

COMMODORE 64 COLOR GRAPHICS: A BEGINNER'S GUIDE

By Shaffer & Shaffer Applied Research & Development





frontispiece

COMMODORE 64 COLOR GRAPHICS: A BEGINNER'S GUIDE

By:

Shaffer & Shaffer Applied Research
& Development

THE BOOK COMPANY

11223 S. HINDRY AVE.
LOS ANGELES, CA 90045

ACKNOWLEDGEMENTS

Special thanks are extended to Penelope Semrau for developing the instructional concepts and graphic designs, to Jeffrey Young for creating the Commodore 64 color graphics tool kit, and to Tamara Sullivan for writing the manuscript. This development team was supported by Lois Augenstein, who coordinated the production of the artwork; Sandra Locke, who produced the artwork; and Andrew Whitman, who tested and edited the manuscript. Thanks also to Kathy Planton and Katherine Harding for typing the manuscript. All of us hope you'll enjoy learning more about your Commodore 64.

Daniel N. Shaffer
President,
Shaffer & Shaffer Applied
Research & Development, Inc.

General Editor
Robert P. Wells, Ph.D.

Editorial Assistant
Elizabeth Anders

Graphics Production
Estela Montesinos
Argelia Navarrete

ISBN 0-912003-06-5

Copyright 1984 © The Book Company, a division of Arrays, Incorporated. All rights reserved. Printed in the United States of America. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording or otherwise, without the prior written permission of The Book Company.

Table of Contents

INTRODUCTION	1
What You Can Expect to Learn	1
Getting Ready	2
How to Use This Book	3
Introduction to High Resolution Graphics	3
Introduction to Your Keyboard	5
CHAPTER 1: SETTING UP THE PROGRAM	9
Entering the Program	10
Running the Program	11
Breakdown of the Program	12
How to "SAVE" a Program	13
Saving on a Disk Drive	14
Saving on a Cassette Tape Recorder	16
CHAPTER 2: BEGINNING GRAPHICS	21
Introduction to Memory	21
Entering the Subroutines	23
Breakdown of the Subroutines	25
Choosing Colors for Your Picture	32
Entering the ZAP Routine	34
Drawing the Lighthouse	36
Breakdown of the Lighthouse Routine	38
Summary	40
CHAPTER 3: FINDING AND PLOTTING POINTS	43
Locating a Point	43
Painting the Background	46
Breakdown of the Paint Background Routine	47
Drawing the Water	49
Breakdown of the Water Routine	51
Summary	56
CHAPTER 4: PLOTTING LINES AND PAINTING SHAPES	59
Designing Foreground Images	59
Plotting Lines	60
Plotting Shapes	64
Painting Shapes	71
Outlining the Land and Waves	82
Painting the Land and Waves	87
More on Colors	91
Shading the Lighthouse	92
Finishing Up the Program	94
Saving a Picture on Disk or Tape	95
Summary	98

CHAPTER 5: TEST PLOTTING AND DUPLICATING SHAPES	99
Defining a Shape with Data Lists	99
Entering Data Lists in the Program	104
Test Plotting	108
Duplicating Shapes	112
Drawing and Placing the Ship's Hull	113
Drawing the Front Sails	120
Drawing the Rear Sails	123
Drawing and Duplicating the Large Seagull	125
Drawing and Duplicating the Small Seagull	127
Design Ideas	128
Summary	131
CHAPTER 6: MAKING AND MOVING SPRITES	135
Introduction to Sprites	135
Special Features of Sprites	141
Drawing and Placing the Sun Sprite	148
Animating the Sun Sprite	165
Design Ideas	169
Summary	177
POSTSCRIPT	183
APPENDIX A: THE PROGRAMMER'S TROUBLE SHOOTING GUIDE	189
APPENDIX B: COMPLETE LISTING OF TOOLS	191
APPENDIX C: ADDITIONAL TOOLS	194
APPENDIX D: SPEEDING UP YOUR TOOLS	200
APPENDIX E: DESIGN CHARTS	205
APPENDIX F: BIBLIOGRAPHY—SUGGESTED READING LIST	207
APPENDIX G: COLOR CHARTS	209
APPENDIX H: TOOL KIT REFERENCE CARD	

INTRODUCTION

Welcome to the world of computer graphics! In this book, you will learn to navigate the Commodore 64 through an ocean of colors. With colors at your command, you can create your very own works of art!

As you may have guessed, the picture shown in the frontispiece was produced using the Commodore 64. The computer program that created the picture is carefully explained step-by-step in the following chapters. After reading this book, you will be able to make the same picture, as well as write programs to create your own pictures on the Commodore 64. The example program and illustrations presented in this book are written for specific use on the Commodore 64. You will learn how to use BASIC programming to instruct the Commodore 64 in drawing various kinds of different lines and shapes. In addition, you will learn how to use an assortment of colors, and how to make shapes move within your pictures.

This book was designed to be read and used by people of all ages. “Computer jargon” has been replaced with plain, everyday English wherever possible. Our only requirement is that you have *some* experience with the BASIC programming language. If words like “program,” “line number,” “GOTO,” and “RUN” are at all familiar sounding, you should have no problem using this book. If, instead, these words make you break out in a cold sweat, you will need to do a little homework. Read chapters 2 through 4 of your *Commodore 64 User's Guide*, spend some time practicing, and you should be prepared to use this book.

What You Can Expect to Learn

In this book you will learn to:

- paint in a background color for any picture
- plot a point
- draw a line
- place lines exactly where you want them
- make shapes
- position a shape at any screen location
- control colors
- make and move small objects

Each of these techniques is a necessary part of drawing any picture on your screen. Because each *is* so necessary, we have arranged the book in a special way. When you finish with Chapter 6, you will have a “tool kit” containing these graphic techniques, each as a separate “tool.” This tool kit can be easily transferred from program to program. Do you need to draw a line? No problem. Just pick up the “draw line” tool, specify where you want the line drawn, and the job is done! (This will be clearer when the first tool is introduced in Chapter 2.)

We will concentrate on teaching *how* a picture can be drawn on the Commodore 64. Often, knowing the *why* is not essential to creating the picture. Think of using your radio. You may not care *why* it works, just *how* it works (where the switch is). Beginning in Chapter 1, any “why” that is not necessary to understand has been separated from the rest of the text and placed in a box. These technical descriptions can be read or passed over, as you please. Passing over the technical descriptions will in no way keep you from learning how to create your own graphic displays.

Getting Ready

To use this book, you will need the following equipment:

1. 1 disk drive (with a blank diskette) *or* a cassette recorder for the Commodore 64 (with a blank tape);
2. A Commodore 64 keyboard;
3. A color video monitor or color TV screen; and
4. Some graph paper to work out your own designs.

Each time you sit down to work with this book you should be at your computer. You will need a monitor (such as a TV) which is properly connected to the Commodore. Instructions on setting up your equipment are provided in the User's Guide that came with your Commodore. It also explains how to connect a cassette recorder to the Commodore, if you plan to use one.

