the entire board is filled:

```
51   OS=0: REM START WITH NO OCCUPIED SQUARES
274  OC(R,C)=PN: OS=OS+1
412  REM GO ON TO NEXT MOVE IF ANY UNOCCUPIED SQUARES
413  IF OS<9 THEN 150
415  REM END GAME IF NO UNOCCUPIED SQUARES
416  PRINT "{HOME}^^^BOARD FILLED!"
417  INPUT "^^^^E-EXIT,^R-REPLAY"; K$
418  IF K$="R" THEN 30
419  IF K$<>"E" THEN 417
420  GOTO 470
```

This still leaves the stalemate situation in which there are empty squares but neither player can win. As the program is now, the players must stop the game manually or fill all the squares. See if you can write a routine that recognizes a stalemate, reports it, and asks the players whether they want to quit or play again.

Notes

Line 170 switches CN (character number) from 1 to 2, or from 2 to 1. In general, to alternate N between VAL1 and VAL2, use:

$$N = VAL1 + VAL2 - N$$

Line 300 does the same thing with PN to switch players for the next move.

GAMES OF CHANCE

The next two programs are computerized versions of traditional game elements. DICE and CARDS provide an honest dice thrower and card dealer, respectively. After all, how can you bribe a Commodore 64?

Program Name: "DICE"

Purpose

Throws a pair of dice. Figure 4–2 shows a typical result.

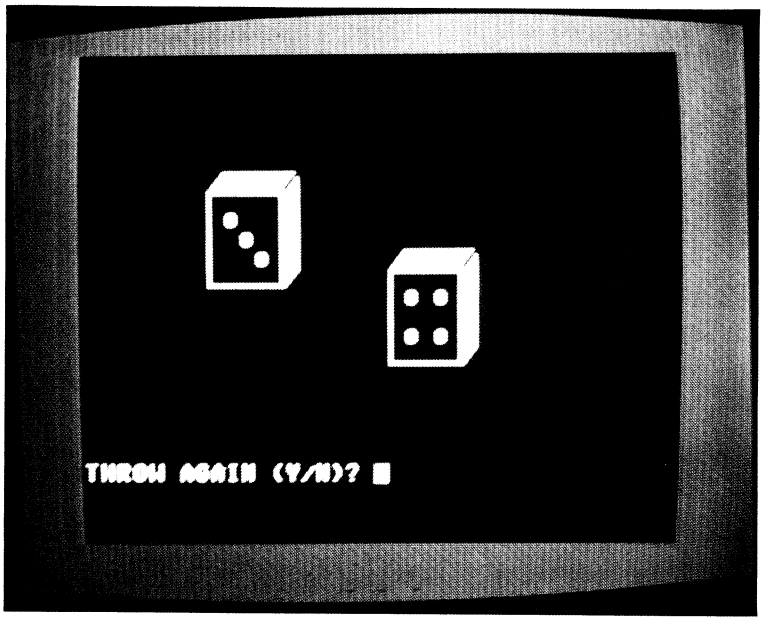


Figure 4–2 DICE throws a seven

Techniques Demonstrated

Use of graphics characters to draw dice. Generation of a random throw. Use of diagonal lines and triangles to produce the appearance of three-dimensional forms.

Procedure

The program generates two random die values and draws the thrown dice. It then asks the user whether to throw the dice again and repeats the procedure if the answer is Y.

Variables

D: Value of a die, randomly chosen between 1 and 6 inclusive
J: Counter to help draw the outline of a die
R$: User's response of Y or N
VERT$: Vertical positioning string
X, Y: Coordinates of the die's top left-hand corner

Brief Description

Line 10 ensures a different sequence of random numbers each time the program runs.
Lines 40 to 50 set the starting position for drawing the first die.
Line 60 calls the routine that draws a die face.
Lines 65 to 90 draw the second die to the right of and slightly below the first.
Lines 95 to 140 ask the user whether to draw another pair of dice and reject any answer other than Y or N.
Lines 6995 to 7130 compute a random number between 1 and 6, and draw the corresponding die.
Line 7000 sets the print position to the top left corner of the die.
Lines 7005 to 7040 draw the outline of a die.
Line 7050 calculates the random number.
Line 7060 prints the center spot.
Line 7080 prints the top left-hand and bottom right-hand spots.
Line 7100 prints the top right-hand and bottom left-hand spots.
Line 7120 prints the center top and center bottom spots.
Lines 9995 to 10010 set the current printing position. Note that PLOT$ is the empty (null) string here.

Listing

```
5 REM "DICE"  DRAW A PAIR OF DICE
10 J=RND(0): REM RANDOMIZE
20 VERT$="{HOME}{24 DOWN}"
30 PRINT "{CLR}"
35 REM DRAW FIRST DIE
40 X=8
50 Y=18
60 GOSUB 7000
65 REM DRAW SECOND DIE
70 X=20
80 Y=14
90 GOSUB 7000
95 REM DOES USER WANT ANOTHER THROW?
100 PLOT$="": X=0: Y=3
110 GOSUB 10000
120 INPUT "THROW^AGAIN^(Y/N)";R$
130 IF R$="Y" THEN 30
135 REM RESPONSE WAS NOT Y OR N.  TRY AGAIN
140 IF R$<>"N" THEN PRINT "PLEASE^ANSWER^Y^OR^N": GOTO 120
150 PRINT "{CLR}"
160 END

6995 REM DRAW A DIE AT CHARACTER POSITION X,Y
7000 PLOT$="": GOSUB 10000
7005 REM DRAW OUTLINE OF DIE
7010 PRINT "{RVS}£^^^^N"
7020 PRINT TAB(X) "Q{3 Y}P{RVS}^"
7030 FOR J=1 TO 3: PRINT TAB(X) "{G}^^^{M}{RVS}^": NEXT J
7040 PRINT TAB(X) "L{3 P}@£"
7045 REM GENERATE DIE VALUE RANDOMLY
7050 D=INT(RND(1)*6+1)
7055 REM DRAW SPOTS ON DIE
7060 IF 2*INT(D/2)<>D THEN GOSUB 10000: PRINT "{3 DOWN}{2 RIGHT}Q"
7070 IF D=1 THEN RETURN
7080 GOSUB 10000: PRINT "{2 DOWN}{RIGHT}Q{2 DOWN}{RIGHT}Q"
7090 IF D<4 THEN RETURN
7100 GOSUB 10000: PRINT "{4 DOWN}{RIGHT}Q{RIGHT}{2 UP}Q"
7110 IF D<6 THEN RETURN
7120 GOSUB 10000: PRINT "{2 DOWN}{2 RIGHT}Q{2 DOWN}{LEFT}Q"
7130 RETURN

9995 REM PRINT CHARACTERS STARTING AT X,Y
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN
```

Modifications

You can use this program with popular board games such as backgammon or Monopoly. In Monopoly, the player's next move is the sum of the two die values. The following additions make the program generate a Monopoly move:

```
62  REM SAVE FIRST DIE VALUE
64  D1=D
91  REM MOVE IS SUM OF DIE VALUES
92  M=D1+D
93  PRINT "{HOME}{19 DOWN}MOVE^IS"; M
```

One twist is that the Monopoly player gets another move automatically if he or she throws doubles. We could put this in the program by adding:

```
94  REM CHECK FOR DOUBLES
95  IF D1=D THEN 160

155  REM ANOTHER THROW AUTOMATIC IF DOUBLES THROWN
160  PRINT "{DOWN}DOUBLES--PRESS^SPACE^BAR^TO^THROW^AGAIN";
165  REM GIVE PLAYER TIME BY WAITING FOR A KEY
170  GET K$: IF K$="" THEN 170
180  GOTO 30
```

Line 170 gives the player a chance to see the previous throw and perhaps wait for the right moment before having the computer throw the dice again. Unfortunately, you can't breathe on computer-generated dice or shake them in your fist. Technology seldom makes allowances for superstitions.

The subroutine in lines 6995 through 7130 would also fit in programs that play dice games, such as craps. Here you may have messages commenting on different values, such as:

```
96  IF M=2 THEN PRINT "SNAKE EYES! BETTER LUCK NEXT TIME"
97  IF M=7 THEN PRINT "A WINNER! KEEP THE STREAK GOING"
```

Notes

Line 7010 and the final characters in lines 7020, 7030, and 7040 give the die its three-dimensional appearance. The keys here are the diagonal lines produced by a triangular character (shifted British pound sign) and reverse shifted N. These lines, together with a row and column of solid squares, give perspective to the die's top and side.

To understand the subroutine in lines 6995 through 7130, consider the spot patterns on a die (see Figure 4–3). Note that:

1. The center spot is present whenever the value is odd (1, 3, or 5). The test 2 * INT(D/2) <> D is true only if D is odd, since D/2 then contains a fraction (1/2) which INT discards. Thus line 7060 draws a center spot for an odd value. Since the center spot completes the pattern for 1, line 7070 exits if D is 1.

2. All other patterns have spots in the upper left-hand and lower right-hand corners. Line 7080 draws those spots. Since this completes the patterns for 2 and 3, line 7090 exits if D is less than 4.

3. Values 4, 5, and 6 have spots in the upper right-hand and lower left-hand corners. Line 7100 draws those spots. Since this completes the patterns for 4 and 5, line 7110 exits if D is less than 6.

4. Value 6 has spots in the middle on the top and bottom lines. Line 7120 draws those spots.

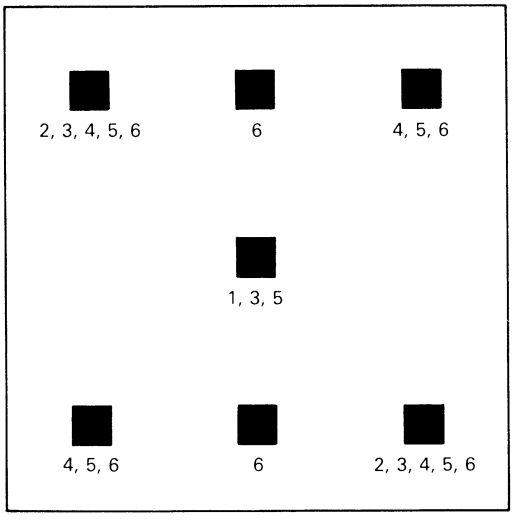


Figure 4–3 Spot positions on a die

Program Name: "CARDS"

Purpose

Deals and displays four cards. Figure 4–4 shows a typical result.

Techniques Demonstrated

Use of ON . . . GOSUB. Use of graphics symbols to draw playing cards. Also shows how to simulate a deck of cards, including shuffling and dealing.

Procedure

The program shuffles a deck of cards and deals four cards on the screen. Shuffling takes time, so the cards will not appear immediately.

Variables

CARD: Number of a card that has been dealt

CN: Card number

DECK: Array representing a deck of cards

FV$: String containing the face values of cards (ace, 1, 2, . . . , jack, queen, king)

J: Counter

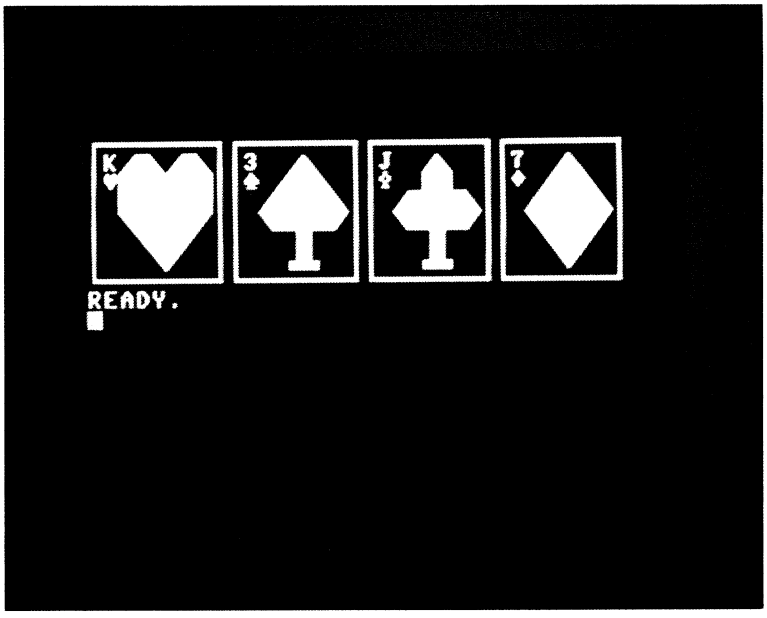


Figure 4–4 Four computer-dealt cards

RP: Random position in the deck (1 to 52)
SN: Suit number
TC: Top card position remaining in the deck (1 to 52)
VERT$: Vertical positioning string
X, Y: Coordinates of the top left-hand corner of a card
X2: Horizontal coordinate (column) plus 2

Special Cases

A face value of 10 is displayed as "T" to provide a single-character designation.

Brief Description

Lines 50 to 70 set up the deck by giving each element of DECK a card number. This program numbers cards from 0 to 51; 0 is the ace of hearts, 1 is the two of hearts, . . . ; 11 is the queen of hearts ("Off with her head!"), and 12 is the king of hearts. Similarly, 13 through 25 are ace through king of clubs; diamonds follow clubs, and, finally, spades follow diamonds.

Lines 85 to 150 deal 4 cards and display them.

Lines 8995 to 9070 shuffle the deck by considering each position separately.

Line 9010 chooses a random position somewhere between the front of the deck and the current position.

Lines 9020 to 9040 interchange cards between the random position RP and the current position TC. That is, the program puts the TCth card in position RP and the RPth card in position TC. This random switching shuffles the deck.

Line 9060 sets TC to the position of the top card in the undealt deck.

Lines 9095 to 9130 deal the TCth card and then add 1 to TC so it is the new top card. CARD is the card's value. If there are no cards left, line 9100 reshuffles the deck before the deal continues. Since this program deals only four cards, no reshuffle will ever occur. The test is handy, however, if you use this subroutine in a program that deals continuously.

Lines 9195 to 9760 draw the card with value CARD. Its top left-hand corner is at X, Y.

Line 9200 sets the current printing position to X, Y.

Lines 9210 to 9270 draw the outline of the card.

Line 9220 prints the symbol for the card's face value. The numerical face value is one more than the remainder left from dividing the card's number by 13. That value is used to select the face value symbol from FV$.

Line 9230 prints the symbol for the card's suit. It determines the suit by dividing the card's value by 13. The quotient (a number between 0 and 3) is used to select the suit character, and later, the graphic representation of the suit.

Line 9290 selects a routine to print the graphic suit symbol.

Lines 9395 to 9760 draw a large graphic representation of the suit symbol.

Listing

```
5 REM "CARDS"  DEAL 4 CARDS
10 PRINT "{CLR}": VERT$="{HOME}{24 DOWN}"
20 J=RND(0): REM RANDOMIZE
25 REM SET UP UNSHUFFLED DECK
30 DIM DECK(52)
40 FV$="A23456789TJQK"
50 FOR CN=1 TO 52
60 DECK(CN)=CN-1
70 NEXT CN
75 REM SHUFFLE CARDS
80 GOSUB 9000
85 REM DEAL 4 CARDS AND DISPLAY THEM
```

```
90 X=0: Y=20
100 FOR CN=1 TO 4
105 REM SHUFFLE DECK IF EMPTY
110 GOSUB 9100
115 REM DEAL A CARD
120 GOSUB 9200
125 REM MOVE TO NEXT PRINT POSITION
130 X=X+9
135 REM INCLUDING VERTICAL MOVE IF NECESSARY
140 IF X>30 THEN X=0: Y=Y-9
150 NEXT CN
160 END

8995 REM SHUFFLE DECK BY INTERCHANGING CARDS
9000 FOR TC=52 TO 2 STEP -1
9005 REM INTERCHANGE TWO CARDS
9010 RP=INT(RND(1)*TC)+1
9020 CARD=DECK(RP)
9030 DECK(RP)=DECK(TC)
9040 DECK(TC)=CARD
9050 NEXT TC
9060 TC=1
9070 RETURN

9085 REM DEAL A CARD
9095 REM RESHUFFLE DECK IF NO CARDS LEFT
9100 IF TC>52 THEN GOSUB 9000
9110 CARD=DECK(TC)
9120 TC=TC+1
9130 RETURN

9195 REM PRINT CARD STARTING AT COLUMN X, LINE Y
9197 REM POSITION TO COORDINATES X,Y
9200 GOSUB 10000
9205 REM DRAW TOP BORDER OF CARD
9210 PRINT "{A}*******{S}"
9212 REM DRAW LINE WITH FACE VALUE
9215 SN=INT(CARD/13)
9220 PRINT TAB(X) "|";MID$(FV$,CARD-13*SN+1,1);"{6 ^}|"
9225 REM DRAW LINE WITH SUIT SYMBOL
9230 PRINT TAB(X) "|";MID$("SXZA",SN+1,1);"{6 ^}|"
9235 REM DRAW SIDE BORDERS OF CARD
9240 FOR J=1 TO 4
9250 PRINT TAB(X) "|{7 ^}|"
9260 NEXT J
9265 REM DRAW BOTTOM BORDER OF CARD
9270 PRINT TAB(X) "{Z}*******{X}"
9275 REM DRAW LARGE SUIT SYMBOL IN CENTER
9280 GOSUB 10000
9285 X2=X+2
9288 REM PICK SUIT SYMBOL DRAWING FROM SUIT NUMBER
9290 ON SN+1 GOSUB 9400,9500,9600,9700
9300 RETURN

9385 REM PRINT SUIT SYMBOLS
9395 REM LARGE HEART
9400 PRINT "{DOWN}{2 RIGHT}{RVS}£^{*}£^{*}"
9410 PRINT TAB(X2) "{RVS}{6 ^}"
9420 PRINT TAB(X2) "{RVS}{6 ^}"
9430 PRINT TAB(X2) "{*}{RVS}^^^^{OFF}£"
9440 PRINT TAB(X2) "^{*}{RVS}^^{OFF}£"
9450 PRINT TAB(X2) "^^{*}£"
9460 RETURN

9495 REM LARGE CLUB
9500 PRINT "{DOWN}{2 RIGHT}^^{RVS}£{*}"
```

```
9510 PRINT TAB(X2) "^^{RVS}^^"
9520 PRINT TAB(X2) "{RVS}£^^^^{*}"
9530 PRINT TAB(X2) "{*}{RVS}^^^^{OFF}£"
9540 PRINT TAB(X2) "^^{RVS}{K}{OFF}{K}"
9550 PRINT TAB(X2) "^^{RVS}{V}{C}"
9560 RETURN

9595 REM LARGE DIAMOND
9600 PRINT "{DOWN}{2 RIGHT}^^{RVS}£{*}"
9610 PRINT TAB(X2) "^{RVS}£^^{*}"
9620 PRINT TAB(X2) "{RVS}£^^^^{*}"
9630 PRINT TAB(X2) "{*}{RVS}^^^^{OFF}£"
9640 PRINT TAB(X2) "^{*}{RVS}^^{OFF}£"
9650 PRINT TAB(X2) "^^{*}£"
9660 RETURN

9695 REM LARGE SPADE
9700 PRINT "{DOWN}{2 RIGHT}^^{RVS}£{*}"
9710 PRINT TAB(X2) "^{RVS}£^^{*}"
9720 PRINT TAB(X2) "{RVS}£^^^^{*}"
9730 PRINT TAB(X2) "{*}{RVS}^^^^{OFF}£"
9740 PRINT TAB(X2) "^^{RVS}{K}{OFF}{K}"
9750 PRINT TAB(X2) "^^{RVS}{V}{C}"
9760 RETURN

9995 REM PRINT STRING STARTING AT X,Y
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN
```

Modifications

Each card is 9 columns wide and 8 lines high. This would ordinarily mean that you could draw only 4 cards across the 40-column screen. If you are a card player, however, you know that you can identify a card without seeing all of it. If this were not possible, it would be difficult to hold a poker hand. Try changing line 130 to:

```
130 X=X+6
```

and see what happens when you RUN the program. Then try:

```
100 FOR CN=1 TO 10
130 X=X+3
```

Sometimes you can make objects look almost three-dimensional by overlapping them. The second modification lets the computer display two five-card poker hands at one time.

We can easily color the cards according to their suits. That is, make hearts and diamonds red, and spades and clubs black. The following modification adds this enhancement:

```
14 REM MAKE SCREEN MEDIUM GREY
15 POKE 53281,12
```

```
9207 PRINT "{WHT}";
9221 REM SET SUIT COLOR
9222 SC$="{BLK}"
9223 IF SN=0 OR SN=2 THEN SC$="{RED}"
9230 PRINT TAB(X) "_"; SC$; MID$("SXZA",SN+1,1);"{WHT}{6 ^}_"
9295 PRINT "{WHT}";

9400 PRINT "{RED}{DOWN}{2 RIGHT}{RVS}£^{*}£^{*}"
9500 PRINT "{BLK}{DOWN}{2 RIGHT}^^{RVS}£{*}"
9600 PRINT "{RED}{DOWN}{2 RIGHT}^^{RVS}£{*}"
9700 PRINT "{BLK}{DOWN}{2 RIGHT}^^{RVS}£{*}"
```

Notes

We determine the card's suit and value as follows:

1. Its suit number SN is INT(CARD/13), where CARD is its card number. Remember, hearts are cards 0 through 12, clubs 13 through 25, diamonds 26 through 38, and spades 39 through 51. So 13 goes into the card number 0 times if the card is a heart, once if it is a club, twice if it is a diamond, and three times if it is a spade. The INT function discards the remainder.
2. Its symbol in the FV$ string (A, 2, 3, 4, 5, 6, 7, 8, 9, T, J, Q, K) depends only on its numerical face value, not on its suit. Any ace uses the first character in the FV$ string, any 2 the second character, any 3 the third character, and so forth. A card's numerical face value is related to the difference between its number and that of its suit's ace. The card number of the ace in a suit is 13*SN, where SN is the suit number (0, 1, 2, or 3) determined as above. The character we want from the FV$ string is given by CARD − 13*SN + 1. The extra "+ 1" is there because A is the first character in FV$, not the character zero.

You can use the subroutines that shuffle, deal, and print cards in any program that plays card games. For example, a blackjack (twenty-one) game, with the computer as dealer, should be easy to write. To compute the value of a hand, add the values of all cards in it. The following program figures the blackjack value of a card:

```
1005  REM COMPUTE CARD VALUE FROM CARD NUMBER
1010  V=CARD-13*INT(CARD/13)+1
1020  IF V>10 THEN V=10
1030  RETURN
```

Of course, there is an added complication, since this routine returns an ace's blackjack value as 1. The program must figure out which value (1

or 11) to assign an ace, depending on the hand. That is, it sums the values of the other cards. If a value of 11 would make the total more than 21, the computer uses 1.

Note how lines 9020 through 9040 interchange two cards. We need CARD to hold one card value while line 9030 puts the other one in its place. Line 9040 then completes the interchange by moving CARD to its new position. Be careful—a program cannot interchange values without a place to keep one of them while it is moving the other.

Our standard notation is awkward with some graphics symbols, particularly those located on symbol keys such as +, −, *, and the British pound sign. For example, the minus sign, when underlined, looks like a misaligned equals sign, while underlined plus signs and asterisks look strange. It is difficult to tell whether the British pound sign is underlined; fortunately, you can assume it is, since we have no use for its normal meaning in this book. Things could be worse; imagine trying to type Chinese characters.

The Commodore 64 keyboard groups most similar or related graphics characters together as either left-hand or right-hand graphics. For example, all card symbols, curved corners, and angled corners are right-hand graphics while all quarter-spaces and solid partial spaces (e.g., **{I}**, **{J}**, **{K}**, **{L}**, and **{U}**) are left-hand graphics. The only major exception to this rule comes in the triangles. One (the upper left-hand triangle) is a right-hand graphic on the British pound sign key, while the other (the upper right-hand triangle) is a left-hand graphic on the asterisk key. Just reading the last sentence should be enough to cause total confusion. The only thing we can recommend is to be extremely careful when using the triangles. Watch the screen; if something unexpected appears, you probably pressed SHIFT instead of the Commodore key, or vice versa.

The ON . . . GOSUB in line 9290 works as follows. The computer first evaluates the formula following ON. In this case, the formula is INT(CARD/13)+1 (suit number + 1). It then uses that value to determine which subroutine to execute from the list following GOSUB. It executes the first entry if the value is 1, the second if the value is 2, and so on. Thus, line 9290 does a GOSUB to:

9400 if SN is 0 (i.e., the card is a heart).
9500 if SN is 1 (i.e., the card is a club).
9600 if SN is 2 (i.e., the card is a diamond).
9700 if SN is 3 (i.e., the card is a spade).

Remember, the suit number SN is INT(CARD/13). The single ON . . . GOSUB replaces four IF . . . THENs in this situation. If, for some reason, SN + 1 were not 1, 2, 3, or 4, the computer would simply continue to the next statement in the normal numerical order. That is, it would not do a GOSUB at all.

SCREEN MEMORY

The computer always keeps a record of what appears on the screen. It keeps this in what is (not surprisingly) called its *screen memory.* "Who cares?," you may ask. Well, this information is essential in games and other graphics applications. For example, a game program can examine screen memory and determine whether a car is moving along an open road or has crashed into a wall. It can also determine whether one player's soldier has entered the other player's fortress or has stepped on a land mine. Remember, the computer cannot gaze at the actual screen the way a player can.

PEEKing at Screen Memory

How does the program examine screen memory? Where is it, how do you look at it, and what does it contain? To look (or peek) into the computer's memory, the program uses the BASIC statement:

```
PEEK(MEMADD)
```

where MEMADD (or *memory address*) is the numerical designation of a particular place in memory. You can compare memory addresses to the street addresses used to identify houses or buildings for mail delivery or other services.

But which memory address do you use? The answer is that you must look up the proper address in this book, your *User's Guide,* or other documentation. This is much like looking up a person's or business's address in a telephone book or other directory. There is no way of guessing at computer memory addresses, any more than there is a way of guessing at street addresses in a city you've never visited. In both cases, the numbering is arbitrary.

The next question is, "When the computer looks at (or PEEKs at) an address, what does it find there?" If it is PEEKing into screen memory, it finds the screen code for a character. Table 4–1 lists these codes. You can compare them to Morse Codes for typed characters. Like a telegraph system, a computer has no way of recognizing or handling the actual characters, so it uses more suitable designations (numbers in the computer's case) instead.

Screen memory starts at address 1024. This is where the computer keeps the screen code for the character in the top left-hand corner. The program can PEEK at it with:

```
SC=PEEK(1024)
```

For convenience, we will refer to address 1024 by the name SMEM to remind us of its meaning.

TABLE 4–1 SCREEN CODES

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
@		0	U	u	21	*		42
A	a	1	V	v	22	+		43
B	b	2	W	w	23	,		44
C	c	3	X	x	24	—		45
D	d	4	Y	y	25	.		46
E	e	5	Z	z	26	/		47
F	f	6	[		27	ø		48
G	g	7	£		28	1		49
H	h	8	]		29	2		50
I	i	9	↑		30	3		51
J	j	10	←		31	4		52
K	k	11	SPACE		32	5		53
L	l	12	!		33	6		54
M	m	13	"		34	7		55
N	n	14	#		35	8		56
O	o	15	$		36	9		57
P	p	16	%		37	:		58
Q	q	17	&		38	;		59
R	r	18	'		39	<		60
S	s	19	(		40	=		61
T	t	20	)		41	>		62

SET 1	SET 2	POKE	SET 1	SET 2	POKE	SET 1	SET 2	POKE
?		63		T	84			106
		64		U	85			107
♠	A	65		V	86			108
	B	66	○	W	87			109
	C	67	♣	X	88			110
	D	68		Y	89			111
	E	69	♦	Z	90			112
	F	70			91			113
	G	71			92			114
	H	72			93			115
	I	73			94			116
	J	74			95			117
	K	75	SPACE		96			118
	L	76			97			119
	M	77			98			120
	N	78			99			121
	O	79			100		✓	122
	P	80			101			123
●	Q	81			102			124
	R	82			103			125
♥	S	83			104			126
					105			127

Moving Around the Screen

The computer keeps its records of other characters on the screen in consecutively higher-numbered addresses, moving first along the columns and then down the rows or lines. Thus address 1024 + 1 contains the character just right of the one in 1024, 1024 + 2 contains the character two columns right of the one in 1024, and so forth. Similarly, 1024 + 39 contains the character in the top right-hand corner, while 1024 + 40 contains the left-most character on the next to top line.

Thus, if address MEMADD in screen memory contains a particular character, we know that:

1. Address MEMADD + 40 contains the character immediately below it.
2. Address MEMADD − 40 contains the character immediately above it.
3. Address MEMADD − 1 contains the character to the left of it.
4. Address MEMADD + 1 contains the character to the right of it.

This assumes we are looking at an interior rather than a border character. Border characters obviously have no adjacent characters off the screen.

POKEing Screen Memory

Not only can a program look at screen memory, but it can also change it. For example, a game program could explode a target that had been struck by a bomb or crumple the fender of a car that hit a wall. The instruction that changes screen memory is:

```
POKE MEMADD, NEWCODE
```

where MEMADD is a memory address and NEWCODE is a code from Table 4–1. Note that we need PEEK to examine screen memory and POKE to change it.

As a simple example of POKEing screen memory, let us draw a solid line across the center of the screen (ten rows down from the top). To go down ten rows, we must move 400 (10 times 40) addresses beyond SMEM. A solid square (reversed space) is screen code 160 (see Table 4–1).

```
5    REM SOLID LINE NEAR CENTER OF SCREEN
10   PRINT "{CLR}";
15   REM MAKE BACKGROUND BLACK
20   POKE 53281,0
25   REM FIND START OF SCREEN MEMORY
30   SMEM=1024
35   REM FIND START OF LINE TEN ROWS DOWN
40   SLINE=SMEM+400
45   REM DRAW A SOLID LINE ACROSS SCREEN
```

```
50   FOR C=0 TO 39
60   POKE SLINE+C,160
70   NEXT C
75   REM WAIT HERE WHILE USER ADMIRES LINE
80   GET K$: IF K$="" THEN 80
90   POKE 53281,6: PRINT "{CLR}{7}";
100   END
```

You will see a solid white line against a black background. Press any key to regain control and restore the normal background.

AMAZE YOUR FRIENDS!

Program MAZE is a more complex example, involving both PEEKing and POKEing screen memory. It draws a maze by first filling the screen with borders around an interior of checkerboard characters. It then randomly selects a starting position and creates a path by burrowing in a direction in which it finds two consecutive checkerboards. This approach makes the program create narrow winding paths rather than large open areas.

Note that the program must PEEK at the screen memory to see which characters lie in a particular direction from where it is. It must later POKE the screen memory to burrow (that is, to replace checkerboards with spaces).

Program Name: "MAZE"

Purpose

Draws a maze. Figure 4–5 shows a typical example.

Techniques Demonstrated

Manipulating the screen display using PEEKs and POKEs of screen memory.

Procedure

RUN the program to draw a maze. It stops only after you press a key (such as the space bar).

Variables

ADDMOVE: Distance to the screen memory address for the screen position one character position up, down, left, or right. Value depends on the direction (see Notes for details)

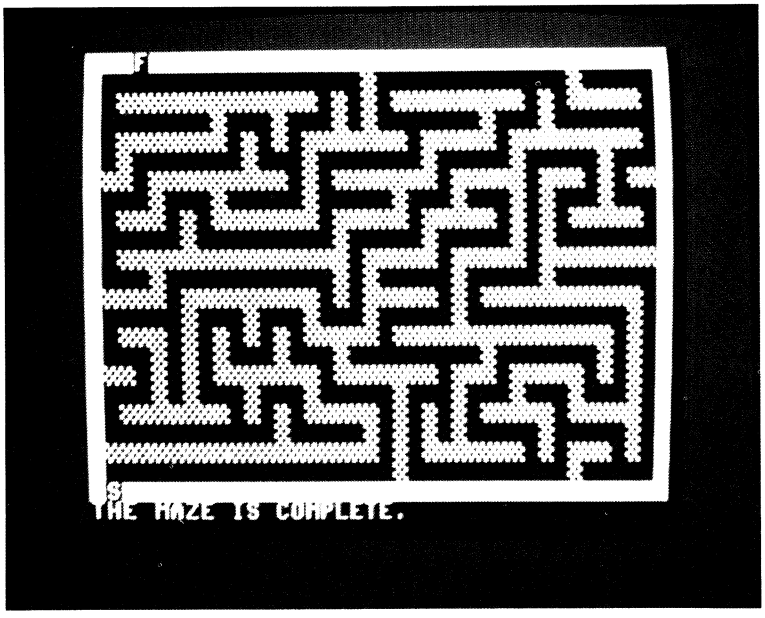


Figure 4–5 Find your way through the MAZE

BLANK: Screen code value of a space character from Table 4–1

CURRENT: Screen memory address of the maze position currently being investigated

DIR: Number (1 to 4) representing direction (right, left, down, or up) to move

FIRST: Screen memory address of the first maze position

J: Counter

K$: Keystroke used to terminate program.

LAST: Number of screen memory addresses on the path from FIRST to CURRENT

LF, LS: Screen codes for the letter F and S, respectively, from Table 4–1

PATH: List of screen memory addresses on the path from FIRST to CURRENT

S: Array containing distances in screen memory between characters that are a row or a column apart

SMEM: Address where screen memory begins

WALL: Screen code value of a checkerboard character, from Table 4–1

Brief Description

Line 40 makes the current path length zero initially.

Lines 50 to 80 set up the array S of distances in screen memory.

Lines 85 to 130 draw the borders of the maze and fill the interior with wall (checkerboard) characters.

Line 140 sets SMEM to the starting address of screen memory.

Line 150 sets FIRST to the screen memory address of a random position along the lower wall of the maze.

Line 160 starts the maze building at FIRST.

Line 170 puts a path (space) character in position CURRENT in screen memory.

Lines 175 to 200 choose a random direction (up, down, left, or right). If the next two characters in that direction are wall characters (checkerboards), this is a viable move.

Lines 205 to 240 try all four possible directions if the random move is not successful.

Lines 245 to 280 exit if the path length is zero, meaning the maze is complete. Otherwise, the program goes back one move and tries again to generate a viable move.

Lines 285 to 330 record CURRENT before carrying out a newly discovered, viable move.

Lines 335 to 360 finish the maze by marking the start and end points with the letters S and F, respectively, from Table 4–1. Line 340 places an S on the lower border just below the randomly chosen FIRST position. Line 350 places an F on the upper border at a randomly chosen position.

Line 370 waits for the user to press a key. This keeps the computer from moving the maze up the screen when it prints READY.

Listing

```
5 REM "MAZE"
10 DIM S(4), PATH(11*19)
15 REM SCREEN CODES (FROM TABLE 4-1)
20 BLANK=32: WALL=102
30 LS=19: LF=6
40 LAST=0
45 REM DISTANCES IN SCREEN MEMORY TO ADJACENT CHARACTERS
50 S(1)=1
60 S(2)=-1
70 S(3)=40
80 S(4)=-40
85 REM DRAW UPPER BORDER
90 PRINT "{CLR}{RVS}{39 ^}"
95 REM FILL INTERIOR
```

```
100 FOR J=1 TO 21
110 PRINT "{RVS}^{OFF}{37 +}{RVS}^"
120 NEXT J
125 REM DRAW LOWER BORDER
130 PRINT "{RVS}{39 ^}"
135 REM FIND STARTING ADDRESS OF SCREEN MEMORY
140 SMEM=1024
145 REM PICK RANDOM STARTING POINT ALONG LOWER WALL
150 FIRST=SMEM+21*40+1+2*INT(RND(0)*19)
160 CURRENT=FIRST
170 POKE CURRENT,BLANK
175 REM TRY ONE RANDOM MOVE
180 DIR=INT(RND(1)*4)+1
190 ADDMOVE=S(DIR)
195 REM MOVE VALID ONLY IF TWO WALLS IN THAT DIRECTION
200 IF PEEK(CURRENT+ADDMOVE)=WALL
        AND PEEK(CURRENT+2*ADDMOVE)=WALL THEN 290
205 REM RANDOM MOVE FAILED, TRY ALL MOVES
210 FOR DIR=1 TO 4
220 ADDMOVE=S(DIR)
230 IF PEEK(CURRENT+ADDMOVE)=WALL
        AND PEEK(CURRENT+2*ADDMOVE)=WALL THEN 290
240 NEXT DIR
245 REM NO GOOD MOVES, TRY BACKING UP
250 IF LAST=0 THEN 340
260 CURRENT=PATH(LAST)
270 LAST=LAST-1
280 GOTO 180
285 REM GOOD MOVE DISCOVERED, RECORD CURRENT, THEN MOVE
290 LAST=LAST+1
300 PATH(LAST)=CURRENT
310 POKE CURRENT+ADDMOVE,BLANK
320 CURRENT=CURRENT+2*ADDMOVE
330 GOTO 170
335 REM MARK START WITH S AND FINISH WITH F
340 POKE FIRST+40,LS
350 POKE SMEM+1+2*INT(RND(1)*19),LF
360 PRINT "THE^MAZE^IS^COMPLETE.";
370 GET K$: IF K$="" THEN 370
380 END
```

Modifications

Insert

```
7  POKE 53281,1: PRINT "{BLK}";
```

to make the passageways white and the border black.

The current program tries one random direction before testing all four directions. You can draw a more complex maze by having the program try more than one random direction. To do this, enter the following lines:

```
180 FOR J=1 TO 3
185 DIR=INT(RND(1)*4)+1
205 NEXT J
```

Now the program will try three random moves. Unfortunately, this slows it down significantly.

We can now add a routine that lets you work your way through the maze using the CRSR keys. Press any key (say, the space bar) to start and then move the blinking asterisk toward the endpoint. Your score is the total number of CRSR keystrokes you use.

```
5   REM "MAZE GAME"
25   REM SCREEN CODES FOR S, F, ASTERISK
30   LS=19: LF=6: AS=42
360  PRINT "PRESS^ANY^KEY^TO^START";
365  REM WAIT FOR USER TO PRESS A KEY
370  GET K$: IF K$="" THEN 370
375  REM ERASE MESSAGE
380  PRINT "{HOME}{23 DOWN}{39 ^}";
385  REM START PLAYER (BLINKING ASTERISK) JUST ABOVE S
390  PLYR=FIRST
400  NMOVES=0
405  REM BLINK ASTERISK
410  POKE PLYR,AS
420  FOR K=1 TO 50: NEXT K
430  POKE PLYR,BLANK
440  FOR K=1 TO 50: NEXT K
445  REM LOOK FOR KEYSTROKE
450  GET K$
460  IF K$="" THEN 410
465  REM LOOK FOR CRSR KEYS
470  DIR=0
480  IF K$="{RIGHT}" THEN DIR=1
490  IF K$="{LEFT}" THEN DIR=2
500  IF K$="{DOWN}" THEN DIR=3
510  IF K$="{UP}" THEN DIR=4
515  REM IGNORE OTHER KEYS
520  IF DIR=0 THEN 410
525  REM ADD 1 TO PLAYER'S SCORE
530  NMOVES=NMOVES+1
535  REM GET CHARACTER IN DIRECTION OF MOVE
540  PNXT=PLYR+S(DIR)
550  CNXT=PEEK(PNXT)
555  REM MOVE PLAYER IF WAY IS FREE (CHARACTER IS BLANK)
560  IF CNXT=BLANK THEN PLYR=PNXT
565  REM END OF MAZE IF MOVE IS TO LETTER F
570  IF CNXT<>LF THEN 410
575  REM THROUGH MAZE - PRINT SCORE
580  PRINT "{HOME}{23 DOWN}SCORE^IS"; NMOVES; "?=HELP"
585  REM WAIT FOR PLAYER TO PRESS KEY
590  GET K$: IF K$="" THEN 590
595  REM ? FOR EXPLANATION OF COMMANDS
600  IF K$="?" THEN PRINT "{UP}E=EXIT,N=NEW,S=SAME^^"
605  REM N MEANS NEW MAZE
610  IF K$="N" THEN 90
615  REM S MEANS REPEAT SAME MAZE
620  IF K$="S" THEN 380
625  REM REJECT KEYS OTHER THAN E, N, OR S
630  IF K$<>"E" THEN 590
640  PRINT "{CLR}";
650  END
```

When you finish the maze, you can press:

1. E to exit the program and clear the screen.
2 N to try a new maze.

3. S to try the same maze again.
4. ? to see a brief description of the command options.

To minimize your score, check whether you need SHIFT before pressing a CRSR key. Each incorrect use of SHIFT (e.g., {UP} instead of {DOWN} or {RIGHT} instead of {LEFT}) will cost you a point. Also, wait for the asterisk to move before pressing another CRSR key. The Commodore 64 saves keystrokes it cannot immediately handle, so it will remember every move you make. Thus, if you press a CRSR key several times quickly, the 64 will catch up with you later; the catch-up may send the asterisk past your goal or leave it stuck in a corner. While this memory is a nuisance here, it does help the computer keep up with fast typists.

A better scoring method is to determine elapsed time. This involves using the Commodore 64's built-in clock string TI$. TI$ consists of six digits. From left to right there are two for hours, two for minutes, and two for seconds. To measure elapsed time in minutes and seconds, we need only subtract the starting values from the ending values. The extra statements are:

```
395  REM GET STARTING TIME - MINUTES, SECONDS
400  M1=VAL (MID$(TI$,3,2)): S1=VAL (RIGHT$(TI$,2))
571  REM GET ENDING TIME - MINUTES, SECONDS
572  M2=VAL (MID$(TI$,3,2)): S2=VAL (RIGHT$(TI$,2))
576  REM DETERMINE ELAPSED TIME
578  REM FIRST MINUTES
580  MD=M2-M1
581  REM THEN SECONDS WITH BORROW IF NECESSARY
582  SD=S2-S1: IF SD<0 THEN SD=SD+60: MD=MD-1
583  REM PRINT ELAPSED TIME AS SCORE
584  PRINT "{HOME}{23 DOWN}TIME:"; MD; "M"; SD; "S^?=HELP"
```

Note the following:

1. Minutes are the 2 middle characters in TI$, starting with character 3, while seconds are the 2 right-most characters.
2. The VAL function converts strings of digits to numerical values that lines 580 and 582 can use in addition and subtraction.
3. Line 582 borrows a minute (60 seconds) if the number of elapsed seconds is negative.