If you own a disk drive, it should be connected to the Commodore. All disk drives come with an instruction manual on how to set them up properly. Check your manual if you need help.

Once you have your equipment in order, do the following:

1. Turn on the disk drive, *if* you are using one. (Note: cassette recorders are automatically “on” whenever they are connected to the computer.)
2. Turn on the Commodore 64.
3. Turn on the TV (or whatever monitor you are using).

Your screen should say “**** COMMODORE 64 BASIC V2 ****” at the top. Below this, a line describing your computer's memory will be shown. (Don't worry if you don't know what “memory” means.) Finally, you will see “READY.”, followed on the next line by a blinking box. This blinking box is called a *cursor*, and it serves as a place-marker on the screen. Each time you type a letter or symbol, the letter or symbol will be displayed on the screen at the cursor's location. The cursor then moves to the right one space to mark where the next typed letter or symbol will be displayed. You will see this cursor many more times as you enter the program that draws your ship.

How to Use This Book

Beginning in Chapter 2, the best way to use this book is to:

1. Load the previous chapter's program into the computer.
2. Read the introduction to the new chapter.
3. Type in the BASIC lines when requested in the text.
4. "RUN" the program and watch what happens on your TV screen.
5. "LIST" the program and correct any typing errors.
6. Read the chapter pages that explain what is happening.
7. "SAVE" your program with a new name at the end of a session or chapter.

By following through each chapter, you will develop a better understanding of what goes into making a computer-generated picture. We suggest that, as you read through a chapter, you "RUN" the program several times to see how the computer is using it to create the visual display on your TV. The chapters are arranged so that each chapter builds upon the previous one. This allows you to start off easy, and gradually work your way up to the final picture.

In the back of this book are several Appendices providing additional information about Commodore 64 graphics. Included you will find a trouble-shooting guide to help locate program errors, and a recommended reading list of Commodore 64-related writings. Chapter 6 discusses the contents of the Appendices in more detail.

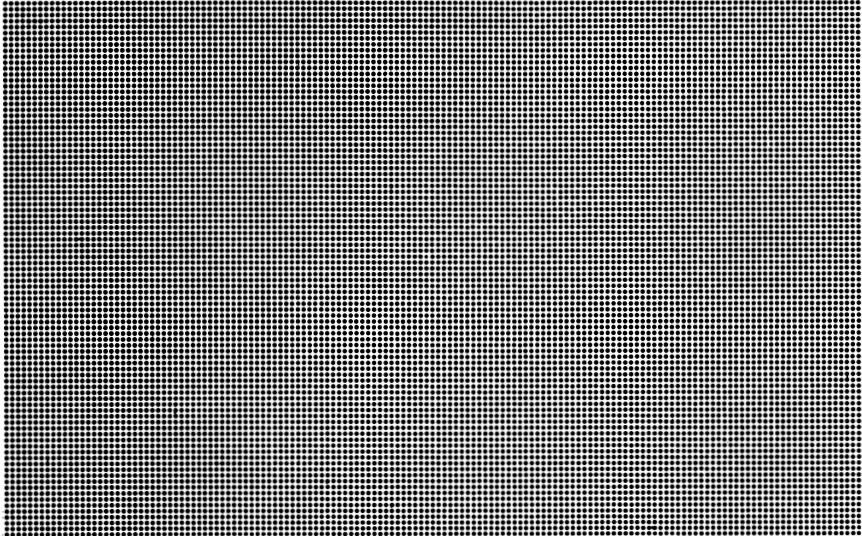
Introduction to High Resolution Graphics

Most of the software products that you see today (such as computer games, graphing programs, and educational products) make use of sophisticated colored pictures. Computer pictures like these are made up of thousands of tiny dots of colored light. These colored dots are called *pixels*. Each Commodore 64 picture is made up of a total of 64,000 pixels. There are 320 pixels across each row, and 200 down each column. The pixels are very small and close to each other. These thousands of pixels, viewed together, look like a solid picture when you are at a distance from the TV.

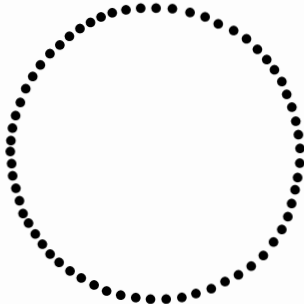
The total number of pixels a computer uses to display a picture on the screen is called the *resolution*. The quality of a computer picture varies from one computer to another because some computers can display pictures in a resolution of 64,000 pixels, while other computers may display pictures with considerably less, say 1,600 pixels per picture.

The higher the number of pixels, the higher the resolution. This is called *high resolution*. All of the graphic programs presented in this book will be displayed in high resolution. Your high resolution screen will appear with rows and columns of pixels similar to this:

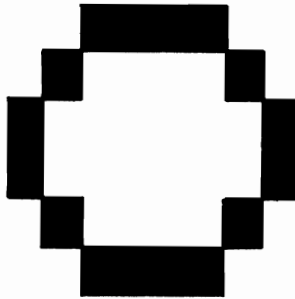
HIGH RESOLUTION SCREEN



Pictures made on a screen using high resolution are termed *high resolution graphics*. The resolution of the screen plays an important part in making pictures: the higher the resolution, the more detail the picture can have. For example, if you are making a circle in high resolution, your circle might look like this:



Whereas, if you were working on another computer system in low resolution, your circle might look like this:



In low resolution, the TV screen has fewer pixels. To fill the screen with fewer pixels, the computer has to use *larger* pixels. This makes objects, like circles, appear block-like and chunky when compared to the refined, smooth look of high resolution pictures.

Whether you are creating bar graphs for business use or a map for geography class, you will appreciate working in high resolution graphics on your Commodore 64.

Introduction to Your Keyboard

If you have already used your Commodore, you probably know how some of the special keys are used. For example, some keys allow you to change the color of the letters and symbols that are typed.

To see how this works, let's begin by changing the typing color to yellow. To do this, press the **CTRL** key (located on the left-hand side of the keyboard) and, *at the same time*, press the key numbered **8** (located on the top row of the keyboard). Then release both keys.

NOTE: Throughout this book, we will boldface the name of the keys on the keyboard. So, for example, when we say "press **CTRL**," we mean *press* the key marked **CTRL**. We do *not* mean to press each letter (**C**, **T**, **R**, and **L**).



To see what happened, type in your name. It should be printed in yellow. Press the **CTRL** key while pressing the **3** key. Release both keys and type another word—any word. What color are the letters now? Try out some other colors (**CTRL** **6**, release), and then type in some more words and characters. Adjust your TV screen for a clear picture.

This is just one way to direct the computer in the use of colors. Another unique color feature of the Commodore 64 is its use of the special "Commodore key" (**C=**). This key produces a second set of character colors. On the lower left-hand side of the keyboard is the key marked **C=**. Press both the **C=** key and the **5** key at the same time.

Release both keys and type in some more characters. You are now using the second color set. Press C= 6 and type the word "commodore." Take some time and try out the other color options.