Notes

You may have noticed that the program slows down considerably toward the end. This is because it is spending a lot of time rejecting moves and backtracking. Note how it occasionally finds a broad expanse of walls and puts an extra dead-end passageway through it. Even after finishing the

maze, the program takes a while to verify that there are no more valid moves. Thus, you will see a delay before the start and finish markers appear.

Looking up screen codes in Table 4–1 is a nuisance. Unfortunately, the Commodore 64 does not have a screen code function. It does, however, have a function that calculates a related code called ASCII, the American Standard Code for Information Interchange. Appendix D contains a listing of the ASCII characters. The following program sets C to the screen code of the character C$ by using the 64's ASCII (ASC) function:

```
5000 C=ASC(C$)
5010 IF C<64 THEN RETURN
5020 IF 96<=C AND C<=127 THEN C=C-32: RETURN
5030 C=C-64
5040 RETURN
```

ADDMOVE is the distance in memory between CURRENT and the screen memory address containing the next character in a particular direction. Note that we need not worry about there being no next character, since we have specifically excluded the maze's borders.

References

Pages 62 through 64 and 132 through 134 of the *Commodore 64 User's Manual* describe screen memory and screen codes.

David Matuszek describes other approaches to maze construction in "How to Build a Maze," in the December 1981 issue of *BYTE* magazine.

SIMULATION OF MOTION

So far, our programs have not involved motion. Computers can make things move in the same way cartoonists do. A cartoonist animates an object by drawing pictures of it in several slightly different locations and showing them in quick succession. Similarly, by drawing an object, erasing it, and redrawing it in a slightly different location, the computer can make the object appear to move. Program BOUNCE illustrates how this works for a ball.

Program Name: "BOUNCE"

Purpose

Displays a ball that starts at a given position with a given velocity. It bounces off the walls of a box, slowing down gradually because of the effects of the collisions.

Techniques Demonstrated

Simulation of motion.

Procedure

The ball starts in the top left-hand part of a box with initial velocity to the right. Under the influence of gravity, it falls until it hits the floor and bounces. The bounce makes the ball lose speed. It continues to bounce around the box until the user presses RUN/STOP. Eventually, it will start rolling back and forth at the bottom, moving slower and slower, until it stops completely.

Variables

AV: Acceleration of gravity (vertical)
COL: Horizontal position for border character
DAMPH: Damping (loss of speed) caused by collision with wall during a horizontal bounce
DAMPV: Damping caused by collision with floor during a vertical bounce
J: Counter
PH: Previous horizontal position of the ball
PLOT$: Character to draw
PV: Previous vertical position of the ball
ROW: Vertical position for border character
SH: Horizontal position
SV: Vertical position
VERT$: Vertical positioning string
VH: Horizontal velocity
VV: Vertical velocity
X, Y: Coordinates to draw at

Special Cases

Some modifications may cause the ball to go above the top of the box. This stops the program with an "ILLEGAL QUANTITY" error on line 10000. This means that the program attempted to print a character off the screen. As shown, the program will never generate this error.

Brief Description

Lines 40 to 80 draw the floor of the box.
Lines 90 to 140 draw the walls.
Lines 155 to 190 set the starting conditions.
Lines 195 to 390 move the ball through one time interval.
- Line 200 saves the ball's current position.
- Line 210 adds the ball's velocity to its position.
- Line 220 detects a bounce off the floor.
- Line 230 places the ball on the floor.
- Line 240 reverses the ball's vertical velocity.
- Lines 255 to 330 act similarly for wall bounces.
- Line 340 adds gravity.
- Lines 345 to 360 erase the old ball.
- Lines 365 to 380 draw the new ball.

Lines 9995 to 10010 print a character at a specific position.

Listing

```
5 REM "BOUNCE" ANIMATE A BALL
10 PRINT "{CLR}";
20 VERT$="{HOME}{24 DOWN}"
25 REM DRAW BORDER
30 PLOT$="{RVS}^{OFF}"
40 Y=0
50 FOR COL=0 TO 38
60 X=COL
70 GOSUB 10000
80 NEXT COL
90 FOR ROW=0 TO 24
100 X=0: Y=ROW
110 GOSUB 10000
120 X=38
130 GOSUB 10000
140 NEXT ROW
155 REM SET STARTING CONDITIONS
160 VH=1.5: VV=0
170 SH=2: SV=19
180 AV=-.2
190 DAMPH=.6: DAMPV=.9
195 REM SAVE CURRENT POSITION OF BALL
200 PH=SH: PV=SV
205 REM ADD VELOCITY TO POSITION
210 SH=SH+VH: SV=SV+VV
215 REM HANDLE A BOUNCE OFF FLOOR
220 IF INT(SV)>0 THEN 260
230 SV=1
240 VV=-VV*DAMPV
255 REM HANDLE A BOUNCE OFF WALL
260 IF INT(SH)>0 THEN 300
270 SH=1
```

```
280 VH=-VH*DAMPH
300 IF INT(SH)<38 THEN 340
310 SH=37
320 VH=-VH*DAMPH
335 REM APPLY GRAVITY TO VELOCITY
340 VV=VV+AV
345 REM ERASE BALL FROM OLD POSITION
350 PLOT$="^": X=PH: Y=PV
360 GOSUB 10000
365 REM DRAW BALL AT NEW POSITION
370 PLOT$="Q": X=SH: Y=SV
380 GOSUB 10000
390 GOTO 200

9995 REM PRINT CHARACTER AT X,Y
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN
```

Modifications

To make the ball flicker less (but move more slowly), add:

```
385 FOR K=1 TO 25: NEXT K
```

A simple change is to vary the starting conditions. For example, change the starting position (SH, SV) or speed (VH, VV) of the ball inlines 160 and 170.

Another change is to detect when the ball is about to bounce out of the box. You could then either restrict its motion or stop the program.

Since the program includes equations of ballistic motion, it could simulate a cannonball fired at a fortress. Simply remove the bounce code and provide the starting conditions (SH, SV, VH, and VV) for the cannonball. Draw the cannon and a fortress, and add statements that detect when the ball strikes the fortress. You could also modify the program to bounce a basketball into a net.

For a simple but interesting modification, remove lines 345 through 360 and see what happens. This change leaves all drawings on the screen, so you can see the ball's entire history. Figure 4–6 shows an example history.

The Commodore 64 can generate a wide range of interesting sounds to accompany pictures. The following modification produces a "springy" sound during each bounce:

```
145 REM SET UP SOUND CHIP
150 SID=54272: GOSUB 500
250 GOSUB 600: REM MAKE BOUNCING SOUND
290 GOSUB 600: REM MAKE BOUNCING SOUND
330 GOSUB 600: REM MAKE BOUNCING SOUND

495 REM INITIALIZE THE SOUND CHIP
500 FOR J=0 TO 24
510 POKE SID+J,0
```

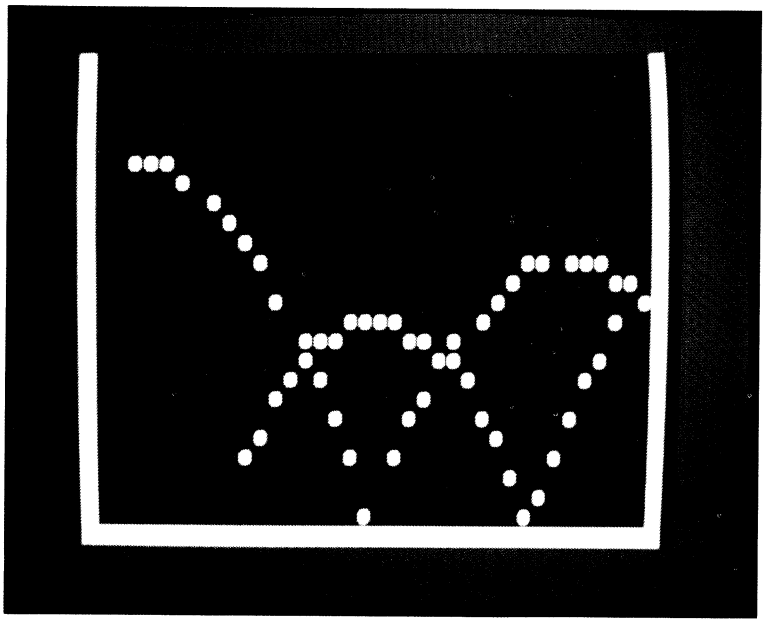


Figure 4–6 Follow the BOUNCEing ball

```
520 NEXT J
530 RETURN

595 REM MAKE A BOUNCING SOUND
600 POKE SID+5,6: POKE SID+6,6: REM SET SOUND ENVELOPE
605 REM SET VOLUME TO MAXIMUM
610 POKE SID+24,15
615 REM SET SOUND FREQUENCY
620 POKE SID+1,5
625 REM BEGIN SOUND
630 POKE SID+4,33
635 REM END SOUND
640 POKE SID+4,32
650 RETURN
```

Be sure you turn up the volume on your television set before running this modification. To turn the sound off after pressing RUN/STOP, enter:

```
SYS 0
```

(without a line number), or press RUN/STOP and RESTORE.

Notes

Simple (two-dimensional) motion simulation on a home computer usually involves three pairs of variables. One pair is the moving object's current

horizontal and vertical coordinates. Another pair holds onto the object's previous coordinates until the old picture can be erased. A third pair holds the horizontal and vertical velocity (speed and direction).

In BOUNCE, a positive horizontal velocity makes the ball move right, while a positive vertical velocity makes it move up. VH and VV determine how far the ball moves in one time step.

To produce more complex motion, such as acceleration, the program must change the object's velocity. Since gravity produces acceleration only in the vertical direction, BOUNCE needs only one acceleration variable (AV).

References

You can read more about motion simulation in a four-part article in the November 1977 through February 1978 issues of *BYTE* magazine.

The next program uses animation and simulation to produce a test of piloting skill.

Program Name: "LANDER"

Purpose

Simulates a spaceship attempting to land in a Martian volcano. Figure 4–7 shows a typical scene.

Techniques Demonstrated

Game design, simulation of spaceship motion, and use of the keyboard to control a game object.

Procedure

The ship enters the crater from the top left-hand corner. The landing pad is on the floor of the volcano. The player must slow the ship and land it gently on the pad. The S, E, and D keys allow the player to control the ship by applying thrust left, up, and right, respectively. A modification describes how to control the ship with a joystick. The edges of the screen represent the sides of the volcano.

To land the ship successfully, you must slow it down right away. Press E two or three times right away to decrease its vertical speed. Then reduce its horizontal speed by pressing S two or three times.

Once you have the ship under control, nudge it toward the landing pad by applying a little thrust right and left as needed. Remember to press E

Figure 4–7 Landing on Mars

occasionally to counteract gravity and keep the ship aloft. When the ship nears the landing pad, slow it down as much as possible and allow it to drop gently to the ground. Don't start congratulating yourself too soon; the ship may appear to have landed when it is still hovering just off the ground. The program will tell you when the ship has actually landed.

LANDER rates the pilot on the landing's accuracy and gentleness.

Variables

COL: Current column (horizontal) position of the ship
J: Counter
KEY$: Key pressed by the pilot
LINE: Current line (vertical) position of the ship
MESSAGE$: Comments on the landing
P: Column position of the landing pad
PLOT$: Characters to plot
RATING: Computer's numeric rating of the landing
SHIP$: Characters that draw the ship
TT, T: Counters for time delays
VERT$: Vertical positioning string

VL, VC: Number of lines and columns the ship moves during each time step

X, Y: Character-plotting coordinates

Brief Description

Line 10 ensures that the landing pad is in a different location each time the program runs.

Lines 25 to 60 set up the pilot rating messages.

Lines 70 to 140 draw the Martian surface with the landing pad starting in a random column.

Lines 145 to 160 set up the lander's starting position and speed.

Lines 165 to 370 simulate the lander's motion for one time period.

- Lines 170 to 180 display the lander at its current position.
- Lines 195 to 290 read the keyboard and apply thrust if the player presses the S (left), E (up), or D (right) key.
- Line 300 increases the vertical speed to account for gravity.
- Lines 305 to 320 calculate where the lander will be after a time step at its current speed.
- Lines 325 to 350 check whether the lander has crashed into a wall, gone into orbit (i.e., moved above the top of the screen), or reached the surface.
- Line 360 erases the lander from its old position.

Lines 375 to 410 rate the landing according to impact speed and distance from the center of the landing pad.

Lines 425 to 540 animate the lander crashing.

Lines 9995 to 10010 print characters starting at given X and Y coordinates.

Lines 14995 to 15070 read a key from the keyboard and pause.

Listing

```
5 REM "LANDER"  SIMULATE A MARTIAN LANDING
10 J=RND(0): REM RANDOMIZE
15 REM SET UP VERTICAL POSITIONING STRING
20 VERT$="{HOME}{24 DOWN}"
25 REM SET UP MESSAGE ARRAY
30 DIM MESSAGE$(2)
40 MESSAGE$(0)="AN^EXCELLENT^LANDING."
50 MESSAGE$(1)="A^BIT^ROUGH,^CAPTAIN."
60 MESSAGE$(2)="YOU^CAN^DO^BETTER...^TRY^AGAIN!"
70 PRINT "{CLR}{8}"
75 REM DRAW SURFACE AND LANDING PAD
80 PRINT "{HOME}{24 DOWN}{I}{O}{P}{@}{P}{O}{I}{P}{I}{O}{I}{P}
          {2 O}{5 I}{O}{2 @}{2 P}{O}{I}{P}{3 I}{P}{2 @}{2 P}
          {O}{I}{O}{P}";
```

```
90 SHIP$="{RVS}£{*}{DOWN}{2 LEFT}^^{DOWN}{2 LEFT}
         {OFF}NM{2 UP}{2 LEFT}"
95 REM MAKE SCREEN BLACK
100 POKE 53281,0
125 REM PUT LANDING PAD IN RANDOM POSITION
130 P=INT(RND(1)*36)
135 REM YELLOW LANDING PAD
140 PRINT "{HOME}{24 DOWN}" TAB(P) "{YEL}{RVS}^^^^{8}";
145 REM SET UP STARTING CONDITIONS FOR LANDER
150 LINE=22: COL=1
160 VL=-.125: VC=.45
165 REM SIMULATE LANDER MOTION
170 PLOT$=SHIP$
180 X=COL: Y=LINE: GOSUB 10000
195 REM GET COMMAND
200 GOSUB 15000
205 REM THRUST LEFT?
210 IF KEY$<>"S" THEN 240
220 PRINT "{DOWN}{RIGHT}{1}<{8}{UP}{2 LEFT}";
230 VC=VC-.15
235 REM THRUST UP?
240 IF KEY$<>"E" THEN 270
250 PRINT "{2 DOWN}{1}{RVS}£{*}{OFF}{8}{2 UP}{2 LEFT}";
260 VL=VL+.15
265 REM THRUST RIGHT?
270 IF KEY$<>"D" THEN 300
280 PRINT "{DOWN}{1}>{8}{UP}{LEFT}";
290 VC=VC+.15
295 REM APPLY FORCE OF GRAVITY
300 VL=VL-.06
305 REM APPLY VELOCITY
310 LINE=LINE+VL
320 COL=COL+VC
325 REM CRASH IF SHIP HITS A WALL
330 IF COL<0 OR COL>38 THEN 450
335 REM RATE LANDING IF SHIP ON SURFACE
340 IF LINE<2 THEN 380
345 REM OFF TOP OF SCREEN?  LANDER BACK IN ORBIT
350 IF LINE>=24 THEN 550
355 REM ERASE OLD SHIP
360 PRINT "^^{DOWN}{2 LEFT}^^{DOWN}{2 LEFT}^^"
370 GOTO 170
375 REM RATE LANDING
380 RATING=ABS(VL*2)+ABS(VC)+ABS(P-COL+1)
390 IF RATING>=3 THEN 450
395 REM PRINT MESSAGE
400 PLOT$=MESSAGE$(RATING): X=5: Y=11: GOSUB 10000
420 END

425 REM CRASH LANDER
450 FOR T=1 TO 15
460 PLOT$="{RED}"+SHIP$: GOSUB 10000
470 PLOT$="{YEL}"+SHIP$: GOSUB 10000
480 PLOT$="{BLU}"+SHIP$: GOSUB 10000
490 PLOT$="{RED}"+SHIP$: GOSUB 10000
500 NEXT T
510 PLOT$="{1}*CRASH!*^^TRY^AGAIN{8}"
520 X=5: Y=11: GOSUB 10000
540 END

550 PLOT$="{1}IN^ORBIT.^^TRY^AGAIN{8}"
560 X=5: Y=11: GOSUB 10000
580 END

9995 REM PRINT STRING STARTING AT X,Y
```

```
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN

14995 REM TAKE COMMAND FROM KEYBOARD
15000 GET KEY$
15060 FOR TT=1 TO 100: NEXT TT
15070 RETURN
```

Modifications

To add sound effects, enter the following program lines:

```
65 GOSUB 600: REM INITIALIZE SOUND REGISTERS
330 IF COL<0 OR COL>38 THEN 430
390 IF RATING>=3 THEN 430
410 POKE SID+24,0
430 POKE SID+24,15: POKE SID+1,2: POKE SID+4,129
530 POKE SID+24,0
570 POKE SID+24,0

595 REM INITIALIZE SOUND REGISTERS
600 SID=54272
610 FOR J=0 TO 24
620 POKE SID+J,0
630 NEXT J
635 REM SET MODERATE VOLUME
640 POKE SID+24,7
645 REM SET SOUND ENVELOPE
650 POKE SID+5,16: POKE SID+6,240+7
660 RETURN


15005 REM SET SOUND TO MODERATE VOLUME
15010 POKE SID+24,7
15020 IF KEY$="E" THEN POKE SID+1,3
15030 IF KEY$="S" THEN POKE SID+1,4
15040 IF KEY$="D" THEN POKE SID+1,5
15050 IF KEY$<>"" THEN POKE SID+4,129: REM BEGIN SOUND
15060 FOR TT=1 TO 100: NEXT TT
15070 POKE SID+4,128: REM END SOUND
15080 RETURN
```

Games that involve piloting are natural candidates for joystick control. Insert the following lines to add this capability to LANDER:

```
15006 JOY=PEEK(56320)
15007 IF (JOY AND 1)=0 THEN KEY$="E"
15008 IF (JOY AND 4)=0 THEN KEY$="S"
15009 IF (JOY AND 8)=0 THEN KEY$="D"
```

This modification does not interfere with using the keyboard to control the lander. See the Notes for more details about the joystick.

You can change many other things in this program. Try changing the starting position and speed in lines 150 and 160. If you slow the lander down and start it near the center of the screen (LINE = 10, COL = 19), it becomes

much easier to land. You can give the lander more powerful engines by increasing the thrust (say from 0.15 to 0.2) in lines 230, 260, and 290.

An interesting change is to track fuel usage. You can do this by first creating a variable FUEL, which you set to some starting value (say, 50). Then in lines 225, 255, and 285 add the statement FUEL = FUEL − 1. At line 355 add an IF statement that tests whether the ship has run out of fuel. If it has, the program reports "OUT OF FUEL" and crashes the lander. Still another option is to have an obstacle (such as a large boulder) the pilot must avoid during the landing. You can add the obstacle by PRINTing it on the surface, near the landing pad. You must then add comparisons to the IF statement (line 330) to determine whether the lander has hit the obstacle.

You may also want to make the lander crumple into pieces if it crashes. You could even have fragments fly up into space.

Notes

The Commodore 64 can accommodate two joysticks. They may be plugged into sockets on the right side of the computer, marked "Control Port 1" and "Control Port 2." Each joystick unit contains five switches. Four are triggered by moving the joystick handle, while the fifth is triggered by pressing the fire button. The computer can determine the switch positions (open or closed) of each joystick by PEEKing at special memory locations. Locations 56321 (Control Port 1) and 56320 (Control Port 2) contain the positions of the joystick switches (Up, Down, Left, and Right) and the fire button. Pressing a fire button or tilting a joystick handle closes the corresponding switch. Table 4–2 lists the formulas that a program can use to determine whether a switch is open or closed. If the formula's value is 0, the switch is open; otherwise, the switch is closed.

The scoring computation in line 380 uses the ABS function. ABS (absolute value) is the magnitude or size of a number, regardless of whether it is positive or negative. For example, ABS(28) = ABS(−28) = 28, since 28 and −28 differ only in sign. In the scoring computation, ABS gives equal weight to a positive or negative velocity and to a landing left or right of the center of the launch pad.

TABLE 4-2 FORMULAS FOR DETERMINING STATES OF JOYSTICK SWITCHES, WHERE PORT=56321 (PORT 1) OR 56320 (PORT 2)

Switch	Formula for detecting open or closed switch
Up	(PEEK(PORT) AND 1)
Down	(PEEK(PORT) AND 2)
Left	(PEEK(PORT) AND 4)
Right	(PEEK(PORT) AND 8)
Fire	(PEEK(PORT) AND 16)

Note that the exact way you calculate the score is arbitrary. You should consider how far the ship lands from the center of the pad (COL − (P + 1)) and how fast it is going on impact (VL vertically, VC horizontally). However, you can combine these factors in different ways according to what you want to stress. For example, you might regard any landing off the pad (that is, in columns other than P or P + 1) as an automatic failure.

Good game design depends on several principles. First, a game must not be too easy. If it is, it will quickly become boring. On the other hand, a game must not be so difficult that players become frustrated. In fact, the ideal game starts out easy and becomes more difficult as you play it. This is true of most arcade games. You could modify LANDER to automatically increase its difficulty each time it is played. You could, for example, increase the starting speed, decrease the engine thrust, increase gravity, or make the scoring stricter.

Another important rule is that a game should not be predictable. If it is, players will soon master it and become bored. One example of unpredictability in LANDER is the random placement of the landing pad. Sometimes the program places the pad near the center of the crater, where there is lots of room to maneuver. At other times the landing pad is near the wall, and you must pilot the ship very carefully. Another way to make this game more interesting would be to randomize the starting position and speed.

References

Pages 343 through 345 of the *Commodore 64 Programmer's Reference Guide* discuss the joystick further.

To learn more about game design, see the excellent article by Chris Crawford in the December 1982 issue of *BYTE* magazine. Some background on motion simulation is in the November 1977 through February 1978 issues of *BYTE*.

ANIMATION BY SCROLLING

Sometimes we can take advantage of the computer's built-in operations to produce faster animation. For example, we can use scrolling to move a picture up the screen. All the program must do is draw the picture near the bottom of the screen and then PRINT lines underneath it. Note that the program does not have to erase and redraw the picture or move the cursor; the computer handles all this on its own.

Program LAUNCH demonstrates the use of scrolling to animate a launch of the space shuttle. The speed is high because the computer is handling the details rather than the program. Unfortunately, we can use scrolling

only to move entire scenes upward; we can't easily move anything in other directions or move objects against a stationary background.

Program Name: "LAUNCH"

Purpose

Draws the space shuttle and launches it. Figure 4–8 and Plate 6 show a typical scene from the launch.

Techniques Demonstrated

Animation with scrolling.

Procedure

RUN the program!

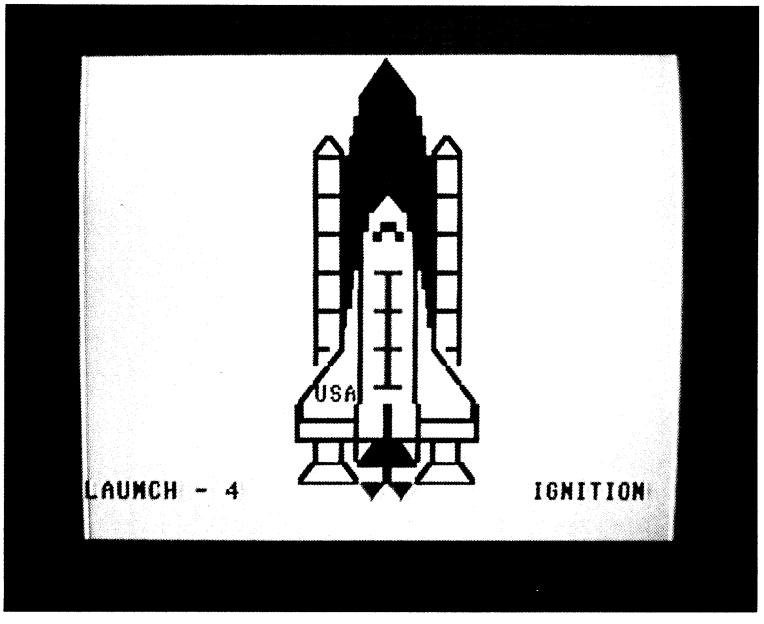


Figure 4–8 Space shuttle during LAUNCH

Variables

J: Column for a smoke character
K: Counter
SM: Beginning column for smoke
T: Seconds counter
TT: Pause counter

Brief Description

Lines 15 to 250 draw the space shuttle.
Lines 290 to 320 produce a countdown from 9 to 4.
Line 340 indicates main engine ignition by displaying the blue exhaust shock diamonds (along with the word "FIRE").
Lines 345 to 380 complete the countdown.
Lines 415 to 490 draw the smoke and scroll the shuttle upward.
Lines 450 to 470 draw one line of smoke, pushing the rocket up the screen.
Line 480 makes the smoke spread out until it nearly fills the screen.
Lines 995 to 1020 pause for one second.

Listing

```
5 REM "LAUNCH"  LAUNCH SPACE SHUTTLE
20 PRINT "{CLR}";
25 REM MAKE SCREEN LIGHT GREY
30 POKE 53281,15
35 REM DRAW SPACE SHUTTLE
40 PRINT TAB(19) "{2}{RVS}£{*}"
50 PRINT TAB(18) "{RVS}£^^{*}"
60 PRINT TAB(18) "{RVS}^^^^"
70 PRINT TAB(17) "{L}{RVS}^^^^{OFF}{J}"
80 PRINT TAB(15) "{BLK}NM{2}{RVS}{J}^^^^{L}{OFF}{BLK}NM"
90 PRINT TAB(15) "OP{2}{RVS}{6 ^}{OFF}{BLK}OP"
100 PRINT TAB(15) "{H}{N}{2}{RVS}{6 ^}{OFF}{BLK}{H}{N}"
110 PRINT TAB(15) "OP{2}{RVS}^^{OFF}£{*}{RVS}^^{OFF}{BLK}OP"
120 PRINT TAB(15) "{H}{N}{2}{RVS}^{N}{BLK}{OFF}
          {D}{F}{2}{RVS}{H}^{OFF}{BLK}{H}{N}"
130 PRINT TAB(15) "OP{2}{RVS}^{OFF}{J}{BLK}
          {V}{C}{2}{L}{RVS}^{OFF}{BLK}OP"
140 PRINT TAB(15) "{H}{N}{2}{RVS}^{BLK}{OFF}
          {H}^^{N}{2}{RVS}^{OFF}{BLK}{H}{N}"
150 PRINT TAB(15) "OP{2}{RVS}{N}{BLK}{OFF}
          {H}{5}PO{BLK}{N}{2}{RVS}{H}{OFF}{BLK}OP"
160 PRINT TAB(15) "{H}{N}{2}{RVS}{L}{BLK}{OFF}
          {H}{5}{N}{H}{BLK}{N}{2}{RVS}{K}{OFF}{BLK}{H}{N}"
170 PRINT TAB(15) "OP{2}{H}{BLK}{OFF}{H}{5}PO{BLK}
          {N}{2}{L}{OFF}{BLK}OP"
180 PRINT TAB(15) "{H}{N}^{H}{5}{N}{H}{BLK}{N}^{H}{N}"
```

```
190 PRINT TAB(15) "ON^{H}{5}PO{BLK}{N}^MP"
200 PRINT TAB(15) "N^^{H}{5}{N}{H}{BLK}{N}^^M"
210 PRINT TAB(14) "N{RED}USA{BLK}{H}{5}{2 Y}{BLK}{N}^^^M"
220 PRINT TAB(13) "{N}L{2 P}@^{N}{H}^L{2 P}@{H}"
230 PRINT TAB(13) "{N}{3 P}@{P}@L{P}L{3 P}{H}"
240 PRINT TAB(15) "{H}{2 N}{RVS}£^^{*}{OFF}{2 H}{N}"
250 PRINT TAB(14) "N{2 T}M{Y}PO{Y}N{2 T}M"
260 PRINT TAB(14) "{4 T}^^^^{4 T}"
265 REM BEGIN COUNTDOWN
280 PRINT "{UP}{BLU}LAUNCH^-^{RED}9"
290 FOR T=9 TO 4 STEP -1
300 GOSUB 1000: REM WAIT ONE SECOND
310 PRINT "{UP}" TAB(8);T
320 NEXT T
335 REM DRAW MAIN ENGINE SHOCK DIAMONDS
340 PRINT "{UP}" TAB(18) "{7}{*}£{*}£{RED}" TAB(30) "IGNITION"
345 REM RESUME COUNTDOWN
350 FOR T=3 TO 0 STEP -1
360 GOSUB 1000: REM WAIT ONE SECOND
370 PRINT "{UP}" TAB(8);T
380 NEXT T
385 REM BEGIN LIFTOFF SEQUENCE BY ERASING LAUNCH STATUS INFORMATION
390 PRINT "{UP}{10 ^}" TAB(30) "{8 ^}"
405 REM DRAW FIRST LINE OF SMOKE
410 PRINT TAB(12) "{1}{RVS}£{6 ^}{*}£{6 ^}{*}"
415 REM SET BEGINNING COLUMN FOR SMOKE
420 SM=11
430 FOR K=0 TO 27
445 REM PRODUCE BILLOWING SMOKE-DIAGONAL LEFT SIDE
450 PRINT TAB(SM) "{RVS}£";
455 REM SOLID SMOKE IN CENTER
460 FOR J=SM+1 TO 38-SM STEP 2: PRINT "^^"; : NEXT J
465 REM DIAGONAL RIGHT SIDE
470 PRINT "{*}"
475 REM EXPAND SMOKE OUTWARD AS SHUTTLE RISES
480 IF SM>1 THEN SM=SM-1
490 NEXT K
510 END

995 REM WAIT ONE SECOND
1000 FOR K=1 TO 700: NEXT K
1010 RETURN
```

Modifications

You can slow the launch down by inserting

```
485 FOR L=1 TO 100: NEXT L
```

Entering the following lines will add sound effects to the launch sequence:

```
10 GOSUB 600: REM INITIALIZE SOUND REGISTERS
325 REM BEGIN ROCKET THRUST SOUNDS
330 GOSUB 700
395 REM INCREASE THRUST SOUND TO MAXIMUM
400 POKE SID+24,32+15
425 REM VARY SOUND FREQUENCY
430 FOR FF=0 TO 255 STEP 10
```

```
440 GOSUB 800
490 NEXT FF
500 GOSUB 900

595 REM INITIALIZE SOUND CHIP
600 SID=54272
610 FOR J=0 TO 24
620 POKE SID+J,0
630 NEXT J
640 RETURN

695 REM BEGIN LAUNCH THUNDER
700 POKE SID+5,192: POKE SID+6,240+13: REM SET ENVELOPE
705 REM SET FILTER PARAMETERS
710 POKE SID+24,32+7: REM BANDPASS, MEDIUM VOLUME
720 POKE SID+22,0:    REM FREQUENCY
730 POKE SID+23,161:  REM RESONANCE, VOICE
735 REM SET FREQUENCY
740 POKE SID+1,3: POKE SID+0,0
745 REM BEGIN SOUND
750 POKE SID+4,129
755 REM SET FREQUENCY VARIABLE FF
760 FF=0
770 RETURN


795 REM VARY FILTER FREQUENCY
800 POKE SID+22,FF
805 REM PAUSE
810 FOR T=0 TO 500-FF: NEXT T
820 RETURN

895 REM FADE OUT THUNDER
900 POKE SID+4,128
910 RETURN
```

This program can launch any kind of spacecraft. For instance, try launching the shuttle's European counterpart, Ariane (see Figure 4–9). Even a flying saucer will work (if you believe in that sort of thing). Rumor has it that flying saucers do not trail fire and smoke like conventional spacecrafts. To produce a smokeless liftoff, delete lines 460 through 480 and insert:

```
450 PRINT
```

The subroutine starting in line 1000 is a simple way to produce a delay of about one second. A more precise approach is to use the computer's internal counter TI, which it updates every one-sixtieth of a second (every one fiftieth of a second outside North America). Note that the computer keeps TI in addition to its clock string TI$, which we discussed earlier.

To use TI to generate a one-second delay, we simply obtain its current value and then wait for the value to increase by 60. This is like waiting for a one-minute egg to boil by marking down the current time on a clock and

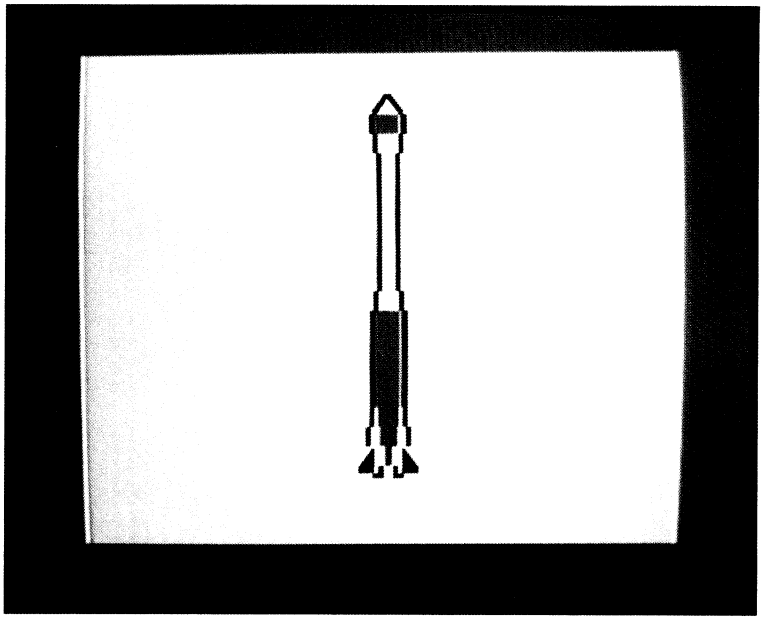


Figure 4–9 Ariane

waiting until the clock is exactly one minute further along. The required changes are:

```
265  REM GET INITIAL CLOCK COUNT
270  TBEGIN=TI
995  REM WAIT ONE SECOND USING CLOCK COUNT
1000  IF (TI-TBEGIN)<60 THEN 1000
1005  REM UPDATE BEGINNING TIME FOR NEXT COUNT
1010  TBEGIN=TI
1020  RETURN
```

If you are outside North America, use 50 instead of 60 in line 1000.

References

The *Space Shuttle Operator's Manual* (Ballantine Books, 1982) contains excellent drawings and descriptions of the shuttle.

5

COMPUTER ART

ARTIST'S TOOLBOX

Arts and crafts fascinate most people, regardless of whether they have any talent or training in them. Computers not only provide a way for people to draw pictures or patterns electronically, but they can even help amateur or professional artists. The computer can perform routine tasks ranging from the generation of lines and geometrical forms to the development of a variety of random or planned patterns. Of course, today's computers cannot substitute for the skill, experience, and creativity of a trained artist. But they can be useful tools for artists, much as they are for businesspeople, engineers, scientists, teachers, doctors, and lawyers.

What exactly can an artist do with a computer? He or she can, for example:

1. Use it as an electronic sketchpad for making preliminary drawings. The artist can load a sketch, change its shape or color, contract or expand it, erase it, or move it at electronic speeds rather than at the speed of a brush or pencil.

2. Keep files of earlier works on disk or tape. These can serve as back-up copies, examples for a portfolio, or starting points for new tasks.

3. Let the computer handle routine jobs such as drawing lines and geometrical forms, shading areas, expanding or contracting figures, mixing or changing colors, adjusting or checking dimensions, determining relative sizes, and copying entire drawings or parts of drawings.

4. Use the computer's media and communications abilities. The artist

can transfer pictures or sketches on disk, tape, or even by telephone to other artists or to production facilities.

5. Keep a library of standard forms, trademarks, figures, scenes, characters, or components. These could include everything from diamond or oval shapes through company logos, cartoon figures, or landscape backdrops.

The two programs in this chapter, PIC–EDIT and SKETCH, are typical of the tools that make a computer into an artist's assistant. With a few simple commands, the user can draw, change colors, specify geometrical forms, and load, save, or edit pictures. Just as word processors help writers, picture editors give artists a forgiving medium with which to work, the ability to save completed products for later use, and a rapid, accurate way of performing humdrum but essential jobs. The computer thus provides the artist with more time to experiment and more freedom to use his or her imagination.

DEVELOPING AN ARTIST'S TOOLBOX

To create the artist's tools, we need the following:

1. A simple way for the user to specify what the computer should do. Note that we cannot use the regular keys for editing commands, since the user needs them to enter characters, move the cursor, change the printing color, turn the reverse mode on or off, and clear the screen. One approach is to take advantage of the *function keys* at the far right side of the keyboard. The computer does not normally do anything with these keys, so they are available for program use. They can, for example, substitute for entire sequences of keyboard inputs (that is, act as shorthand keys) or serve as command inputs when the other keys are already in use. Editors of all types, whether picture editors or word processors, generally need all the regular keys for data entry.

2. Routines that perform building-block tasks such as drawing lines, circles, and other geometrical forms. We developed these in Chapter 3.

3. A way to determine the printing colors of spaces on the screen. This is essential for loading and saving color pictures. The key here is the computer's color memory. It is organized like the screen memory we mentioned earlier, except that it contains color codes rather than character codes.

4. Methods for transferring data to or from disk or tape. We already know how to load and save programs (see Chapter 1); now we must extend those procedures to data such as screen and color codes.

5. A way to determine where the cursor is. You may say, "I know where the cursor is. I can see it." Yes, but remember that the Commodore 64 does not display the cursor when a program is running. Our artist's toolbox program must provide a cursor for the artist to use without destroying

part of the picture. Thus the program must save the character and color at the cursor position, display its cursor, and then restore the original character and color when the cursor moves. Note that the 64 must follow a similar procedure to display its cursor.

You may remember that we already provided cursors in program TTT (tic-tac-toe) and in the modifications to MAZE. In those cases, however, we usually restricted the cursor to blank areas. In fact, one problem we solved in the modifications to TTT was that of the cursor leaving a space in the middle of an X drawing. The change replaced the cursor with the original character before moving it.

Color Memory

Like screen memory, color memory consists of one address for each character on the screen. However, here the contents are codes for the colors as listed in Table 5–1. The codes for the colors marked on the number keys are all 1 less than the corresponding numbers; the codes for the secondary colors (the ones you obtain by pressing the Commodore key and a number key simultaneously) are all 8 larger than the corresponding numbers.

Color memory begins at address 55296. We will refer to this address as CMEM for convenience; it contains the code for the printing color in the top left-hand character space. The rest of color memory is arranged just

TABLE 5-1 COLOR CODES FOR COLOR MEMORY

Code	Color
0	Black
1	White
2	Red
3	Cyan
4	Purple
5	Green
6	Blue
7	Yellow
8	Orange
9	Brown
10	Light Red
11	Dark Gray
12	Medium Gray
13	Light Green
14	Light Blue
15	Light Gray

like screen memory, that is, by rows starting in the top left-hand corner and moving first right and then down. For example:

CMEM + 1 contains the color code for the character space just right of the top left-hand corner.

CMEM + 39 contains the color code for the character space in the top right-hand corner.

CMEM + 40 contains the color code for the character space just below the top left-hand corner.

To demonstrate the use of color memory, let us write a simple program that draws a horizontal bar. We will make it half yellow and half red. It will appear near the middle of the screen against a black background.

```
10   PRINT "{CLR}";
15   REM MAKE BACKGROUND BLACK
20   POKE 53281,0
25   REM SET START OF SCREEN, COLOR MEMORY
30   SMEM=1024
40   CMEM=55296
45   REM GO DOWN 12 LINES
50   SB=SMEM+12*40
60   CB=CMEM+12*40
65   REM GET SCREEN CODE FROM TABLE 4-1, COLOR CODES FROM TABLE 5-1
70   SQ=160: YEL=7: RED=2
75   REM DRAW 20 SOLID YELLOW SQUARES
80   FOR C=0 TO 19
85   REM PUT SOLID SQUARE IN SCREEN MEMORY
90   POKE SB+C,SQ
95   REM TURN SQUARE YELLOW
100  POKE CB+C,YEL
110  NEXT C
115  REM DRAW 20 SOLID RED SQUARES
120  FOR C=20 TO 39
125  REM PUT SOLID SQUARE IN SCREEN MEMORY
130  POKE SB+C,SQ
135  REM TURN SQUARE RED
140  POKE CB+C,RED
150  NEXT C
160  GET K$: IF K$="" THEN 160
170  POKE 53281,6: PRINT "{7}{CLR}";
180  END
```

You must press a key (the space bar will do) to exit.

To obtain a color code CC from address CADDR in color memory, we need the statement:

```
CC=PEEK(CADDR) AND 15
```

We will use this to save color codes on disk or tape when saving pictures, and also to save the current color while displaying the artist's cursor.

Finding the Cursor

The following magic formula determines the location in screen memory that contains the character under the cursor:

```
SPTR = PEEK(209) + PEEK(210)*256 + PEEK(211)
```

The corresponding location in color memory is:

```
CPTR = SPTR + CMEM - SMEM
```

where CMEM and SMEM are, as we mentioned, the starting addresses of color and screen memory, respectively.