Below is a chart illustrating 16 different colors. To use a color in the first (upper) set, such as red, you would press **CTRL** and **3**. To use a second (lower) set color, like light green, you would press **C=** and **6**.

Character Color Chart								
first set								
	Black	White	Red	Cyan	Purple	Green	Blue	Yellow
[CTRL]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
[C=]	Orange	Brown	Light Red	Grey 1	Grey 2	Light Green	Light Blue	Grey 3
second set								

Looking at your keyboard you will find a set of graphic characters on the front of each alphabetic key. For example, on the front of the **N** key are the two graphic characters:  on the right, and  on the left. To display the graphic character on the right side, hold down the **SHIFT** key and press the **N** key. The graphic character on the left is obtained by pressing **C=** while pressing the **N** key. Try using these graphic characters and others in different colors. Although these characters are good for some graphic examples, they are too limited for the graphic programs presented in this book and so we will not use them.

To clear the screen, hold down the **SHIFT** key and press the **CLR/HOME** ("clear/home") key on the upper right-hand side of the keyboard. This will clear the screen and place the little blinking light (the cursor) in the upper left-hand corner of the screen. This upper left-hand corner position is called the **HOME** position.

Another special key that you will need to use often is the **RUN/STOP** key (on the left-hand side of the keyboard). If you are running a program, as we will in the next chapter, holding down the **RUN/STOP** key *stops* the program. This key will be very useful to you. For instance, if a program is not doing what you expect it to do, press the **RUN/STOP** key. You can then go back and check the program for typing errors. We will remind you about this key again later, when you run your first program.

Finally, there will be times when pressing **RUN/STOP** won't get you back to your program listing. If you have entered a program wrong, and the **RUN/STOP**

key does not seem to help: hold down the **RUN/STOP** key and *tap* the **RESTORE** key. This should get you back to your program listing. If not, a *last* resort is to turn the computer off and on again. The problem with this method is that any portion of your program that has not been saved on a diskette or tape will be “forgotten” (erased) by the computer.

Below is a chart that reviews some of the special keys discussed so far:

Key	Description of Use
[CTRL] [8]	set the color of the typed characters to yellow
[C=] [6]	sets the color of the typed characters to light green
[SHIFT] [N]	types the graphic character shown on the bottom right side of the “N” key
[C=]	types the graphic character shown on the bottom left side of the “N” key
[SHIFT] [CLR/HOME]	clears screen and places cursor at home position
[RUN/STOP]	stops a program when it is running, returning you to the text screen; anything that was displayed on the text screen prior to running the program will again be displayed, as well as any syntax errors found during the execution of the program.
[RUN/STOP] [RESTORE]	stops a program when it is running, returning you to the text screen; the text screen will be cleared and the cursor will be placed in the home position

These keys and others are used in the upcoming chapters to enter your program. Before going on to Chapter 1, take some time to look over your keyboard. You should note the location of any keys you have not see before. Especially note the location of the **RETURN** key, as this is one of the most often used keys.

Chapter 1

SETTING UP THE PROGRAM

Each chapter in this book covers a portion of the program needed to draw your ship. A “program” is a list of instructions that tells the computer what to do. Each instruction (or line) in the program is given a number. The line numbers tell the computer the order in which it should carry out the instructions. Line numbers can start at 0, and continue up to as high as 63999.

The instructions within each line are made up of various letters and symbols. There are many different ways to write instructions to the computer. Each method is called a computer *language*. You will be using a language called BASIC. With it, you will learn to instruct the Commodore 64 to draw lines and shapes, change colors, and even make objects move.

This chapter’s program has two goals: (1) to establish the organization of the final program; and (2) to completely familiarize you with “subroutines.” In general, a program can be divided into two sections: the main routine, and the subroutines. Subroutines can be thought of as “tools” that the main routine uses to do various tasks. Consider the tools used to plant seeds, such as a hoe, a spade, or a shovel. Each of these tools serves a different purpose. A hoe is used for one task, while a spade is used for another. When a tool is no longer needed, it is set back down until needed again. In the meantime, the gardener returns to the general task of planting seeds.

Subroutines are similar in that they are set in a specific place, and used only when called for in the main routine. After performing their tasks, they are “set back down” and the computer returns to the main routine. Because subroutines are used many times, they are grouped together. This way, they are easily found the next time one is needed.

The computer can be thought of as the gardener. It has the responsibility of getting the main routine (the “planting of the seeds”) done, using whatever tools are available. To “pick up” a subroutine in the middle of the main routine, a special programming code is used: GOSUB. GOSUB is always followed by a number, such as GOSUB 20. When the computer comes across such a statement, it knows to GO directly to a SUBroutine. The number tells the computer which line number the subroutine begins on. The computer stops performing the main routine instructions and begins performing the subroutine. To let the computer know when the subroutine task has been completed, another special code is used: RETURN. RETURN tells the computer to stop using the subroutine lines and return to the main routine.

Where does the computer return to in the main routine? This is the beauty, and the importance, of subroutines. Whenever the computer has completed a subroutine task, it returns to the *exact* same place it was before it went to the subroutine.

To give you an example, suppose a portion of the program read:

```
1100 GOSUB 20
1101 PRINT "SUBROUTINE COMPLETED"
```

When the computer came across line 1100, it would immediately go to line 20 to begin performing the subroutine there. When it came across a RETURN statement, it would return to the location just after GOSUB 20. This would be line 1101, which tells the computer to print SUBROUTINE COMPLETED. The commands GOSUB and RETURN allow you to “pick up” a subroutine tool, use it, and then return to the same place in the main routine.

Now, how should you organize a program? Since subroutines need to be set apart from the main routine for easy access, they are assigned their own set of line numbers. This is usually done by setting aside 1,000 to 2,000 line numbers to be used only by subroutines. The final program in this book will contain 23 subroutines. To make sure we leave enough room to enter all of them, we will set aside 1,000 line numbers to be used only by our subroutines. To make sure we leave enough room for the main routine, the program’s organization is:

- (a) lines 1 through 999 will only be used for the subroutines;
- (b) lines 1000 and up will only be used for the main routine.

This setup allows the main routine to become as large as necessary. Remember, the main routine is responsible for the majority of the work. Thus, it usually requires more lines of instructions.

Entering the Program

To see how this works, let’s enter the first program, a simple one to start with. This program has the computer print a message on the screen whenever one of 3 “special” keys is pressed. If any other key is pressed, the computer will do nothing. Sound easy? It is.

Begin first by pressing CTRL and 2 at the same time. You remember what this does. It changes the typing color to white, which is easier to see on the screen than the normal color of blue.