The next example program prints a multicolored version of the word RAINBOW near the center of the screen. It then moves the cursor to the middle of that word, saves the current character and color, displays a flashing asterisk while waiting for a key entry, restores the original character and color, and prints the key. This allows the user to enter text by just typing, and move the cursor or change printing colors without affecting the text.

```
10   POKE 53281,1
15   REM MOVE NEAR CENTER OF SCREEN
20   PRINT "{CLR}{12 DOWN}";TAB(17);
25   REM PRINT A MULTICOLORED WORD "RAINBOW"
30   PRINT "{BLK}R{RED}A{CYN}I{PUR}N{GRN}B{1}O{YEL}W{BLU}";
35   REM MOVE CURSOR TO MIDDLE OF WORD
40   PRINT "{4 LEFT}";
45   REM SET START OF SCREEN, COLOR MEMORY
50   SMEM=1024
60   CMEM=55296
65   REM DETERMINE WHERE CURSOR IS
70   SPTR=PEEK(209)+PEEK(210)*256+PEEK(211)
75   REM SAVE CHARACTER UNDER CURSOR
80   CHAR=PEEK(SPTR)
85   REM SAVE COLOR OF CHARACTER UNDER CURSOR
90   CPTR=CMEM+SPTR-SMEM
100  CC=PEEK(CPTR) AND 15
105  REM DISPLAY FLASHING ASTERISK
110  PRINT "*{LEFT}";
120  FOR K=1 TO 100: NEXT K
125  REM ERASE ASTERISK BY OVERPRINTING WITH SPACE
130  PRINT "^{LEFT}";
140  FOR K=1 TO 100: NEXT K
145  REM LOOK FOR KEY
150  GET K$: IF K$="" THEN 110
155  REM RESTORE CHARACTER UNDER CURSOR
160  POKE SPTR,CHAR
170  POKE CPTR,CC
175  REM THEN RESPOND TO KEY PRESSED
180  PRINT K$;
185  REM NEXT DETERMINE WHERE CURSOR IS NOW AND REPEAT
190  GOTO 70
```

The flashing blue asterisk will originally replace the purple N in RAINBOW. Press {RIGHT}, and you will see the flashing asterisk move right and the purple N reappear. Now the flashing asterisk is covering the green B. Press {DOWN}, and the flashing asterisk will move down and the green B will reappear. Try changing the word RAINBOW to RAINING; the ING will appear in the current printing color. If you want to produce the effect of the asterisk flashing on top of a character, replace line 130 with:

```
125 REM OVERPRINT ASTERISK WITH ORIGINAL CHARACTER AND COLOR
130 POKE SPTR,CHAR: POKE CPTR,CC
```

Now lines 160 and 170 are unnecessary, since line 130 restores the original character and color.

You may find this program quite confusing, since it seems as if it is stopping the computer from doing what it would do normally. That is, when you press a key, the computer normally displays the corresponding character or does something such as change the printing color. It then moves the cursor, if necessary.

So what is our program doing? Its main job is to display a cursor. Remember that the computer does not display its cursor while a user program (such as an artist's toolbox) is running.

Our program also illustrates how the computer's built-in editor program works. This editor normally obtains the character at the cursor position, flashes the cursor (by alternating the character and its reverse), reads the keyboard, restores the original character, and responds to the key entry. There is always a program interposed between the keyboard and the computer. Our program is just taking the place of the Commodore 64's built-in program.

Let us look at a few examples and see what happens. Say you have the flashing asterisk over the N in RAINBOW. What happens when you press {RIGHT}? In line 150, K$ = "{RIGHT}" so the computer continues rather than branching back to line 110. Lines 160 and 170 then put the purple N back on the screen. Line 180 is equivalent to PRINT "{RIGHT}";, so it moves the cursor one column right. Lines 70 through 100 then save the character and color under the cursor (that is, B and "{GRN}"), and lines 110 through 150 display the flashing asterisk and wait for another key entry.

What if you now press I? In line 150, K$ = "I" so the computer continues rather than branching back to line 110. Lines 160 and 170 then put the green B back on the screen. Line 180 is equivalent to PRINT "I";, so it overprints the green B with an I in the current printing color and moves the cursor one column right. Lines 70 through 150 then continue as before. Note that here we didn't really need lines 160 and 170, since the I replaced the B anyway. However, it is simpler just to have the computer do those lines as a precaution than to determine if they are actually necessary.

Function Keys

The function keys at the far right of the keyboard are convenient ways to enter commands. This is because they are off by themselves, easy to find, and not normally used by the computer. Note that f1, f3, f5, and f7 (the designations on the tops of the keys) are lowercase, while f2, f4, f6, and f8 (the designations on the front of the keys) are uppercase.

The obvious question is, "How do you tell the computer to look for a function key?" The answer is that the computer assigns them codes from ASCII as shown in Table 5–2. To look for function key f1, for example, we must have the program look for code 133. To do this, we need CHR$, which converts ASCII characters into their equivalent strings.

The following program is a simple example of recognizing a function key. It waits for you to press f1, and then flashes the message FUNCTION KEY F1 near the center of the screen.

```
10  PRINT "{CLR}";
15  REM WAIT FOR F1 KEY
20  GET K$: IF K$<>CHR$(133) THEN 20
25  REM MOVE NEAR CENTER OF SCREEN
30  PRINT "{12 DOWN}";
35  REM FLASH MESSAGE BY PRINTING IT AND OVERPRINTING IT WITH SPACES
40  PRINT TAB(12); "{UP}FUNCTION^KEY^F1"
50  FOR K=1 TO 100: NEXT K
60  PRINT TAB(12); "{UP}{RVS}FUNCTION^KEY^F1"
70  FOR K=1 TO 100: NEXT K
80  GOTO 40
```

Line 20 waits for you to press the f1 key—that is, the key equivalent to code 133 in ASCII. Note the use of CHR$ to produce a string the computer can

TABLE 5-2 ASCII EQUIVALENTS FOR FUNCTION KEYS

Function Key	ASCII Equivalent
f1	133
f2	137
f3	134
f4	138
f5	135
f6	139
f7	136
f8	140

compare with K$. You can change this program to respond to other function keys, but remember to press SHIFT to get f2, f4, f6, or f8.

File Management

To save data on tape or disk (or load data from either), you must remember to:

1. Open a file, just as if you were preparing to move information to or from a drawer in a file cabinet.

2. Give the file a name. This serves the same purpose as the identifying label one attaches to a cardboard file folder. The name should start with a letter and be short (say, less than 10 characters), meaningful, easy to type, and easy to distinguish from other names. The Commodore 64 allows file names to contain letters, numbers, spaces, and certain symbols such as −, +, and *.

One way to specify the type of information a file contains is to attach a prefix or suffix to its name; we might, for example, attach PIC to indicate a picture, TXT to indicate text, or FIN to indicate financial records. You can then quickly see whether a file is the right type for a particular application.

3. Close the file, again just as if it were in a standard office file cabinet.

A program can open a file with a statement like:

```
OPEN 2,1,0,"FILENAME" (tape input)
OPEN 2,1,1,"FILENAME" (tape output)
```

or:

```
OPEN 2,8,2,"@0:FILENAME,S,R" (disk input)
OPEN 2,8,2,"@0:FILENAME,S,W" (disk output)
```

The 2 immediately after OPEN is a number that identifies the file. That is, subsequent statements can just refer to it as 2 or #2. The next number (1 for the Datassette, 8 for the disk) tells the computer which device to use. The other parts of the command are specific to the particular device and operation.

After opening the file, a program can read data (values for A, B, and C) from it with:

```
INPUT#2, A, B, C
```

The 2 after INPUT# is the identifying number from the OPEN statement. Note that you cannot put a space after INPUT or between # and 2. Similarly, a program can store A, B, and C in the file with:

```
PRINT#2, A, B, C
```

As with INPUT, #2 must come immediately after PRINT; no spaces are allowed.

When the program is finished with the file, it must close it using:

```
CLOSE 2
```

Here again, the 2 is the identifying number from the OPEN statement.

SIMPLE PICTURE EDITOR

Program PIC-EDIT serves as a first example of an artist's toolbox. It produces its own cursor and uses function keys for EXIT, LOAD, and SAVE commands.

Program Name: "PIC-EDIT"

Purpose

Lets the user load, edit, and save pictures. The user can draw pictures and save them on tape or disk; he or she can also load old pictures from tape or disk and change them. Figures 5–1, 5–2 (pages 149 and 155) and Plate 7 show typical pictures drawn with PIC-EDIT.

Techniques Demonstrated

Locating and using color memory.
Using function keys to enter commands.
Determining the cursor's position.
Displaying a cursor without affecting the character at its position.
Moving a cursor with the joystick.
Loading a picture from a disk or tape file.
Saving a picture in a disk or tape file.

Procedure

Load the program from tape or disk, or from the keyboard. You may now use the keyboard to create a picture from graphics or ordinary typewriter characters. Use the CRSR keys ({UP}, {DOWN}, {LEFT}, {RIGHT}, and {HOME}) to move the cursor, which is a large cross. You can also move the cursor by tilting the joystick. To save a picture, press the f1 key. You must enter the name of a tape or disk file (limit it to 12 characters); when using the Datassette, be sure to press the PLAY and RECORD buttons *before*

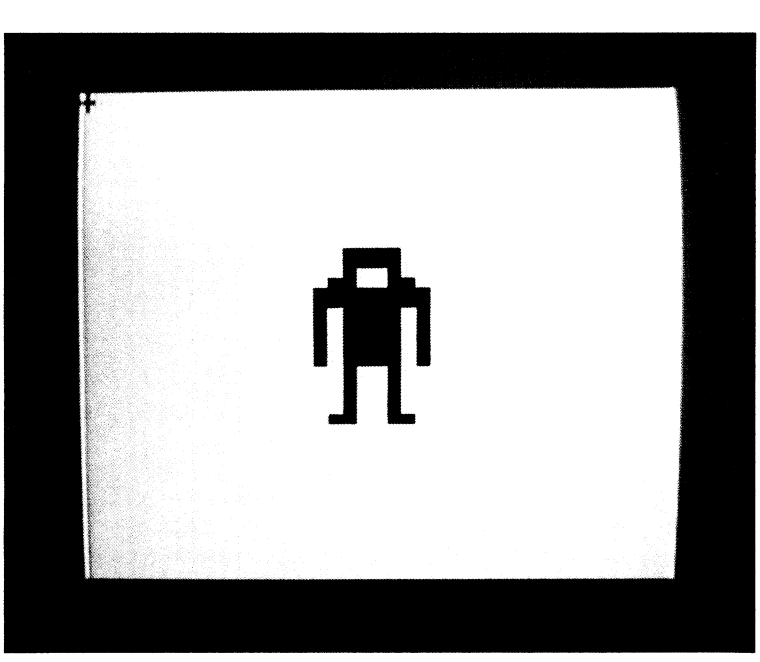


Figure 5–1 Lunar explorer drawn with PIC-EDIT

entering the file name. Similarly, by pressing f2 (shifted f1 key), you can load a picture you saved previously. Pressing f8 (shifted f7 key) ends the program.

Variables

CC: Color code for space at cursor position

CHAR: Screen code of character at cursor position

CMEM: Starting address of color memory

CPTR: Color memory address corresponding to current cursor position

F1$, F2$, F8$: Strings containing the character equivalents of the function keys f1, f2, and f8, respectively

J: Counter

K$: Key pressed by the user

PNAME$: Name of file used to save or load the picture. The program automatically puts "PIC—" ahead of this name to identify the file as a picture

SC: Screen code value

SMEM: Starting address of screen memory

SPTR: Screen memory address corresponding to current cursor position

Special Cases

We do not recommend using a Datassette with this program. It takes several minutes and a lot of tape to load or save a picture this way. Even transferring pictures to and from disk is a slow process.

Do not place the cursor in the bottom right-hand corner of the screen. Also, do not press {DOWN} when the cursor is on the bottom line. Either action will scroll the entire screen up a line.

Brief Description

Line 10 clears the screen and centers the cursor.

Line 13 makes the screen color light gray.

Line 20 sets the starting address of screen memory.

Line 30 sets the starting address of color memory.

Lines 40 to 60 define the character equivalents of the function keys.

Lines 70 and 90 calculate the screen and color memory addresses corresponding to the current cursor position.

Lines 80 and 100 save the screen and color codes at the cursor location, so that they can be restored after the cursor is displayed.

Line 110 to 112 blink the program's cursor.

Line 120 reads the keyboard.

Lines 130 to 150 respond to commands entered via function keys.

Line 160 responds to commands entered from the joystick.

Line 170 loops back to read the keyboard again if the user has not pressed a key or tilted the joystick's handle.

Line 180 makes the computer respond to the user's keyboard or joystick entry.

Lines 7995 to 8090 load a picture from a disk file.

Lines 8010 to 8020 ask for the file name and prepare the file for use.

Lines 8025 to 8070 read screen codes and color codes from the file and POKE them into screen and color memory.

Line 8080 closes the file.

Lines 8995 to 9080 save the current picture in a disk file.

Lines 14995 to 15060 determine the joystick's position and convert it into cursor moves.

Listing

```
5 REM "PIC-EDIT" DRAW AND SAVE PICTURES
7 REM START BLUE CURSOR IN CENTER OF SCREEN
10 PRINT "{CLR}{BLU}{12 DOWN}"; TAB(19);
12 REM MAKE SCREEN LIGHT GREY
13 POKE 53281,15
15 REM SET START OF SCREEN, COLOR MEMORY
20 SMEM=1024
30 CMEM=55296
35 REM FUNCTION KEY STRINGS FOR COMPARISONS
40 F2$=CHR$(137): REM F2 KEY LOADS A PICTURE
50 F1$=CHR$(133): REM F1 KEY SAVES A PICTURE
60 F8$=CHR$(140): REM F8 KEY ENDS PROGRAM
65 REM SAVE CHARACTER UNDER CURSOR
70 SPTR=PEEK(209)+PEEK(210)*256+PEEK(211)
80 CHAR=PEEK(SPTR)
85 REM SAVE COLOR OF SPACE UNDER CURSOR
90 CPTR=CMEM+SPTR-SMEM
100 CC=PEEK(CPTR) AND 15
105 REM DISPLAY CURSOR ON TOP OF CHARACTER
110 PRINT "+{LEFT}";
111 FOR K=1 TO 50: NEXT K
112 POKE SPTR,CHAR: POKE CPTR,CC
115 REM READ KEYBOARD
120 GET K$
125 REM RESPOND TO FUNCTION KEYS
128 REM F8 KEY ENDS PROGRAM
130 IF K$=F8$ THEN END
135 REM F2 KEY LOADS PICTURE
140 IF K$=F2$ THEN GOSUB 8000: GOTO 70
145 REM F1 KEY SAVES PICTURE
150 IF K$=F1$ THEN GOSUB 9000
155 REM LOOK FOR JOYSTICK COMMAND
160 IF K$="" THEN GOSUB 15000
170 IF K$="" THEN 110
175 REM ACT ON USER'S KEYSTROKE - PRINT, MOVE CURSOR, ETC.
180 PRINT K$;
190 GOTO 70

7995 REM LOAD PICTURE
8000 PRINT "{HOME}";
8005 REM ASK USER FOR FILENAME
8010 INPUT "LOAD^NAME"; PNAME$
8015 REM OPEN DISK FILE
8020 OPEN 2,8,2,"@0:PIC-"+PNAME$+",S,R"
8025 REM LOAD PICTURE
8030 FOR J=0 TO 40*25-1
8040 INPUT#2, SC, CC
8045 REM PUT SCREEN CODE IN SCREEN MEMORY
8050 POKE SMEM+J, SC
8055 REM PUT COLOR CODE IN COLOR MEMORY
8060 POKE CMEM+J, CC
8070 NEXT J
8075 REM CLOSE FILE
8080 CLOSE 2
8090 PRINT "{HOME}";
8100 RETURN

8995 REM SAVE PICTURE
9000 PRINT "{HOME}";
9005 REM ASK USER FOR FILENAME
```

```
9010 INPUT "SAVE^AS"; PNAME$
9015 REM ERASE PICTURE NAME
9020 PRINT "{HOME}{39 ^}{HOME}";
9025 REM OPEN DISK FILE
9030 OPEN 2,8,2,"@0:PIC-"+PNAME$+",S,W"
9035 REM SAVE PICTURE
9040 FOR J=0 TO 40*25-1
9045 REM SAVE SCREEN, COLOR CODE
9050 PRINT#2, PEEK(SMEM+J); ","; PEEK(CMEM+J) AND 15
9060 NEXT J
9065 REM CLOSE FILE
9070 CLOSE 2
9080 RETURN

14995 REM READ COMMAND FROM JOYSTICK
15000 JOY=PEEK(56320)
15005 REM CONVERT JOYSTICK POSITION TO ARROW KEYS
15010 IF (JOY AND 1)=0 THEN K$="{UP}"
15020 IF (JOY AND 4)=0 THEN K$=K$+"{LEFT}"
15030 IF (JOY AND 8)=0 THEN K$=K$+"{RIGHT}"
15040 IF (JOY AND 2)=0 THEN K$=K$+"{DOWN}"
15050 RETURN
```

Modifications

To eliminate the joystick control, simply delete lines 160 and 14995 through 15050.

If you must use a Datassette instead of a disk drive, substitute the following lines in the program:

```
8015  REM OPEN TAPE FILE FOR READING
8020  OPEN 2,1,0,"PIC-"+PNAME$
9025  REM OPEN TAPE FILE FOR WRITING
9030  OPEN 2,1,1,"PIC-"+PNAME$
```

Since the solid square (reversed space) is a common character in pictures, you may want to assign a function key (say, f5) to entering it. All we must add to the program is:

```
55  F5$=CHR$(135): REM F5 KEY ENTERS A SOLID SQUARE
131  REM F5 KEY ENTERS A SOLID SQUARE
132  IF K$=F5$ THEN K$="{RVS}^{OFF}"
```

Now you can simply press f5 to enter a solid square; you need not bother with {RVS} and {OFF}. Note, however, that line 132 always turns the reverse off.

When you use f2 to load a picture, the cursor starts in the top left-hand corner. To make it start in the center, add the following lines:

```
10  PRINT "{CLR}{BLU}"
62  REM START CURSOR IN CENTER OF SCREEN
64  PRINT "{HOME}{12 DOWN}"; TAB(19);
140  IF K$=F2$ THEN GOSUB 8000: GOTO 64
```

Notes

Lines 110 to 112 blink the cursor by alternating it with the character beneath it. If the program did not blink the cursor, the user might have a difficult time finding it. After all, there could be many shifted + (cursor) characters on the screen.

How does the Commodore 64's own cursor compare to the one in this program? The 64 forms its cursor by alternately displaying the character in the space (in its original color) and the reverse of that character in the printing color. The idea here is not only to produce a flashing position indicator, but also to let the user see the character under the cursor, that character's color, and the printing color.

An easy way to see how this works is to use triangles {*} as the characters. Enter POKE 53281,1 to make the screen white. Type several red triangles in a row, then change the printing color to green and move the cursor back over a triangle. You should see the original red triangle alternating with the reversed green triangle.

One problem immediately comes to mind. Are we looking at a green cursor on top of a red triangle or a red cursor on top of a reversed green triangle? The only way to find out is to type something and observe its color. The controversy here is like arguing about whether one is looking at an orange cat with white markings or a white cat with orange markings.

Things can become even worse. What if the character underneath the cursor is a solid square (that is, a reversed space)? Now we will see a reversed space in its color alternating with a space in the printing color. But since a space has no color, we will not see the printing color at all. Try this by typing several yellow reversed spaces, changing the cursor to cyan, and moving it back over one of the reversed spaces. All you see is a flashing yellow square. However, when you type something, it appears in cyan. Determining the printing color over a reversed space is like identifying the Invisible Man from his picture!

To make PIC-EDIT's cursor look like the 64's, enter the following lines:

```
108  REM CREATE CURSOR BY ALTERNATING REVERSE AND CHARACTER
109  REM DISPLAY REVERSE OF CHARACTER IN CURRENT PRINTING COLOR
110  POKE SPTR,(CHAR+128) AND 255: POKE CPTR,PEEK(646)
```

The code for a reversed character is 128 larger than the code for the normal character. Line 110 not only converts a normal character into its reversed version, but also converts a reversed character into its normal version. Either way, what we see is a flashing character. Memory address 646 contains the code for the current printing color.

Lines 8020 and 9030 use "PIC−" +PNAME$ as the file name. This gives all picture files a prefix of "PIC−" so that you can pick them out in a disk

directory. Remember that a + between strings puts the second one after the first one (this is usually called *concatenation*).

Example

As an example of using PIC–EDIT, let us draw the lunar explorer shown in Figure 5–1. Load the program and RUN it. Proceed as follows:

1. Press {UP} 4 times and {LEFT} twice to reach the explorer's top left-hand corner (column 17 of line 16).
2. Press {RVS} and enter four spaces to draw the top line. Note that pressing {RVS} reverses the cursor as well as the entries.
3. Press Return and {RIGHT} or the space bar (16 times) to reach the beginning of the second line of the picture. Pressing Return turns the reverse mode off; you could also use {DOWN} to move the cursor down a line without affecting the reverse mode.
4. Press **{I}**, {RVS}, space, {RIGHT} twice, space, {OFF}, and **{I}** to draw the second line. Remember that when you have the reverse mode on, pressing the space bar enters solid squares; you must press {RIGHT} to just move the cursor right.
5. Press Return and use {RIGHT} (15 times) to reach the beginning of the third line. To draw it, press {RVS} and enter eight spaces.
6. The fourth, fifth, and sixth lines are all the same. Press Return and use {RIGHT} to reach the beginning (column 15), then draw each line with {RVS}, space, {RIGHT}, 4 spaces, {RIGHT}, space, and Return.
7. The seventh and eighth lines are the same; both start in column 17 and consist of {RVS}, space, 2 {RIGHT}s, and another space.
8. The ninth (bottom) line starts in column 16 and consists of **{I}**, {RVS}, space, 2 {RIGHT}s, space, {OFF}, and **{I}**.

One way to simplify the drawing of this figure is by implementing the solid square function key described under Modifications. This will allow you to enter the solid squares without pressing {RVS} or {OFF}.

After you finish drawing the figure, press f1 to save it. When the message SAVE AS? appears on the screen, enter LUNEXP as the figure's name. Remember that PIC–EDIT puts the prefix PIC— in front of this name automatically to indicate that the file contains a picture. Be patient; saving or loading a picture takes quite a bit of time, especially with a Datassette.

Now let us reload the picture and edit it. Press {CLR} to clear the screen and f2 (shifted f1) to load a picture. When the message LOAD NAME? appears on the screen, enter LUNEXP and press Return. The figure reappears, but the cursor is in the top left-hand corner. Move the cursor down to the figure and use the color keys to make the top two lines black and the body

and arms (the next four lines) red. Watch that you press {RVS} and {OFF} at the right times. Press f1 to save the modified picture; make its name COLEXP.

Plate 7 and Figure 5–2 contain a more complicated picture you can try drawing with PIC–EDIT. Make the butterfly's body and antennae black, its wings light red, yellow, and blue, and its eyes orange. Put a red border around the outer part of the wings. Save the picture under the name BUTTERFLY.

Hints and Warnings

Watch the following when using PIC–EDIT:

1. Don't press RUN/STOP. This will make the computer leave PIC–EDIT, and you will lose your picture.
2. To keep errors such as accidentally pressing RUN/STOP or [CLR] from being too costly, save your picture on tape or disk occasionally.
3. You can tell whether the reverse mode is on by looking at the cursor. It will be reversed if {RVS} is on. Remember that pressing Return turns {RVS} off.

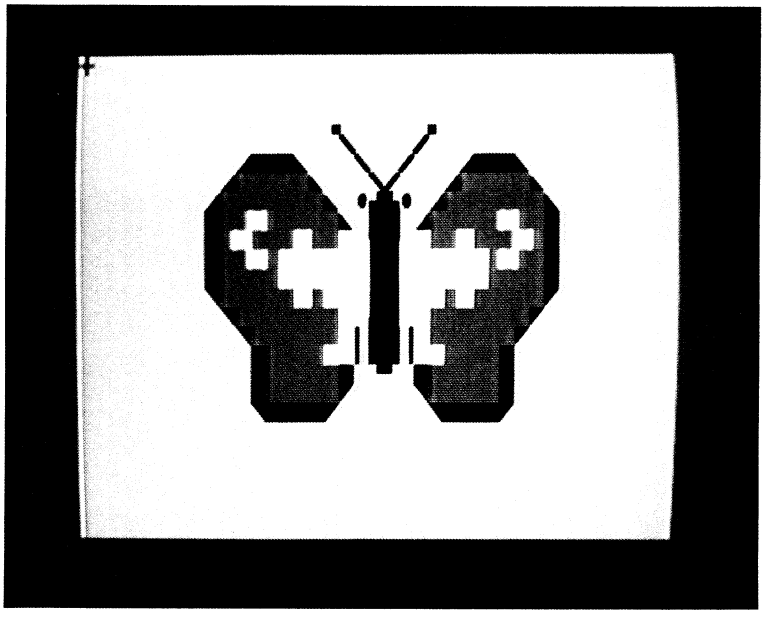


Figure 5–2 A PIC–EDIT butterfly

4. You can also determine the current printing color from the cursor. Be careful; if you make the printing color light gray, the cursor will disappear. Also, you may want to make the printing color blue or black before starting a sequence (e.g., loading or saving a picture) that involves prompts. Otherwise, the prompts may be difficult to read.
5. Remember that you can use {CLR}, {HOME}, {INST}, {DEL}, the CRSR keys, and SHIFT LOCK whenever they are convenient.
6. Be certain to write down the names of all pictures you save. Unfortunately, you cannot get a disk directory without exiting from PIC-EDIT.

References

The *Commodore 64 Programmer's Reference Guide* contains descriptions of screen and color memory (pp. 102–103) and key memory locations (pp. 310–334).

ADVANCED PICTURE EDITOR

Program SKETCH is an expansion of PIC-EDIT. It adds function keys the artist can use to draw a line or circle, or change the background color. If we had more function keys and more memory, we could add even more capabilities. The resulting artist's toolbox would then be a little like the so-called "electronic paint" systems used by television networks and advertising firms. These expensive, high-performance systems provide the graphic artist with tools he or she can use to create and save pictures, draw lines and shapes, mix colors, and simulate various kinds of paintbrushes. While an inexpensive Commodore 64 cannot compete with such marvels, it can demonstrate the ideas behind them.

Program Name: "SKETCH"

Purpose

Allows the user to draw and edit pictures with the aid of the geometric routines from Chapter 3. The user can draw lines or circles, load or save a picture, and change the background color.

Figures 5–3 and 5–4 (page 157 show typical pictures drawn with SKETCH.

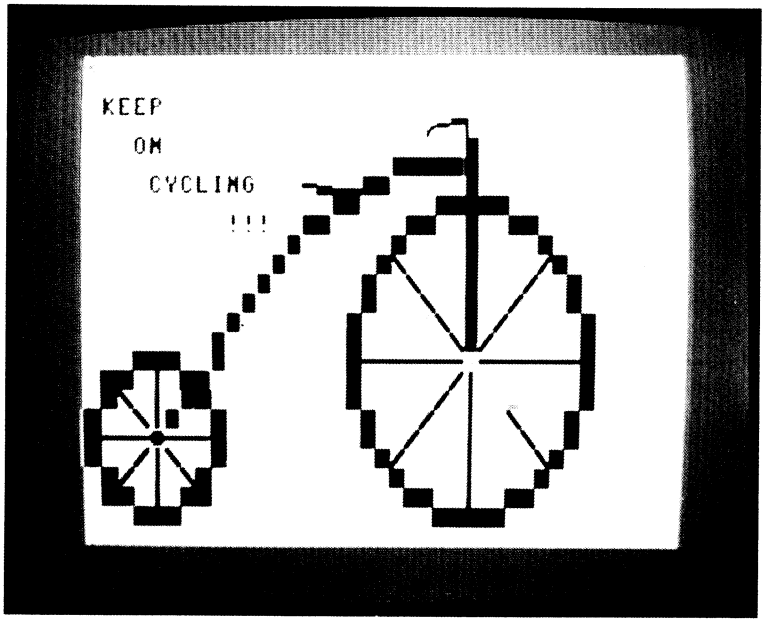


Figure 5–3 SKETCH a bicycle

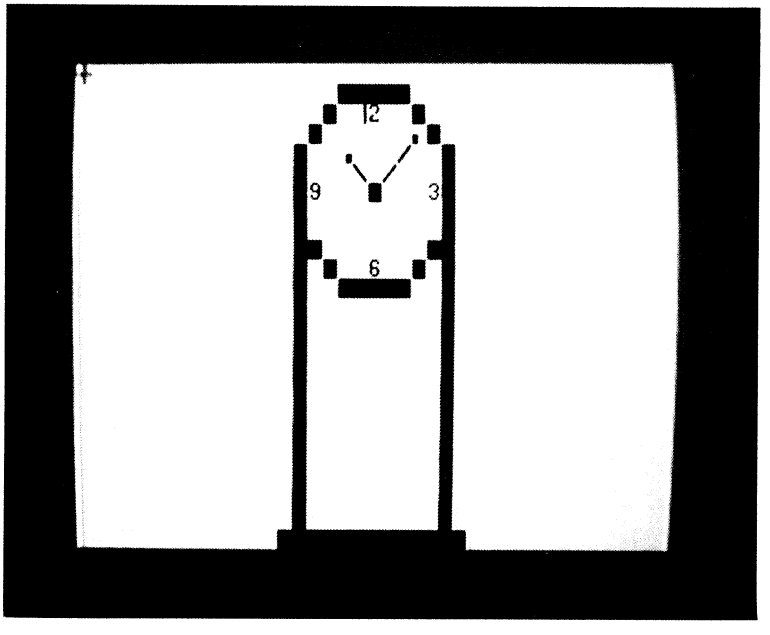


Figure 5–4 Grandfather clock drawn with SKETCH

Techniques Demonstrated

Building a useful program from several separate routines.

Procedure

As with PIC-EDIT, you can move the cursor with the {UP}, {DOWN}, {LEFT}, {RIGHT}, and {HOME} keys, or with the joystick. You can draw objects by entering character or graphics symbols and color commands (such as {RED} or {CYN}. The user can enter commands by pressing the following function keys:

f1 Save the picture in a disk file.
f2 Load a picture from a disk file.
f3 Draw a line from the cursor position to a point you specify.
f4 Draw a circle, with a radius you specify, centered at the cursor position.
f7 Change the background color. The screen will take on a different color each time you press f7; there are 16 possibilities in all.
f8 Stop the program.

Variables

CC: Color code for space where cursor is located
CHAR: Screen code of character where cursor is located
CMEM: Starting address of color memory
CPTR: Color memory address corresponding to current cursor position
F1$, F2$, F3$, F4$, F7$, F8$: Strings containing the character equivalents of function keys
J, K: Counters
K$: Key pressed by the user
PLOT$: Solid square
PNAME$: Name of tape file
R: Radius of a circle
SCODE: Screen color code
SC: Screen code value
SMEM: Starting address of screen memory
SPTR: Screen memory address corresponding to current position
VERT$: Vertical positioning string

X, Y: Coordinates

X1, Y1: Cursor row and column

X2, Y2: Row and column of the line's endpoint

Special Cases

As with PIC-EDIT, you should not place the cursor in the bottom right-hand corner or press {DOWN} when the cursor is on the bottom line.

Also as with PIC-EDIT, loading or saving pictures with a Datassette takes a long time. Be sure you have a lot of tape.

Brief Description

Line 10 starts the cursor near the center of the screen.

Line 13 makes the screen light gray.

Lines 20 to 30 determine the starting addresses of both screen and color memory.

Lines 40 to 60 define the character equivalents of the function keys.

Lines 70 to 190 read the keyboard and joystick and perform commands entered by the user.

Lines 300 to 340 change the screen's background color. The Notes describe in detail how this is done.

Line 400 erases the top screen line. This is in preparation for prompts that ask the user for the endpoint of a line, the radius of a circle, or the name of a picture file.

Lines 495 to 560 ask for the horizontal and vertical distances from the cursor to the line's endpoint. Line 550 calls the line-drawing subroutine.

Lines 695 to 760 ask for the circle's radius, then call the circle-drawing subroutine.

Lines 740 to 750 return the cursor to the center of the circle.

Lines 895 to 930 set X1 and Y1 to the row and column position of the cursor.

Lines 995 to 1160 draw a line from X1, Y1 to X2, Y2, clipping it if it goes off the screen.

Lines 1595 to 1790 draw a circle of radius R centered at X1, Y1, clipping any part that goes off the screen.

Lines 7995 to 8090 restore a picture from a disk file.

Lines 8995 to 9080 save a picture in a disk file.

Lines 9995 to 10010 print characters starting at row X, column Y.
Lines 14995 to 15060 read the joystick position and convert it to cursor moves.

Listing

```
5 REM "SKETCH" COMPUTER-AIDED DRAWING
7 REM START BLUE CURSOR IN CENTER OF SCREEN
10 PRINT "{CLR}{BLU}{11 DOWN}{19 RIGHT}";
12 REM MAKE SCREEN LIGHT GREY
13 POKE 53281,15
15 REM SET START OF SCREEN, COLOR MEMORY
20 SMEM=1024
30 CMEM=55296
32 VERT$="{HOME}{24 DOWN}"
34 PLOT$="{RVS}^{OFF}"
35 REM DEFINE STRINGS FOR FUNCTION KEYS
40 F1$=CHR$(133)
50 F2$=CHR$(137)
52 F3$=CHR$(134)
54 F4$=CHR$(138)
56 F7$=CHR$(136)
60 F8$=CHR$(140)
65 REM SAVE CHARACTER UNDERNEATH CURSOR
70 SPTR=PEEK(209)+PEEK(210)*256+PEEK(211)
80 CHAR=PEEK(SPTR)
85 REM SAVE COLOR UNDERNEATH CURSOR
90 CPTR=CMEM+SPTR-SMEM
100 CC=PEEK(CPTR) AND 15
105 REM DISPLAY CURSOR
110 PRINT "+{LEFT}";
111 REM PAUSE
112 FOR K=1 TO 50: NEXT K
113 REM REPLACE ORIGINAL CHARACTER, COLOR
114 POKE SPTR,CHAR: POKE CPTR,CC
120 GET K$
125 REM RESPOND TO FUNCTION KEYS
128 REM F8 ENDS PROGRAM
130 IF K$=F8$ THEN END
135 REM F2 LOADS A PICTURE
140 IF K$=F2$ THEN GOSUB 8000: GOTO 70
145 REM F1 SAVES A PICTURE
150 IF K$=F1$ THEN GOSUB 9000
151 REM F7 CHANGES SCREEN COLOR
152 IF K$=F7$ THEN GOSUB 300
153 REM F3 DRAWS A LINE
154 IF K$=F3$ THEN GOSUB 500: GOTO 70
155 REM F4 DRAWS A CIRCLE
156 IF K$=F4$ THEN GOSUB 700
157 REM LOOK FOR JOYSTICK COMMANDS
160 IF K$="" THEN GOSUB 15000
170 IF K$="" THEN 110
180 PRINT K$;
190 GOTO 70

295 REM CHANGE THE COLOR BY ONE LINE IN TABLE 1-2
298 REM GET OLD SCREEN COLOR CODE
300 SCODE=PEEK(53281) AND 15
305 REM MOVE DOWN ONE LINE TO NEXT SCREEN COLOR
310 SCODE=SCODE+1
315 REM WRAP AROUND FROM LIGHT GREY TO BLACK
320 IF SCODE>15 THEN SCODE=0
```

```
325 REM REPLACE SCREEN COLOR CODE
330 POKE 53281,SCODE
340 RETURN

395 REM ERASE TOP LINE AND HOME CURSOR
400 PRINT "{HOME}{38 ^}{HOME}";
410 RETURN

495 REM CURSOR MOVE AND LINE-DRAWING SUBROUTINE
498 REM MARK CURSOR POSITION, SET X1,Y1 TO IT
500 PRINT "+{LEFT}";: GOSUB 900
505 REM ASK FOR DISTANCES TO ENDPOINT
510 GOSUB 400: INPUT "X^DISTANCE"; XD
520 GOSUB 400: INPUT "Y^DISTANCE"; YD
525 REM ERASE PROMPT LINE, RESTORE CHARACTER UNDER CURSOR
530 GOSUB 400: POKE SPTR,CHAR: POKE CPTR,CC
540 X2=X1+XD: Y2=Y1+YD
545 REM DRAW LINE IF NECESSARY
547 IF X1=X2 AND Y1=Y2 THEN GOSUB 800: GOTO 570
550 GOSUB 1000
555 REM MOVE CURSOR TO END OF LINE
560 PRINT "{LEFT}";
570 RETURN

695 REM DRAW CIRCLE AROUND CURSOR POSITION
698 REM MARK CURSOR POSITION, SET X1,Y1 TO IT
700 PRINT "+{LEFT}";: GOSUB 900
705 REM ASK FOR RADIUS OF CIRCLE
710 GOSUB 400: INPUT "RADIUS"; R
715 REM ERASE PROMPT LINE, RESTORE CHARACTER UNDER CURSOR
720 GOSUB 400: POKE SPTR,CHAR: POKE CPTR,CC
725 REM DRAW CIRCLE
730 GOSUB 1600
735 REM RETURN CURSOR TO X1,Y1
740 GOSUB 800
750 RETURN

795 REM SET CURSOR POSITION TO X1,Y1
800 X=X1: Y=Y1: PLOT$=""
810 GOSUB 10000
820 PLOT$="{RVS}^{OFF}"
830 RETURN

895 REM SET X1,Y1 TO CURSOR POSITION
900 X1=PEEK(211)
910 Y1=24-PEEK(214)
920 IF X1>39 THEN X1=X1-40: GOTO 920
930 RETURN

995 REM DRAW LINE FROM X1,Y1 TO X2,Y2
1000 M=1000
1010 IF X1<>X2 THEN M=(Y1-Y2)/(X1-X2)
1015 REM IF LINE IS STEEP, USE VERTICAL STEPS
1020 IF ABS(M)>1 THEN GOTO 1100
1025 REM NOT STEEP, SO STEP HORIZONTALLY
1035 REM COMPUTE Y-AXIS INTERCEPT
1040 B=Y1-M*X1
1050 S=SGN(X2-X1)
1060 FOR X=X1 TO X2 STEP S
1070 Y=INT(M*X+B+0.5)
1075 IF X>=0 AND X<=38 AND Y>=0 AND Y<=23 THEN GOSUB 10000
1080 NEXT X
1090 RETURN
1100 IF Y1=Y2 THEN RETURN
1110 A=X1-Y1/M
```

```
1120 S=SGN(Y2-Y1)
1130 FOR Y=Y1 TO Y2 STEP S
1140 X=INT(Y/M+A+0.5)
1145 IF X>=0 AND X<=38 AND Y>=0 AND Y<=23 THEN GOSUB 10000
1150 NEXT Y
1160 RETURN

1595 REM PLOT CIRCLE STEPPING HORIZONTALLY
1600 FOR XP=0 TO R/SQR(2)+0.5
1610 YP=SQR(R*R-XP*XP)
1620 RX=XP
1630 RY=YP
1640 GOSUB 1700
1650 RX=YP
1660 RY=XP
1670 GOSUB 1700
1680 NEXT XP
1690 RETURN

1695 REM PLOT FOUR REFLECTIONS OF RX, RY AROUND X1,Y1
1700 X=INT(X1+RX+0.5)
1710 Y=INT(Y1+RY+0.5)
1720 IF X>=0 AND X<=38 AND Y>=0 AND Y<=23 THEN GOSUB 10000
1730 Y=INT(Y1-RY+0.5)
1740 IF X>=0 AND X<=38 AND Y>=0 AND Y<=23 THEN GOSUB 10000
1750 X=INT(X1-RX+0.5)
1760 IF X>=0 AND X<=38 AND Y>=0 AND Y<=23 THEN GOSUB 10000
1770 Y=INT(Y1+RY+0.5)
1780 IF X>=0 AND X<=38 AND Y>=0 AND Y<=23 THEN GOSUB 10000
1790 RETURN

7995 REM LOAD PICTURE
8000 PRINT "{HOME}";
8005 REM ASK USER FOR FILENAME
8010 INPUT "LOAD^NAME"; PNAME$
8015 REM OPEN DISK FILE
8020 OPEN 2,8,2,"@0:PIC-"+PNAME$+",S,R"
8025 REM LOAD PICTURE
8030 FOR J=0 TO 40*25-1
8040 INPUT#2, SC, CC
8045 REM PUT SCREEN CODE IN SCREEN MEMORY
8050 POKE SMEM+J, SC
8055 REM PUT COLOR CODE IN COLOR MEMORY
8060 POKE CMEM+J, CC
8070 NEXT J
8080 CLOSE 2
8090 PRINT "{HOME}";
8100 RETURN

8995 REM SAVE PICTURE
9000 PRINT "{HOME}";
9005 REM ASK USER FOR FILENAME
9010 INPUT "SAVE^AS"; PNAME$
9020 PRINT "{HOME}{39 ^}{HOME}";
9025 REM OPEN DISK FILE
9030 OPEN 2,8,2,"@0:PIC-"+PNAME$+",S,W"
9035 REM SAVE PICTURE
9040 FOR J=0 TO 40*25-1
9045 REM SAVE SCREEN, COLOR CODE
9050 PRINT#2, PEEK(SMEM+J); ","; PEEK(CMEM+J) AND 15
9060 NEXT J
9070 CLOSE 2
9080 RETURN
```

```
9995 REM PRINT PLOT$ AT X,Y
10000 PRINT LEFT$(VERT$,25-Y);SPC(X) PLOT$;
10010 RETURN

14995 REM TAKE COMMAND FROM JOYSTICK
15000 JOY=PEEK(56320)
15005 REM CONVERT JOYSTICK POSITION TO ARROW KEYS
15010 IF (JOY AND 1)=0 THEN K$="{UP}"
15020 IF (JOY AND 4)=0 THEN K$=K$+"{LEFT}"
15030 IF (JOY AND 8)=0 THEN K$=K$+"{RIGHT}"
15040 IF (JOY AND 2)=0 THEN K$=K$+"{DOWN}"
15050 RETURN
```

Modifications

To allow the user to move the cursor anywhere without drawing a line, add the following:

```
153 REM F3 MOVES CURSOR AND DRAWS LINE TO NEW POSITION ON REQUEST

525 REM ASK USER WHETHER TO DRAW LINE
530 GOSUB 400: INPUT "LINE (Y/N)"; LINE$
532 IF LINE$<>"Y" AND LINE$<>"N" THEN 530
533 REM ERASE PROMPT LINE, RESTORE CHARACTER UNDER CURSOR
534 GOSUB 400: POKE SPTR,CHAR: POKE CPTR,CC
545 REM USE NULL CHARACTER IF NO LINE REQUESTED
550 IF LINE$="N" THEN PLOT$=""
555 REM DRAW LINE IF NECESSARY
557 IF X1=X2 AND Y1=Y2 THEN GOSUB 800: GOTO 590
560 GOSUB 1000
570 PLOT$="{RVS}^{OFF}"
575 REM MOVE CURSOR TO END OF LINE
580 IF LINE$="Y" THEN PRINT "{LEFT}";
590 RETURN
```

Now f3 acts like the G (Go forward) command in TURTLE, where you can have the pen either up (move cursor only) or down (move cursor and draw a line).