Now, type the program shown below on your keyboard. Do *not* forget the line numbers. They tell the computer the order the instruction lines are to be carried out. Type the program carefully, leaving nothing out and adding nothing to it. A slight change in the program could prevent it from working properly. We have indented (pushed to the right) the program to make it easy to see. Do *not* indent the program as you type it. Begin on the left side of your screen, type line 1 exactly as we have it listed, and press RETURN. Then go on to type line number 20.

```
1 GOTO 1000
20 REM:::::::::::GRAPHICS
21 PRINT"SUBROUTINE G WORKS"
```

```

24 RETURN
30 REM:::::::::::::TEXT
31 PRINT"SUBROUTINE T WORKS"
34 RETURN
40 REM:::::::::::::COLORS
41 PRINT"SUBROUTINE C WORKS"
44 RETURN
1000 REM:::::::::::::
1001 REM    MAIN ROUTINE
1002 REM:::::::::::::
1100 GET A$
1110 IF A$ = "G" THEN GOSUB 20
1120 IF A$ = "T" THEN GOSUB 30
1130 IF A$ = "C" THEN GOSUB 40
1140 GOTO 1100

```

Correcting Mistakes:

If you make a typing mistake *before* you press **RETURN**, press the **INST/DEL** key to back up and erase characters (the **INST/DEL** key is on the top, right side of your keyboard).

If you make a mistake *after* you press **RETURN**, just re-type the whole line (including the line number). It will look like you have two lines with the same number, but if you type **LIST** and press **RETURN**, you will see that the corrected line has been inserted in your program.

If you want to erase an entire line (for example, one that was given an incorrect line number), re-type the number of the line to erase, and press **RETURN**. Type **LIST**, press **RETURN**, and you will see that the line has been erased from your program.

Look at the program. You will notice that it contains three subroutines. Each subroutine is located in our 1 to 999 line number group. The main routine begins on line number 1000. The main routine contains the **GOSUB** commands to use the subroutines. The subroutines contain the **RETURN** commands to get back to the main routine.

Running the Program

When you have the program typed on your screen, you can "RUN" it. **RUN** is a command you type when you want the computer to use your program. This command tells the computer to start at the lowest line number in the program (in your program this is line 1), and begin doing what the program tells it to do. Let's try it out. Type the word **RUN** and press **RETURN**.

You will find out why this program works in a moment. For now, we will tell you what to expect and what you should do. If your program does not work as

described below, press the **RUN/STOP** key. After that, type LIST and press **RETURN**. Your program will again be listed on the screen, and you should check for typing errors. Common errors include lines that were omitted, line numbers typed wrong, quotation marks left out, small L's (l) typed instead of ones (1), and "ohs" (O) typed instead of zeroes (0). Re-type any lines that have mistakes, and RUN the program again.

When the program is running, the first thing you should notice is that the cursor disappears. *Nothing else changes*. The computer is waiting for you to press a key. Remember, it is looking for one of 3 special keys. Press the key numbered 5. If you entered the program correctly, nothing will happen. Try H. Again, nothing. Now try T.

The computer should respond by printing "SUBROUTINE T WORKS" on the screen. Press G and C. G, T, and C are the special keys that cause the computer to print a message. If you recall, these are the keys you typed between quotes in your program. Try pressing these and other keys as many times as you like. You will quickly find that only these 3 keys provide you with a message.

What makes this work? To find out, press the **RUN/STOP** key to stop the program. "READY.", along with the cursor, will reappear on the screen. To see the program again, type LIST and press **RETURN**.

Breakdown of the Program

The first line in the program tells the computer to skip over the subroutines and GOTO (go to) line 1000. If this line were omitted, we would not be able to put the subroutines before the main routine.

Lines 1000 through 1002 are "remark" lines. "Remark" lines begin with the word REM and are *always* ignored by the computer. Their function is to label or title various sections of the program. Notice that these lines label the MAIN ROUTINE section of your program. REM lines are there only for *you* to see, and will help you remember what the different parts of the program are for.

Next, the computer comes to line 1100, which contains GET A\$. This is like telling the computer to go get A\$ and "find out" what it is. The computer looks to the keyboard. It is looking for a single keystroke. *Any* key that is pressed will be noticed by the computer and placed in the "space" called A\$. Think of A\$ as a place-holder that is eventually filled with *whatever* letter, number, or symbol you type on the keyboard. Once you type a key, the computer has found out what A\$ is, and so it can continue on to the next line number.

Lines 1110 through 1130 tell the computer what to do now that it knows what A\$ is. These type of lines are called IF...THEN statements. They tell the computer "IF such and such is true, THEN do this." Line 1110 says:

```
IF A$ = "G" THEN GOSUB 20
```

The computer reads this as “IF A\$ is the letter G, THEN GO to the SUBroutine at line 20.” Lines 1120 and 1130 tell the computer what to do if A\$ is T or C. They also direct the computer to a subroutine. If A\$ is not G, T, or C, the computer just continues to line 1140. Line 1140 directs it back to line 1100 to get A\$ *again*. The next key pressed then becomes A\$, and the old key that was pressed is forgotten.

Finally, let’s look at the subroutines. Suppose the program is running and the computer comes to line 1100 GET A\$. You press T on the keyboard. The computer then knows that A\$ is T. Next, it goes to line 1110. Since A\$ is not G, it skips the instructions in this line and continues on again. It reaches line 1120. Since A\$ is T, it follows the instructions in this line to GO to the SUBroutine at line 30.

Line 30 is another title (REM) line, so the computer ignores it. Line 31 gives it a PRINT instruction. It says to print “SUBROUTINE T WORKS.” Whatever is in quotes after the word PRINT will be printed on the screen. This is why your computer printed SUBROUTINE T WORKS every time you pressed the key marked T. Once the message is printed, the computer reads line 34. Line 34 contains the necessary RETURN command to get back to the main routine. Knowing that the subroutine task has now been completed, the computer returns to the main routine.

Remember, the computer goes back to the last place it left off. Since it had last completed line 1120, it goes directly to line 1130. A\$ is still T, so line 1130 is read and passed over. Line 1140 tells it to GOTO line 1100, and the process is started all over again.

It is important that you understand these common principles of BASIC. You will use many subroutines, GOTO statements, GET statements, and PRINT statements in the chapters to come.

Try running your program again (type RUN and press RETURN). As you do, read through the program’s breakdown above. Note how the computer is using your program. If you are having any problems, check the program listing for typing errors. Correct lines by re-typing them.

When you are through, press RUN/STOP to stop the program. Type LIST and press RETURN to see the program again. Your final task in this chapter will be to “save” your program.

How to “SAVE” A Program

Once a program is running properly, you should save it on your disk or cassette tape. What does it mean to “save” a program, and why is it necessary?

When you first type a program, it is stored in the computer’s “memory.” The computer’s memory is a temporary storage place inside the computer that holds any information you type. It is temporary, because as soon as the computer is turned off, the information in memory is wiped out. This includes any program you had typed in. Your disk drive or cassette recorder solves this problem by

making a permanent “recording” of your programs. Each recording can be played over and over again, as many times as you like.

There are different ways to SAVE a program, depending on whether you are using a disk drive or cassette tape recorder. Before you actually begin saving a program, it is a good idea to place your disk drive or cassette recorder as far away from the TV as possible. This is because the TV can interfere with the saving process, and your programs may not be saved correctly.

If you are using diskettes and a disk drive, then continue to read the next section, “Saving On a Disk Drive.” Those of you using a cassette tape recorder should skip to the section entitled “Saving On a Cassette Tape Recorder.”

Saving On a Disk Drive

To begin, make sure your disk drive is connected to the Commodore 64. Place a diskette in the disk drive. To save Chapter 1’s program, type:

```
SAVE “CHAPTER 1”,8
```

Be sure to type CHAPTER 1 in between quotation marks, and do *not* add any extra spaces within the quotes. After you have typed this instruction line, press the **RETURN** key. The **RETURN** key tells the computer that you are finished typing the command.

NOTE: You will need to press **RETURN** after every instruction line in the program listing and after every command. Every now and then we will remind you about the **RETURN** key. Most of the time, however, you will need to remember yourself.

The computer then begins to save the program. You will know that the computer has saved the program when the disk drive’s red “in use” light goes out.

SAVE is the command that puts your program on the diskette. This command is always followed by the program’s filename (title) in *quotation marks*. You may title your programs with any special name you like, providing the name has no more than 16 characters. After you have typed the filename, you type a comma, followed by the number 8. 8 tells the computer you are using a disk drive (and not a cassette recorder).

Note that each new program needs to be saved with its own filename. Two programs cannot have the same filename. However, there will be times when you need to erase an old program and save a new one under the old filename. An example is when you update a program. To *erase and replace* a program on your disk, type:

```
SAVE “@0:filename”,8
```

where *filename* is the name of the old program you want to erase and replace.

Verifying it was Saved:

Whenever you save a program; you should then doublecheck and make *sure* it was saved. To do this, you use another command: **VERIFY**. This command verifies that the program is safely stored on the disk. To check Chapter 1's program, type:

```
VERIFY "CHAPTER 1",8
```

Notice that the only difference between the **SAVE** command and the **VERIFY** command is the first word (**SAVE** or **VERIFY**). "CHAPTER 1",8 is not changed. The computer should respond with **SEARCHING FOR CHAPTER 1 VERIFYING OK**. If not, try entering the **VERIFY** command again. If you still do not get the proper response, then the program did not get placed on the disk. **LIST** your program to make sure you still have it in memory (if you don't, you will need to re-type it) and then **SAVE** it again. Under most circumstances, if the computer can't find your program, you probably made a typing mistake in either the **VERIFY** command or the **SAVE** command.

Loading a Program off the Tape

Now that the program is stored on a disk, you can take a break. Make *sure* the red "in use" light on the disk drive is off, remove the disk, and turn the computer off. The program will be erased from *memory*, but it will remain in place on the disk.

When you turn on the computer and want to put the program back into memory, you **LOAD** it off the disk. The important thing to understand about loading a program is that it is *not* removed from the disk and placed in memory. Instead, it is merely duplicated in memory. Thus, loading a program places a copy of it in memory *and* leaves the original copy on the disk.

LOAD is the command you use to get a specific program off of a disk. If the computer already has a program in memory, **LOADing** another program will erase the first one from memory. To use the **LOAD** command, type:

```
LOAD "CHAPTER 1",8
```

You may not see the program, but it *is* in memory. If you want to see it, you have to **LIST** it. You have already used the **LIST** command several times. Just type **LIST** and press **RETURN**.

Viewing the list of filenames:

If you need to look at a program, but have forgotten what you've named it, you can have the computer list all filenames on the disk. The list of filenames is like a table of contents; it tells you what programs are stored on the disk. A note of warning about listing your filenames: the computer's memory will be *erased* and replaced with the list of filenames. In other words, never list your filenames until you have saved any program you are working on. Otherwise, the unsaved program will be erased.

To see the list of filenames, type:

LOAD "\$",8

It is not simple to explain why **LOAD "\$",8** places the filenames in memory. Trust us, though, it does. At this point, the **LIST** command will list the filenames rather than a program. Try out this command to see how it works.

Summary:

SAVE, **VERIFY**, **LOAD**, and **LIST** are very important commands to remember. At the end of each chapter, you will be told to save your program. Chapter 2's program is saved by typing: **SAVE "CHAPTER 2",8**. Chapter 3's program is saved by typing : **SAVE "CHAPTER 3",8**. And so on. Each time you **SAVE** a program, you should **VERIFY** that it was placed on the disk.

At the beginning of each chapter, you are instructed to **LOAD** the previous chapter's program. If you need help, turn back to this section as a reminder on how these commands work. Be especially careful to type the filenames correctly (for example, "**CHAPTER 1**"). If you were to type something like **SAVE "CAHPTER 1",8**, you could only load Chapter 1's program back into memory by misspelling the name again (**LOAD "CAHPTER 1",8**).

You need to take care of your diskettes by keeping them out of the sun and away from heat, by not bending, folding, or scratching them, and by storing them away from the TV or any metallic object.

From now on, the procedure you should follow at the end of each chapter is:

- (1) **SAVE "Chapter X",8** (replacing X with the current chapter number).
- (2) Continue to the next chapter.
- (3) **LOAD "Chapter X",8** (replacing X with the previous chapter number).
- (4) **LIST** the program.
- (5) Begin reading the new chapter.

Since you have already saved your program, and the next section is for saving on a cassette tape recorder, skip over it to Chapter 2.

Saving On a Cassette Tape Recorder

To "**SAVE**" your program permanently on cassette tape, make sure you are using a tape that can be erased (preferably a new one). Saving your programs on tape is just like recording music on tape. If you record a song over a previous recording, the new song erases the previous song.

Because you can not listen to a program to see where it begins and ends on the tape, you need a special type of cassette recorder. This recorder has a "counter" on it to tell you where you are. The counter is located on top of the cassette player, and is easily identified by the label **COUNTER**. With the help of this counter, you can

keep track of which sections of tape contain a previously recorded program. Let's discuss how this works.

The first thing you need to do when saving a program is to rewind the tape. Next, you set the counter to 000 by pressing the small button next to it. What this does is establish the beginning of the tape and the beginning of the counter's numbers. Nothing has been recorded on the tape yet, so the counter is set to display the number 000.

You need to write down the "filename" and "beginning number" of the program you are saving. The filename is just a title that you give the program so that the computer can later find it easily. The beginning number is the number displayed on the counter just before you save the program. 000 is the beginning number of the first program saved on any tape.

As a program is being saved, the counter begins to turn. As it turns, it continually displays a higher number. When the program has been saved, the counter number displays the "ending number" of the program stored on the tape. You should also write this number down on a piece of paper.

The next time you save a program, you would again start by rewinding the tape and setting the counter to 000. After that, you would press "Fast Forward". When the counter shows a number that is higher than the ending number of the last program saved, press STOP. You will then know that you have passed over any sections of tape that already contain recorded programs.

To save Chapter 1's program, start by writing down the following column headers on a scratch piece of paper:

PROGRAM FILENAME STARTING NUMBER ENDING NUMBER

This piece of paper will be used to write down the location of each program on the tape. If you know where a program is located, you can avoid saving another program on top of it. For this chapter's program, write down CHAPTER 1 under "program filename," and 000 under "starting-number." You will find out the "ending number" in a moment.