To make the cursor return to its current position after you save a program, replace line 150 with:

```
150  IF K$=F1$ THEN GOSUB 900: GOSUB 8000: GOSUB 800
```

The lines starting at 900 save the cursor's position, while the lines starting at 800 restore the old position. This modification is particularly convenient if you are saving a partially completed picture. It allows you to save the picture and then continue without having to reposition the cursor.

To add new commands to SKETCH, proceed as follows:

1. Find a gap in lines 1900 through 7900 in which to fit the routine that performs the command.

2. Select a function key that is not already in use (f5 or f6). Use f6 if you are using f5 to enter a reversed space as described earlier.
3. Insert a line between 40 and 60 that calculates the string equivalent of the function key's ASCII value.
4. Find a gap in line numbers 125 through 160. Insert a statement that GOSUBs to the new routine when the user presses the function key.

For example, to add a command that draws a rectangle when you press f6, you should insert the lines:

```
56  F6$=CHR$(139)
158  IF K$=F6$ THEN GOSUB 3000
```

Then add the rectangle-drawing subroutine starting at line 3000.

When SKETCH saves or loads a picture, it does not consider the background color. We can remedy this by adding the following lines:

```
8075  INPUT#2, SCODE
8077  POKE 53281,SCODE
9065  PRINT#2, PEEK(53281)
```

To save the file on tape instead of disk, use the modification presented in PIC–EDIT.

One way to expand the number of functions we can have is to use sequences of function keys. For example, we could make f6 an entry point into an entire set of functions by adding lines such as:

```
157  REM F6 MEANS LOOK FOR ANOTHER FUNCTION KEY
158  IF K$=F6$ THEN 2000
1995  REM INTERPRET SEQUENCES STARTING WITH F6
2000  GET K$: IF K$="" THEN 2000
2005  REM F6-R DRAWS A RECTANGLE
2010  IF K$="R" THEN GOSUB 3000
```

Now, to draw a rectangle, you must press first f6 and then R. This expansion allows us additional functions at the cost of an extra keystroke. Obviously, the best design approach is to use single keystrokes for the most frequent tasks and double keystrokes for less common operations.

Notes

Lines 900 through 920 show how to determine the cursor's current X and Y coordinates.

The color change produced by pressing f7 works as follows. Memory

location 53281 contains a code for the screen color as described in Table 5–1. Lines 300 through 330 obtain that code, increase it by 1 to move down one line in the table, and then provide wraparound from the bottom line back to the top line. This changes the screen color without affecting the border color. For example:

1. If the screen is black, it becomes white.
2. If the screen is blue, it becomes yellow.
3. If the screen is light gray, it becomes black.

The only way you can determine the new color is by examining Table 5–1. Of course, you can just keep pressing f7 until the screen color is what you want. It could take up to 15 tries. This works like selecting an item from a vending machine that rotates the choices. You just keep moving the items until the one you want is in the dispensing position, then you put in your money and select the item.

This may seem awkward, but note that you hardly ever change the background color. Thus, having a separate key for each color would be wasteful; besides, we have only eight function keys. Remember that the regular color keys are already in use to change the printing color.

While lines 300 to 330 do their job in a simple but obvious fashion, we could actually replace them with the single line:

```
300 POKE 53281,(PEEK(53281)+1) AND 15
```

The AND 15 provides the same wraparound as line 320.

As an example of using SKETCH, let us draw the grandfather clock shown in Figure 5–4. Load the program and RUN it. Proceed as follows:

1. Move the cursor up 5 lines. It should end up in column 19 of line 18 (6 lines down from the top).
2. Press f4 (shifted f3) to draw a circle. Enter 5 as the radius and press Return.
3. Mark the center of the circle with a reversed space.
4. Enter the numerals 3, 6, 9, and 12 where indicated. You can determine the positions from the circle. The only problem is that we cannot center the two digits of 12. Our solution is to center the 2 and type {M} instead of 1 to the left of it. {M}is a solid vertical line at the far right edge of its position. It looks like a 1, but appears further right. The resulting 12 is thus aligned better with the 6 at the bottom of the clock.
5. Enter the clock's hands starting from the center. The hour hand consists of {D} and M, while the minute hand consists of two N's and {F}.

6. Move the cursor just under the solid line that forms the outer left-hand boundary of the clock. Press f3 to draw a line. Enter 0 as the X distance and −15 as the Y distance. This forms the left-hand border of the clock's cabinet.
7. Press {LEFT} to move the cursor left one column.
8. Press f3 to draw a line. Enter 12 as the X distance and 0 as the Y distance. This forms the bottom border of the clock's cabinet.
9. Press {LEFT} to move the cursor left one column.
10. Press f3 to draw a line. Enter 0 as the X distance and 15 as the Y distance. This forms the right-hand border of the clock's cabinet, thus completing the picture. Save it under the name GRANDFATHER.

You may want to try the following modifications:

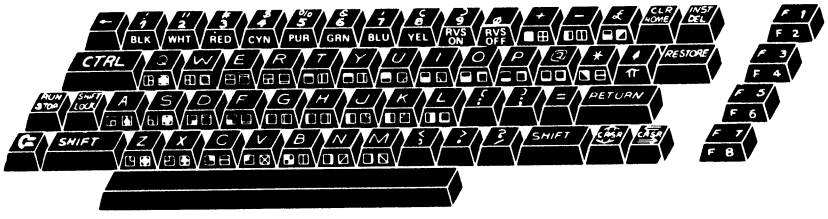
1. Make the clock brown by drawing over the circle and lines. Remember, you change the printing color to brown by pressing {2}.
2. Fill in the clock's cabinet with reversed spaces. You could also add some decorations.
3. Change the 3, 6, 9, and 12 to Roman numerals. 3 should be III, 6 is VI, 9 is IX, and 12 is XII (you do remember your Roman numbers, don't you?).
4. Add a date indicator to create a modern grandfather clock (a "swinging senior," perhaps made out of biodegradable, high-impact plastic). Starting from the clock's center, move the cursor down two lines and left three columns. Then press {RVS} and enter the month as a three-character abbreviation (e.g., 06 for the 6th). The reversed digits add a nice touch.

You can also use the circle-drawing function to produce a colorful bullseye, as illustrated in Plate 8. Simply clear the screen, center the cursor, and use f4 to draw successively larger circles, each in a different color.

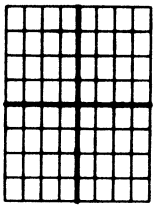
Figure 5–3 contained a more complex picture you can try drawing with SKETCH. Circles form the bicycle's wheels, while lines form its frame. When drawing lines, remember that the X distance is positive to the right and negative to the left, while the Y distance is positive up and negative down.

APPENDICES

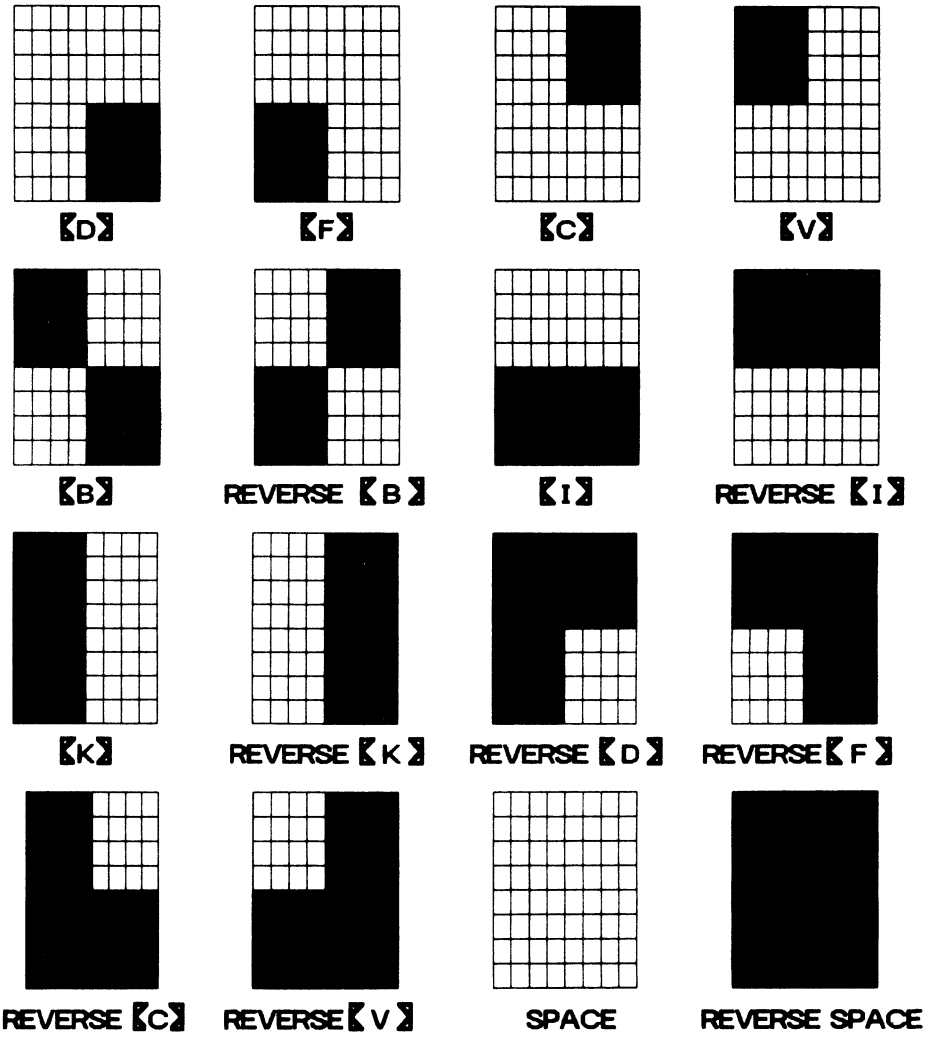
APPENDIX A: THE KEYBOARD



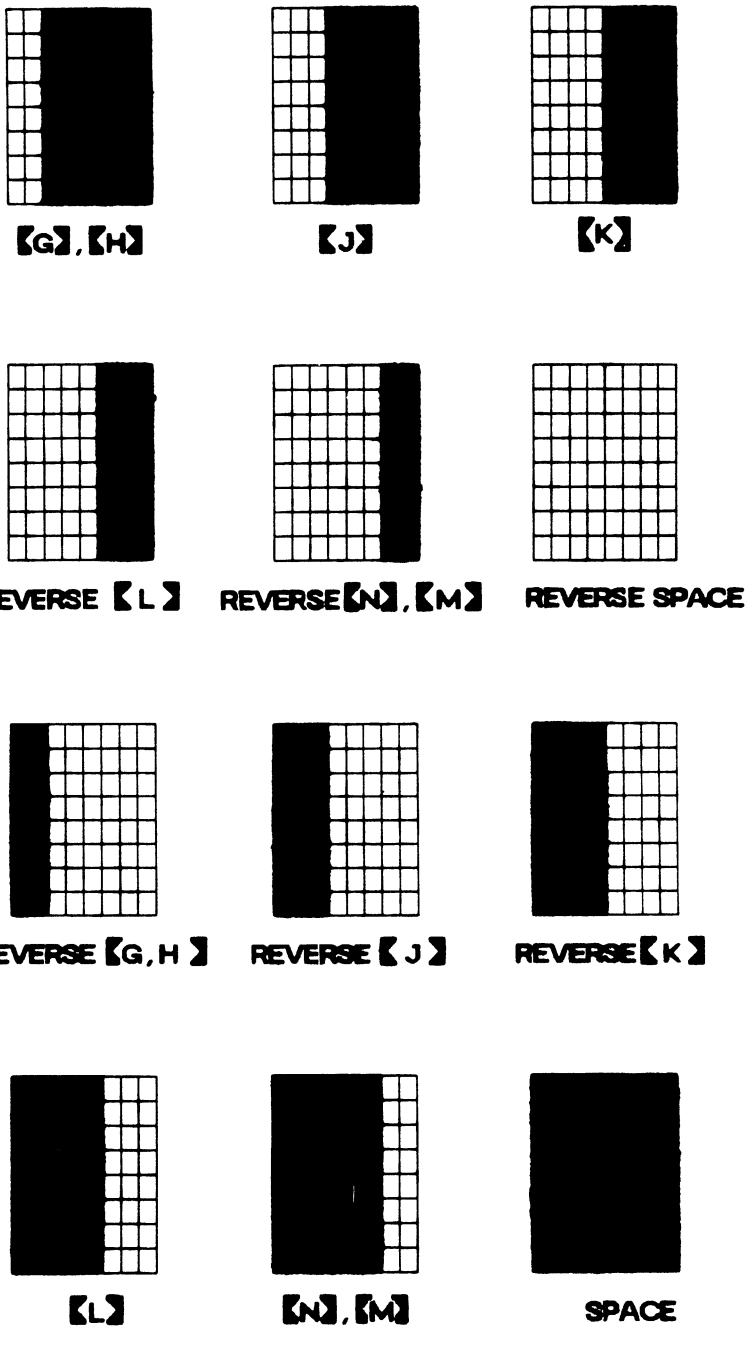
APPENDIX B: THE CHARACTER SET



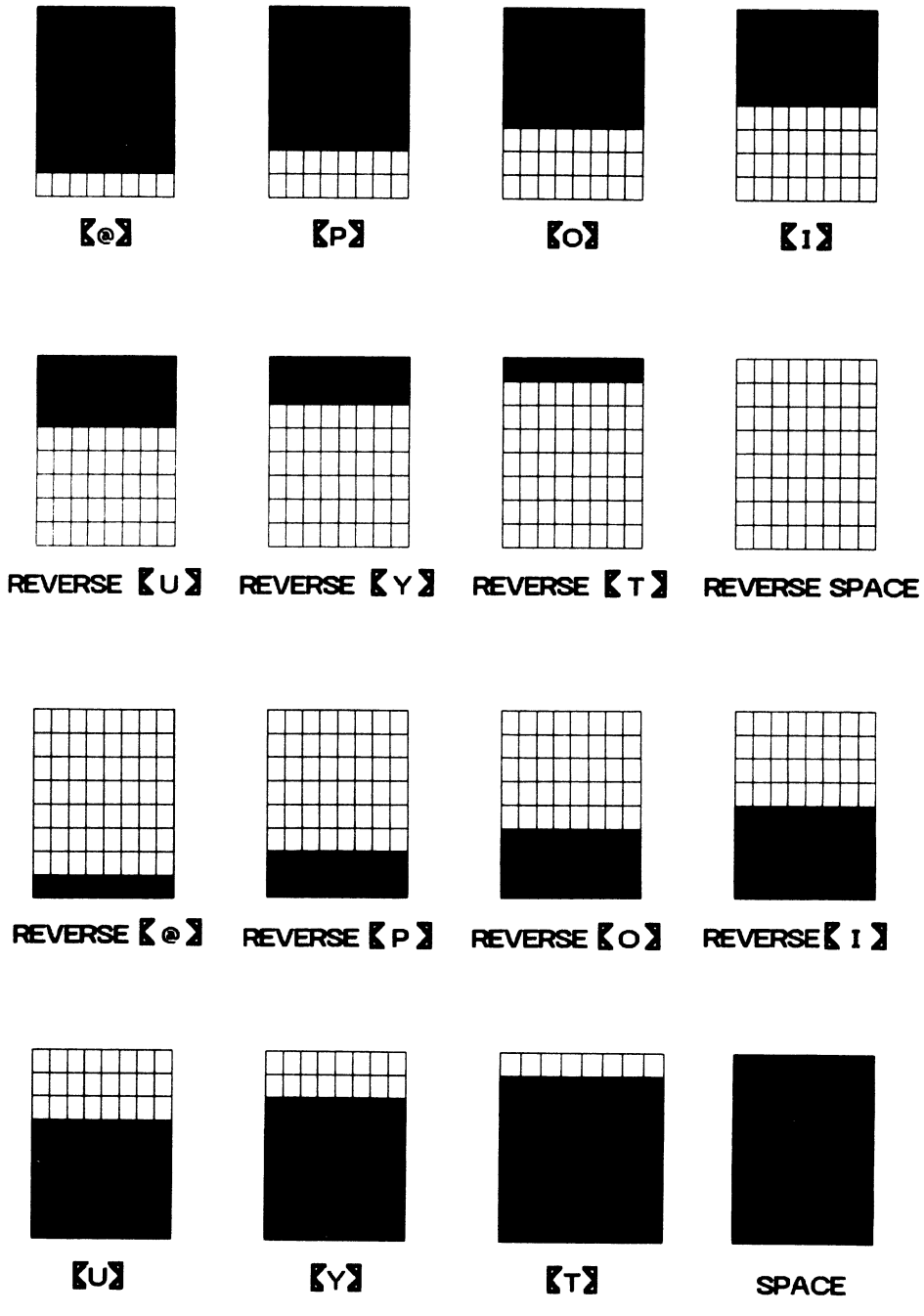
Character space 8 × 8 grid divided into 4 parts



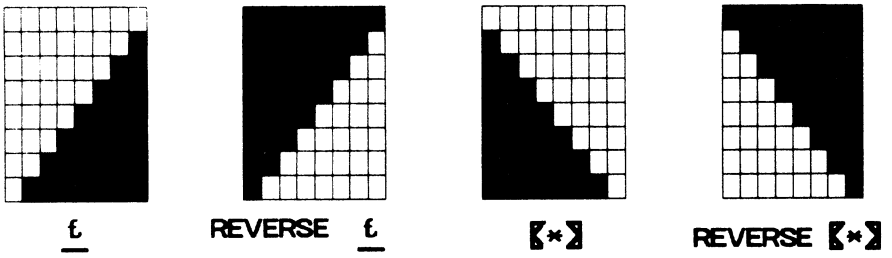
Graphics symbols required to draw with quarter-space resolution



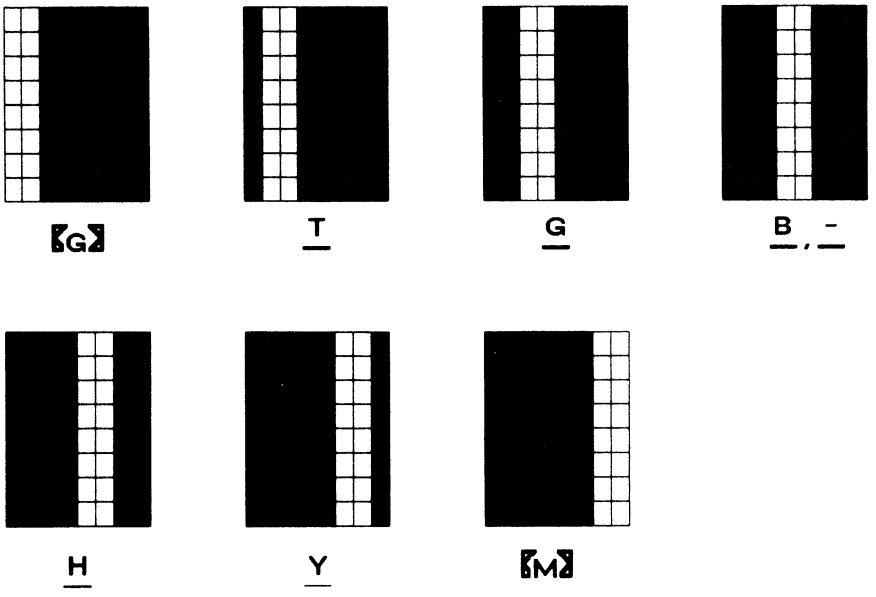
Graphics symbols required to draw with one-eighth space resolution horizontally