To save Chapter 1's program, type:

SAVE "CHAPTER 1",1

Be sure to type CHAPTER 1 in between quotation marks and do *not* add any extra spaces inside the quotes. After you have typed this instruction line, press the RETURN key. The RETURN key tells the computer that you have finished typing in the command.

NOTE: You will need to press RETURN after every instruction line in the program listing and after every command. Every now and then we will remind you about the RETURN key. Most of the time, however, you will need to remember yourself.

The screen then displays a message telling you to press PLAY and RECORD. Make sure your cassette player is as far from the TV as possible. Then, press PLAY

and RECORD on your cassette recorder at exactly the same time.

The computer will begin to save the program. As it does, your screen will be cleared of any information. You will know that the computer has saved the program when the text on your screen reappears. When this happens, press the STOP button on the recorder. Look at the counter number. It will show the ENDING NUMBER of the program, which you should write down on your scratch paper. You now know the beginning and ending points of your CHAPTER 1 program. The next time you save a program, you can be sure to save it somewhere past the ending number of this program. Usually, the next program is saved beginning about 10 counts higher than the last program's ending number.

SAVE is the command that puts your program on the tape. This command is always followed by the program's filename (title) in *quotation marks*. You may title your program with any special name you like, providing the name is no longer than 16 characters. After you have typed the filename, type a comma, followed by 1. 1 tells the computer you are using a cassette recorder (and not a disk drive).

Never save a revised program on top of its original version *unless* the original was the last program stored on the tape. A revised program will usually be a different length than the original. If it were longer, you might tape over part of the next program on the tape. If it were shorter, part of the original would never get taped over, and would be "tacked on" to the end of the revised program.

The best place to save a revised program is right after the last program on the tape. On tape, you can use the same filename as many times you like, so the revised program can have the same filename as the original. You should, however, note on your handwritten list which one is the revised version.

Verifying it was saved:

Whenever you save a program, you should then double-check to make *sure* it was saved. To do this, you use another command: VERIFY. This command verifies that the program is safely stored on the tape. The important thing to know about verifying a program is that the tape must be rewound to a point somewhere before the program's starting number. To do this, rewind the tape, set the counter to 000, and "Fast Forward" (if necessary) to a spot just a little before the program begins. If the program begins at counter number 050, fast forward to 045. If the program begins at 000 (as does the CHAPTER 1 program), do *not* Fast Forward at all.

To check Chapter 1's program, rewind the tape, set the counter to 000, and type:

```
VERIFY "CHAPTER 1",1
```

This time, the message on your screen directs you only to press the PLAY button. Press PLAY and your screen will again go blank. After a moment, the computer should respond with FOUND CHAPTER 1. This lets you know that it has found the program that you want to verify. To have the computer continue with the verification, press the C= key. When the computer has finished verifying the program, the cursor and text will reappear on your screen. Press STOP on your cassette player.

If the computer could not find your program, check to make sure you rewound the tape far enough. Try entering the **VERIFY** command again. If you still do not get the proper response, then the program did not get placed on the tape. **LIST** your program to make sure you still have it in memory (if you don't, you will need to re-type it) and then go through the procedures to **SAVE** it again. Under most circumstances, if the computer can't find your program, you probably did not rewind the tape far enough, or you made a typing mistake in either the **SAVE** command or the **VERIFY** command.

Loading a program off the tape:

Now that your program is stored on the tape, you can take a break and turn the computer off. The program will be erased from *memory*, but it will remain in place on the tape.

When you turn the computer on again and want to put the program back into memory, you load it off the tape. There are two important things to know about loading a program. First, loading a program does *not* remove it from the tape and place it in memory. Instead, it merely duplicates the taped program in memory. Thus, loading a program places a copy of it in memory *and* leaves the original copy on the tape.

The second thing to remember is that the tape must again be rewound to a point somewhere before the program begins. This is done in the usual manner. Rewind the tape, set the counter to 000, and "Fast Forward" (if necessary) to a point just before the program begins. If the program begins at 000, do *not* press the Fast Forward button.

LOAD is the command you type to get a specific program off of a tape. If the computer already has a program in memory, **LOADing** another program will erase the first one from memory. To use the **LOAD** command, rewind your tape, set the counter to 000, and type:

```
LOAD "CHAPTER 1",1
```

The **LOAD** command is very similar to the **VERIFY** command. When instructed on the screen, press **PLAY** on your cassette recorder. After the program to load has been found, press **C=** to have the computer continue with the loading process.

You may not see the program, but it *is* in memory. If you want to see it, you have to **LIST** it. You have already used the **LIST** command several times. Just type **LIST** and press **RETURN**.

Be sure to write down every program stored on your tapes, and keep your lists in a safe place. There is no way to have the computer list all of the filenames on a tape. If you label your tapes and handwritten lists, though, you will always know which list goes with which tape.

Summary:

SAVE, VERIFY, LOAD, and LIST are very important commands to remember. At the end of each chapter, you will be told to save your program. Chapter 2's program must be saved on the tape *past* Chapter 1's program. The filename, beginning counter number, and ending counter number should always be written down for each program. To save Chapter 2's program, type: SAVE "CHAPTER 2",1. Chapter 3's program is saved by typing: SAVE "CHAPTER 3",1. And so on. Each time you SAVE a program, you should VERIFY that it was placed on the tape. To VERIFY a program, the tape must be rewound to a point before the program begins. This is also necessary any time you want to LOAD a program.

From now on, the procedure you should follow at the end of each chapter is:

- (1) Make a note of the counter number under STARTING NUMBER.
- (2) Type SAVE "CHAPTER X",1 (replacing X with the current chapter number).
- (3) Make a note of the PROGRAM FILENAME (this will be the filename typed as Chapter X in step 2 above).
- (4) Make a note of the counter number after the program has been saved.
- (5) Continue to the next chapter.
- (6) Rewind the tape to the starting counter number of the last chapter's program.
- (7) LOAD "Chapter X",1 (replacing X with the previous chapter number).
- (8) LIST the program.
- (9) Begin reading the new chapter.

If you later forget how these commands work, refer to this section for help.

Chapter 2

BEGINNING GRAPHICS

This chapter is the first step toward actually recreating the picture of the ship. You will use the program entered in Chapter 1, but in this chapter you will replace the PRINT statements with actual graphic subroutines. After you change them, the first subroutine will turn on high resolution graphics; the second subroutine will turn off high resolution graphics; and the third subroutine will let you change the colors displayed in high resolution graphics.

These three subroutines are the first tools in your graphics "tool kit." Notice that each of them performs a task that is helpful when drawing *any* picture. After you have entered the new subroutines and have checked to make sure they work properly, we will discuss the new concepts they introduce. Finally, we will add another tool to our tool kit. This tool, called the ZAP routine, will erase (ZAP!) the MAIN ROUTINE in your program. The purpose of the ZAP routine is to allow you to delete the existing main routine, so that you can begin entering a new main routine which draws a different picture. The ZAP routine always leaves your subroutine tools in place for use with the new picture. To illustrate this point, you will zap your current main routine and replace it with a new one: a main routine which draws a lighthouse.