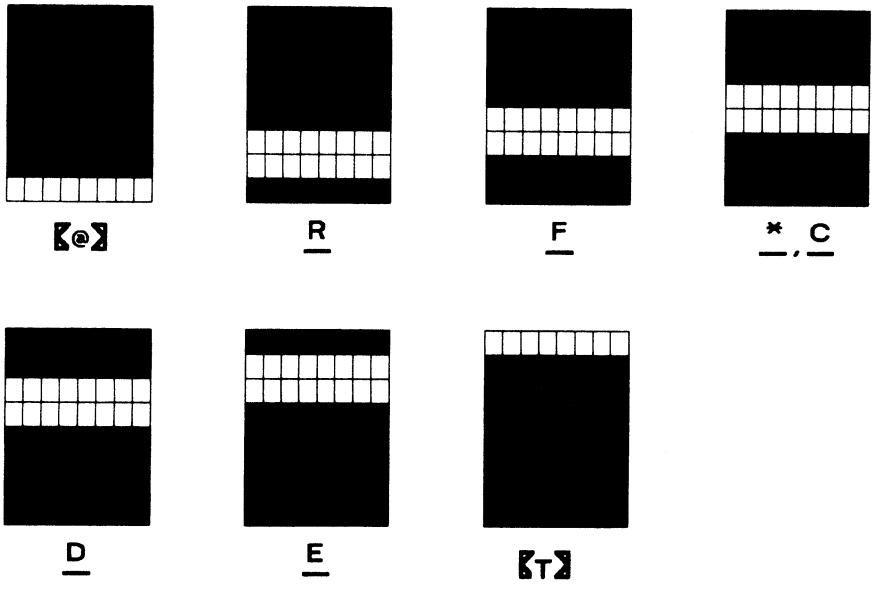
Graphics symbols required to draw with one-eighth space resolution vertically



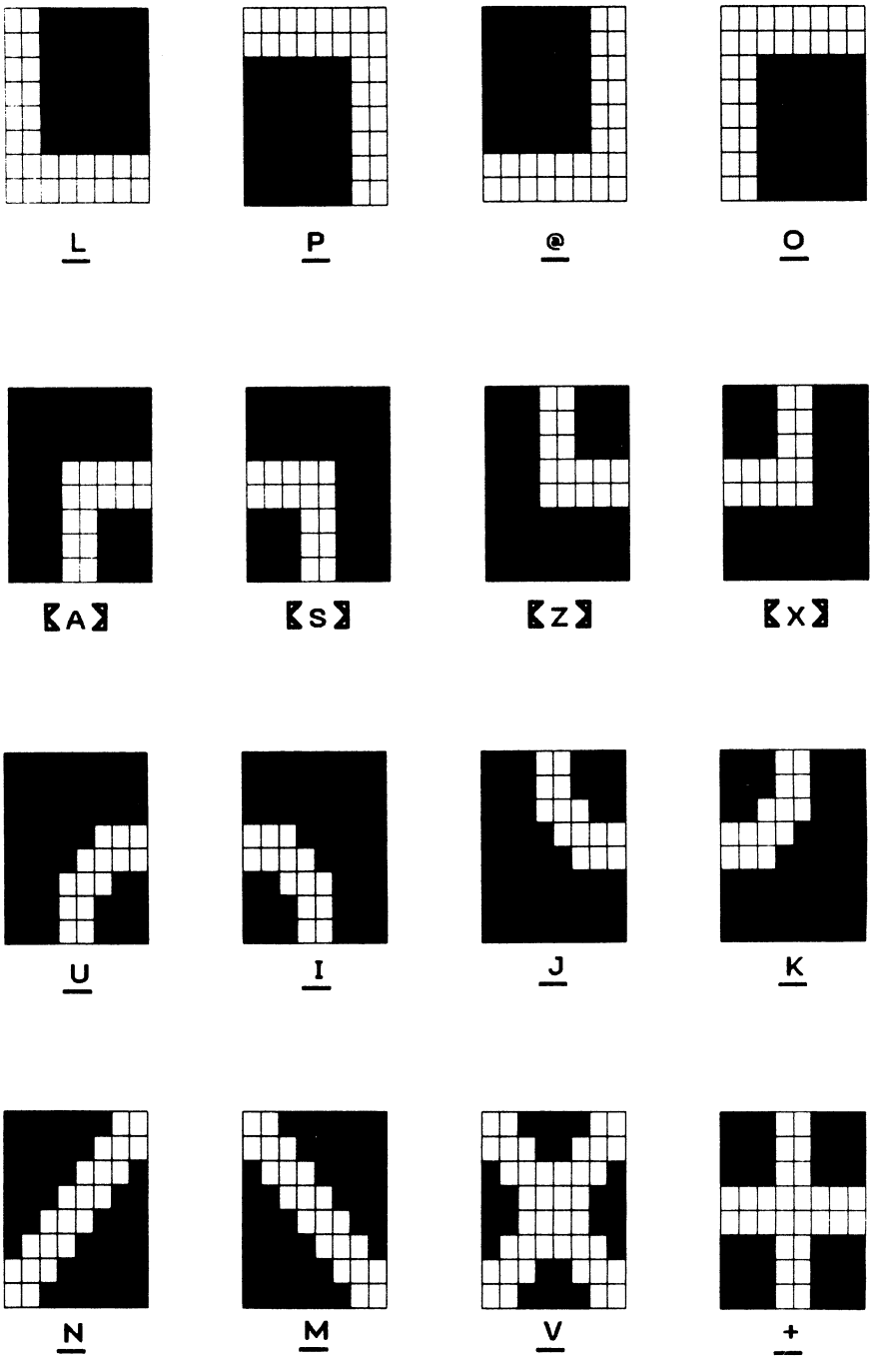
Graphics symbols required to draw with half-space triangles



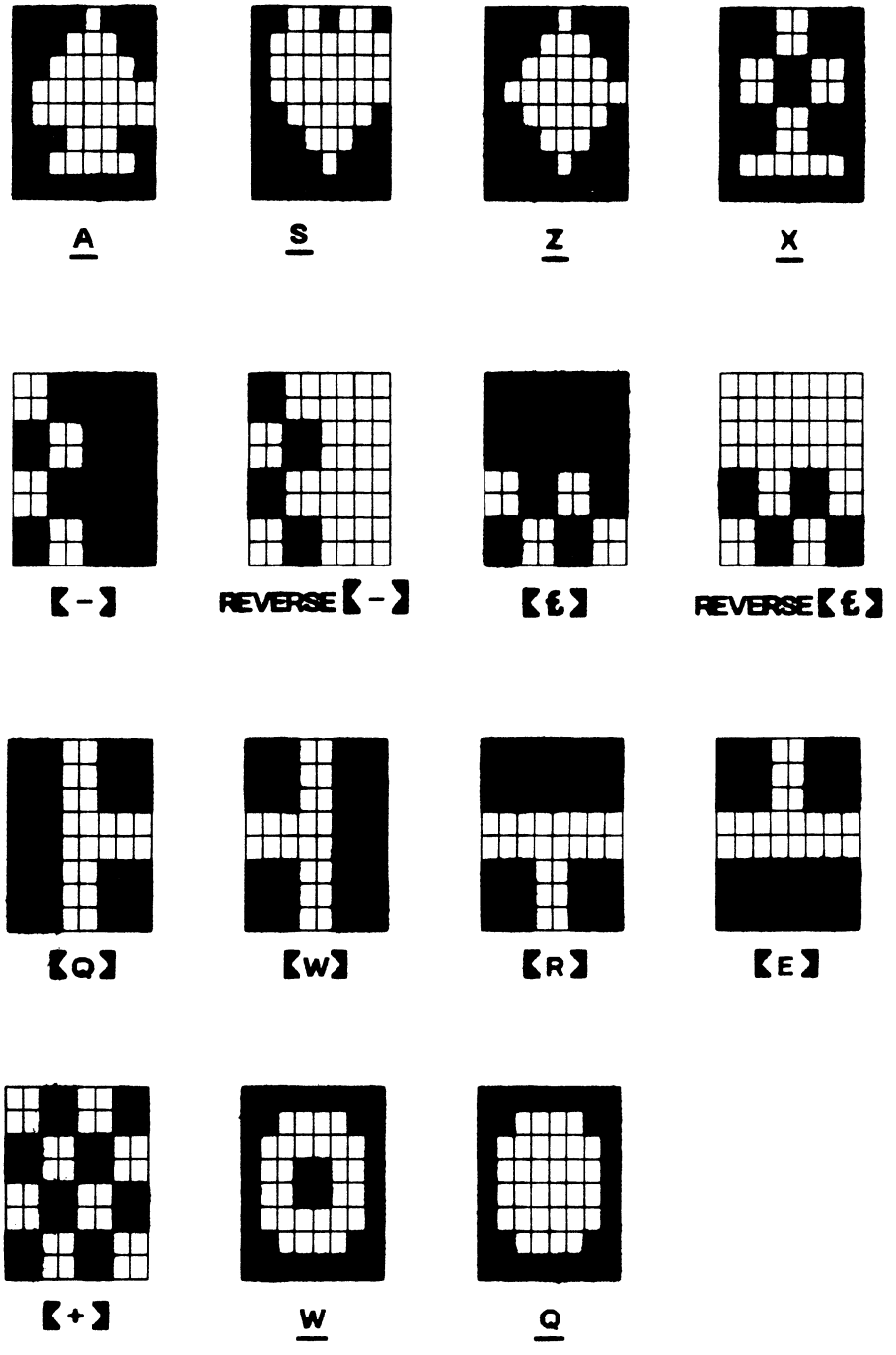
Graphics symbols required to draw vertical lines at one-eighth space resolution



Graphics symbols that draw horizontal lines at one-eighth space resolution

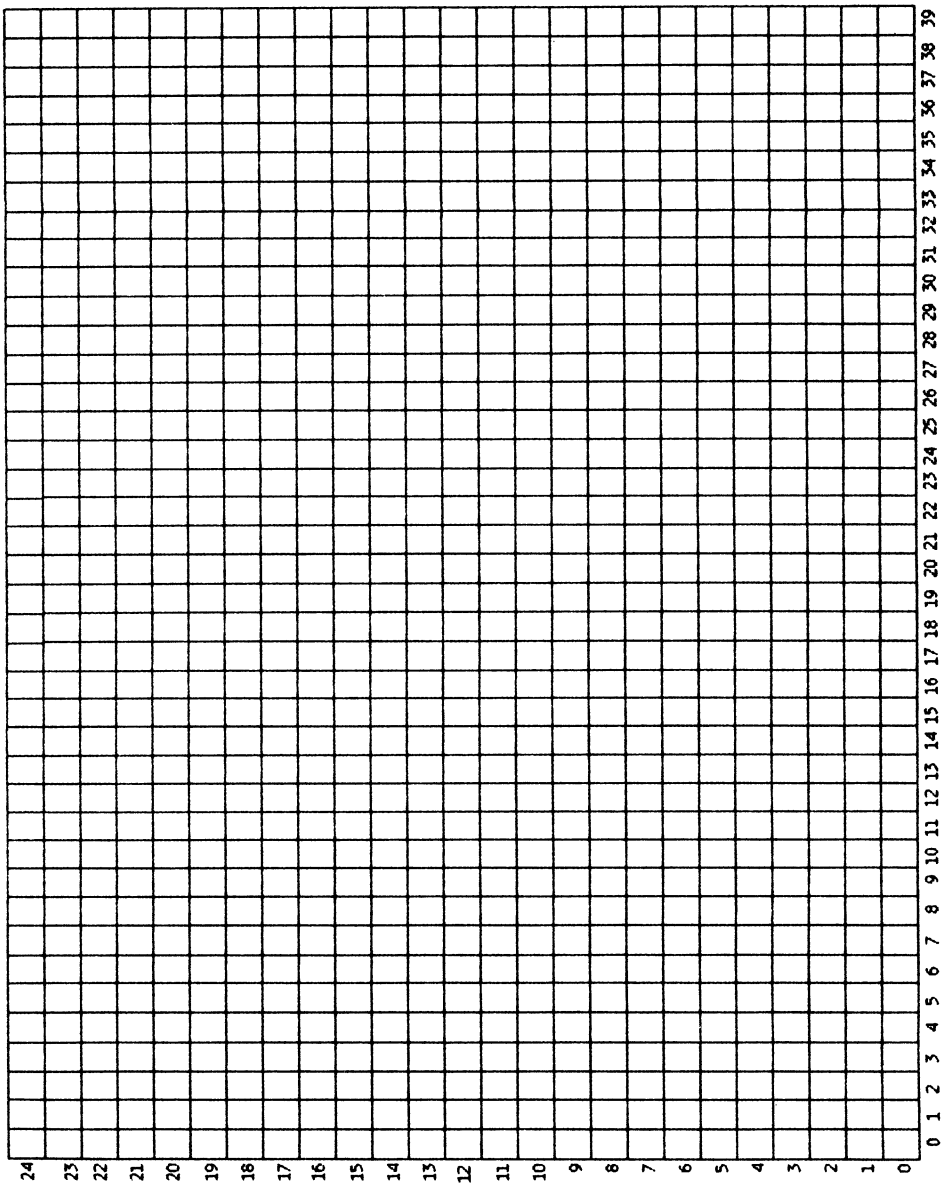


Graphics symbols that draw corners, diagnols, and across

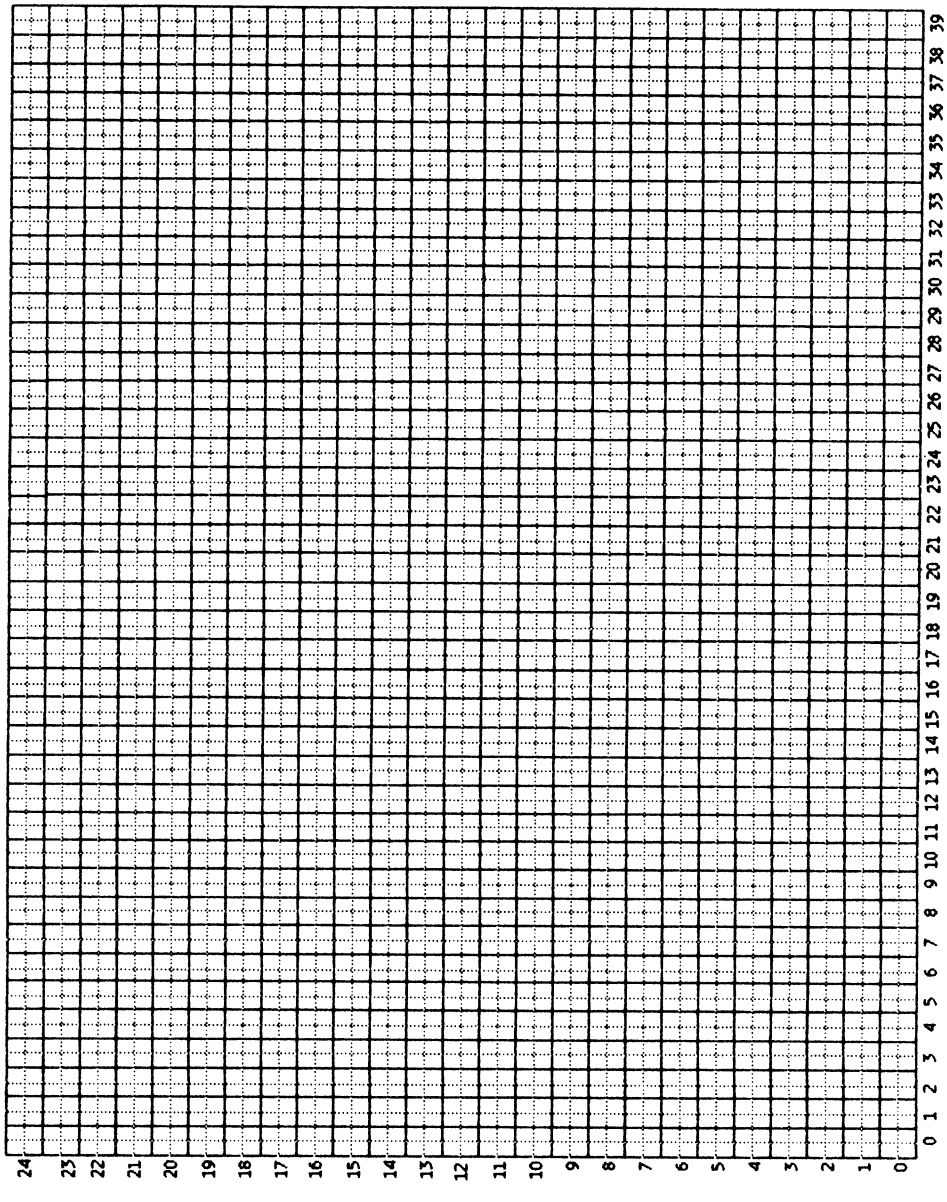


Graphics symbols that draw card symbols, shades areas, and T connectors

APPENDIX C: GRAPHICS WORK SHEETS



Character position chart



Quarter-space character position chart

APPENDIX D: ASCII TABLE

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
	0	CRSR	17	"	34	3	51
	1	RVS ON	18	#	35	4	52
	2	CLR HOME	19	$	36	5	53
	3	INST DEL	20	%	37	6	54
	4		21	&	38	7	55
WHT	5		22	'	39	8	56
	6		23	(	40	9	57
	7		24	)	41	:	58
DISABLES SHIFT C=	8		25	*	42	;	59
ENABLES SHIFT C=	9		26	+	43	<	60
	10		27	,	44	=	61
	11	RED	28	-	45	>	62
	12	CRSR	29	.	46	?	63
RETURN	13	GRN	30	/	47	@	64
SWITCH TO LOWER CASE	14	BLU	31	0	48	A	65
	15	SPACE	32	1	49	B	66
	16	!	33	2	50	C	67
D	68	♠	97	┬	126	Grey 3	155
E	69	▌	98	◥	127	PUR	156
F	70	▬	99		128	CRSR	157

PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$	PRINTS	CHR$
G	71		100	Orange	129		158
H	72		101		130		159
I	73		102		131		160
J	74		103		132		161
K	75		104	f1	133		162
L	76		105	f3	134		163
M	77		106	f5	135		164
N	78		107	f7	136		165
O	79		108	f2	137		166
P	80		109	f4	138		167
Q	81		110	f6	139		168
R	82		111	f8	140		169
S	83		112	SHIFT RETURN	141		170
T	84		113	SWITCH TO UPPER CASE	142		171
U	85		114		143		172
V	86		115	BLK	144		173
W	87		116	CRSR	145		174
X	88		117	RVS OFF	146		175
Y	89		118	CLR HOME	147		176
Z	90		119	INST DEL	148		177
[	91		120	Brown	149		178
£	92		121	Lt. Red	150		179
]	93		122	Grey 1	151		180
↑	94		123	Grey 2	152		181
←	95		124	Lt. Green	153		182
	96		125	Lt. Blue	154		183
	184		186		188		190
	185		187		189		191

CODES	**192-223**	**SAME AS**	**96-127**
CODES	**224-254**	**SAME AS**	**160-190**
CODE	**255**	**SAME AS**	**126**

SOFTWARE FOR YOUR COMMODORE 64

A cassette, containing the program listings in this book, is available from Prentice-Hall, Inc. for $14.95.

To order, mail your check to:

Book Distribution Center
Route 59 at Brook Hill Drive
West Nyack, New York 10995

USING YOUR DISKETTE

For instructions on loading programs from the diskette, turn to pages 17–19 of this book. (Information on cassette use begins on page 16.)

INDEX

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

V

W

COMMODORE 64

COMPUTER GRAPHICS TOOLBOX

Russell L. Schnapp Irvin G. Stafford

Among the most fascinating and exciting uses of home computers is the drawing of figures and pictures. This new book shows you how to create cartoon characters, maps, calendars, geometrical forms, game boards, and game pieces on the Commodore 64 computer. A series of working BASIC programs takes you from drawing flags and hearts through producing complete tic-tac-toe, maze, and Martian-lander games. A turtle graphics program and two editors provide tools for drawing pictures and demonstrate the features of advanced graphics systems. This book lets you learn by doing; it assumes no special background in programming or mathematics.

The book's key features include:

- fully documented listings of all programs with extensive notes, sample results, and suggested modifications;
- grids and demonstration programs that you can use to draw cartoon characters, caricatures, outlines, and figures;
- descriptions of how to generate standard geometrical shapes such as lines, rectangles, and circles;
- complete programs that draw game pieces and boards, throw dice, shuffle and deal playing cards, create mazes, animate a bouncing ball, and simulate a spaceship landing and a rocket launch;
- discussions and examples that illustrate the principles of game design, including keyboard control, player interaction, introduction of random features, scoring, timing, and variations for different skill levels;
- a turtle graphics program that demonstrates the ideas behind LOGO;
- two generalized drawing programs that let you easily create pictures, save them on tape or disk, load them back into the computer, change them, and create geometrical shapes with single key commands.

PRENTICE-HALL, INC., Englewood Cliffs, N.J. 07632

ISBN 0-13-152075-X