The focus of this chapter is on the computer's memory. Each time you draw a picture, the picture's pattern (dots and lines) and the picture's colors are stored in the computer's memory. You will learn how to store your patterns and colors in specific locations within memory. This is a necessary part of picture-drawing on the Commodore 64. The next section briefly introduces how the computer's memory works. If you are already familiar with using memory, you may want to skip over this section.

Introduction to Memory

The computer's memory is made up of many small "boxes," each capable of storing information, and each having its own unique number. To get a better idea of this, think of memory like the many mailboxes that line residential streets. Each mailbox is capable of storing information (mail), and each has its own unique number (address). In all, your Commodore computer has over 64,000 "mailboxes" for storing information. Each 1,000 mailboxes is called "1K" of memory. Thus, your Commodore 64 has 64K of memory.

A mailcarrier always begins a route in the same place everyday, and proceeds to drop letters in each mailbox in turn. If a letter has not been sent to one address, the carrier skips over that mailbox and continues to the next one. The computer uses its memory in a very similar manner. As soon as you turn the computer on and start typing, the computer not only displays your typing on the screen, it also places the

letters and symbols you type into a specific set of memory boxes. This is so it can find them again later when you run the program or list it. The computer always starts at the same memory box to store the text you type. As you type, it places the letters and symbols into each box it comes to. If you leave one or more empty spaces on your screen, the computer leaves one or more memory boxes empty.

When the mailcarrier is through for the day, the letters are removed from the mailboxes by their owners, and the next day the mailcarrier begins the route again. This is how the computer works when you turn it off and on again. When it is turned off, the letters you typed are removed (erased) from the memory boxes. (This is why it is so important to save your programs on a disk or on tape.) When the computer is turned back on again, it starts at the same beginning box and drops your new letters, symbols, and spaces into each box it comes to.

Now, the mailcarrier can both put letters into the mailboxes and take letters out to mail elsewhere. The computer can also put in and take out. For example, when you type your program, the computer is busy placing each character you type into one of its memory boxes. When you use the LIST command, the computer then goes to each memory box, finds out which character is there, and displays it on the screen. When it has displayed the contents of all boxes (which it can do quickly), you see your program listed on the screen.

This is fine for typing your program lines (text). However, when you are running a program that draws a picture, the program gives the computer the graphic codes necessary to draw that picture. As this happens, the computer also stores the graphic codes in its memory. These graphic codes need to be stored in memory boxes other than those used by your text lines. How is this done? Easily. By using the “addresses” of the computer’s memory boxes, you can have the *program* tell the computer where to store and retrieve your picture.

In this chapter’s program, you will direct the computer to a set of memory box addresses where you will store and retrieve your picture pattern and picture colors. When you want to return to regular text, you direct the computer back to the original set of memory boxes where the program text is stored.

It is important to remember that the computer can see more than one set of memory boxes at the same time. So, when you run a program that creates a picture, the computer looks at the boxes containing the program instructions at the same time it looks at the boxes containing the color codes and picture patterns.

A final thing to know about memory boxes is that some boxes have special functions and are used by the computer all of the time. For example, there is one memory box that the computer always looks in as soon as you turn on the computer. This memory box holds a code that means “display everything as regular text.” Thus, when you begin typing, every keystroke is displayed as text and not graphic colors. This box can (and will in this chapter) be changed so that the code tells the computer to display graphic colors on the screen instead.

This and several other memory box locations will be discussed in this chapter in greater detail. You will actually use the address number of some memory boxes to

begin building your picture. Continue to think of memory as many mailboxes having their own address number. Don't feel left out if the computer's memory is still a vague, abstract concept. You will feel much more comfortable after you begin using memory box addresses to instruct the computer.

Entering the Subroutines

To start, you need Chapter 1's program listed on your screen. Type **LIST** and press **RETURN** to check. If the program is not listed, use the **LOAD** command as explained in Chapter 1 to have it place in memory.

The new program lines you need to type are shown below. Some of them are entirely new, while others will be replacing lines already in Chapter 1's program. Type in these lines exactly as we have them shown here. (Use a scrap piece of paper to block out the lines you are not typing as you try to enter these new lines.) Do not change or erase any other lines in Chapter 1's program. If you need to correct a typing mistake, refer to Chapter 1 for help.

```
21 POKE 53265,59
22 POKE 53272,29
23 POKE 56576,198
31 POKE 53265,27
32 POKE 53272,21
33 POKE 56576,199
41 FOR I = 17408 TO 18407
42 POKE I,C
43 NEXT
1005 C = 14
```

When you have typed in these new lines, you will need to **LIST** your program to check them. The new lines will be automatically inserted into Chapter 1's program. Unfortunately, your program is now too long to see on the screen at one time. There is, however, a way to list chunks of your program, depending on which sections you want to look at. For example, type **LIST 1-44** and press **RETURN**. You should now see lines 1 through 44 listed on your screen. Check them against the program below. Common typing errors will include: typing the wrong numbers next to each **POKE**; typing "ohs" (O) instead of zeroes (0), or typing small L's (l) instead of ones (1); typing wrong line numbers and possibly erasing a line that should stay in the program; or leaving out a line entirely. Re-type any line that is missing or incorrect.

To check the rest of your program, type **LIST 1000 - RETURN**.

```

1 GOTO 1000
20 REM::::::::::GRAPHICS
21 POKE 53265, 59
22 POKE 53272, 29
23 POKE 56576, 198
24 RETURN
30 REM::::::::::TEXT
31 POKE 53265, 27
32 POKE 53272, 21
33 POKE 56576, 199
34 RETURN
40 REM::::::::::COLORS
41 FOR I = 17408 TO 18407
42 POKE I, C
43 NEXT
44 RETURN
1000 REM:::::::::::
1001      MAIN ROUTINE
1002 REM:::::::::::
1005 C = 14
1100 GET A$
1110 IF A$ = "G" THEN GOSUB 20
1120 IF A$ = "T" THEN GOSUB 30
1130 IF A$ = "C" THEN GOSUB 40
1140 GOTO 1100

```

Notice that to list only a section of your program, you type:

- (a) the first line number you want to see (for example, 1);
- (b) a dash (-);
- (c) the last line number you want to see (for example, 44).

If you want to see all the lines below a certain point, you type:

- (a) the first line number you want to see (for example, 1000);
- (b) the dash only (-).

Both ways to list program chunks must, of course, be followed by a press of the **RETURN** key.

When you think you have the program entered correctly, "RUN" it to find out. We discuss below what should happen on your screen. If your program is not working as it should, hold down the **RUN/STOP** key and tap **RESTORE**. **LIST** your program and check each line again.

The computer should start out by removing the cursor from your screen (if not, check line 1 and lines 1000-1140 in your program). Just as in the previous chapter, the computer is waiting to GET A\$. This is because you did not change that line in your program. Pressing G, T, and C will still produce a visual response from the

computer, but the response will be different because you changed the subroutines. Let's see what happens.

Press **G**. The computer automatically switches to high resolution graphics. What your screen will display depends on which version of the Commodore 64 you own. If your screen is displaying colors, either in what appears to be a pattern, or just rough, jagged lines, everything is working fine. (If you do not see colors on your screen, check line 1110 and lines 20-24 in your program.) Look closely at your screen. You should be able to see the tiny dots of pixels we spoke of earlier.

Press **C**. Row-by-row your screen should be changing colors. The colors should now be a mixture of blue and black. The blue color makes up the screen's "background" color. The black color makes up the screen's "foreground" color. This chapter will teach you a way to change the foreground and background colors. In later chapters, you will learn how to control which pixels are background colored and which pixels are foreground colored. (If pressing **C** did not change the colors on your screen, check line 1130 and lines 40-44 of your program.)

Press **C** again. What happens? Hopefully, nothing. This is because the new program has the computer change the high resolution screen to blue and black whenever **C** is pressed. You have already changed this screen to blue and black by pressing **C**, so pressing it again should do nothing to the screen.

Press **T**. This "turns off" high resolution graphics and returns the computer to the regular text mode. You should see text on your screen now. (Check line 1120 and lines 30-34 if this does not work.) You can't type **LIST** or any other command yet, however, because the program is *still* running. Press **G** to see that this is true.

Try out this program as long as you like. Remember, **G** takes you to high resolution graphics, **T** takes you back to text mode, and **C** will change the high resolution screen to blue and black (this takes some time, and is done regardless of whether the screen is already blue and black). When you are ready to learn how this program works, you will need to stop it from running. If you have pressed **T** and are at the text screen, just press **RUN/STOP** as usual.

NOTE: If you are looking at the graphics screen, press **RUN/STOP** and tap **RESTORE** at the same time. This is the only way to get *directly* back to your program from a high resolution screen. Use the **RUN/STOP RESTORE** combination as a "panic button" whenever you have problems getting back to the text screen.

Breakdown of the Subroutines

To follow along with this discussion, list the new program lines on your screen (type **LIST 1-44 RETURN** and **LIST 1005 RETURN**). The rest of your program will not be listed. It is the main routine, which has a **GOSUB 20** if **G** is pressed, a **GOSUB 30** if **T** is pressed, and a **GOSUB 40** if **C** is pressed.

In order for the computer to display text or display graphics, it needs two pieces of information. To display text, it needs to know:

- (a) Where to find the text characters;
- (b) Where to find the colors of those text characters.

To display graphics, it needs to know:

- (a) Where to find your picture pattern;
- (b) Where to find the colors that are used with the picture pattern.

Thus, each time you have the computer display text or graphics, you need to tell it where to find this information. This is where your subroutines come into play. We will now tell you *how* these subroutines work. *Why* they work has been explained in three separate boxes, one for each subroutine. If you are interested in the technical explanation of these tools, be sure to read these boxes.

The first thing to notice about these subroutines is that each contains a series of POKE statements. POKE tells the computer to go to a specific memory box and place a code in it. That is all it does. POKE is always followed by two numbers, each separated by a comma. The first number is the "address" of the memory box to go to. The second number is the code that is to be placed in that memory box. This code tells the computer something to do each time you have the computer look in that memory box. This will all be much clearer in a moment. Keep in mind, though, that each memory box address and each code used in your subroutines are addresses and codes that the computer knows about and understands. In other words, you can't just make up your own memory box addresses and codes.

The first subroutine is located in lines 20 through 24. The computer will perform this subroutine task whenever G is pressed while the program is running. Three things take place within this subroutine. First, the computer is switched to high resolution graphics (line 21). This is a very important part of the tool, and a very important part of any program that is to display a picture.

The next two lines (22 and 23) tell the computer where to store and find your picture pattern and colors. The computer has 4 major sections within its memory. Each section is called a "bank," and each holds 16,000 memory boxes. These banks are numbered 0, 1, 2, and 3. Bank 0 is used to store all kinds of information, including your program text. Because Bank 0 is already storing a lot of information, it does not have very many memory boxes that are empty. To allow plenty of room to store your picture, you should always move it to Bank 1, which has plenty of empty memory boxes.

Line 22 tells the computer where to find your picture pattern and colors within a bank (any bank). This tells the computer that, no matter what bank it is looking at, it should always start at a specific place in that bank to get the picture pattern and colors. Line 23 moves the color codes and picture patterns to Bank 1. Since line 22 tells the computer where to find your picture pattern and color codes within a bank, and line 23 tells it which bank they are in, the computer now has all the information necessary to store and retrieve your picture. If these lines are left out of the subroutine, you will end up with a somewhat distorted picture. This is because the

computer will have no way of knowing which boxes hold the picture patterns and which boxes hold the colors to use.

A technical discussion of this subroutine is given in the box below. You should read "What it Does" and "Example Use." As long as you know that this subroutine is necessary in any program in order to display a picture on your screen (instead of text), you do not need to read the "Technical Description."

TOOL 20 :::::::::: GRAPHICS

```
20 REM::::::::::GRAPHICS
21 POKE 53265,59
22 POKE 53272,29
23 POKE 56576,198
24 RETURN
```

What it Does: This tool turns on high resolution graphics to allow you to display your picture in graphic colors instead of symbols and characters (text).

Example Use: To use this tool, all you need is a GOSUB 20 statement in your program. If the sole function of your program is to draw a picture, this GOSUB statement should be one of the first statements in your program.

Technical Description: POKE is a BASIC command that places a code in one of the computer's memory locations. Memory location 53265 always controls whether the computer should display graphics or text. The code 59 that is placed in this location (POKE 53265,59) tells the computer to display color graphics.

Memory location 53272 always controls where the computer stores and finds the pixel patterns and colors within a bank. Placing a 29 in memory location 53272 (POKE 53272,29) tells the computer the proper location to store the color codes and pixel patterns within the bank. As a consequence, the computer knows where to find them later. *Caution:* Numbers other than 29 can cause the color codes to be stored in the *middle* of your pixel patterns. This will cause your picture to be distorted, as the computer will try to use the color codes as pixel patterns as well.

Memory location 56576 controls which "bank" the computer should use to store and retrieve your graphic codes. The Commodore 64 has 4 available banks, each holding 16K worth of memory. Placing 198 into memory location 56576 (POKE 56576,198) tells the computer to store and retrieve your picture in Bank 1. Bank 0 is where it would normally go, but this bank is storing your program lines, and thus has less unused space.

Your next subroutine (lines 30 through 34) returns you to text mode. This is done by reversing everything done in the graphics subroutine. Notice that the memory locations used in this subroutine (locations 53265, 53272, and 56576) are exactly the same as those used in the graphics subroutine. The difference is the code that you tell the computer to put in those locations.

TOOL 30 :::::::::: TEXT

```
30 REM::::::::::TEXT
31 POKE 53265,27
32 POKE 53272,21
33 POKE 56576,199
34 RETURN
```

What it Does: This tool turns off high resolution graphics and returns you to the text mode.

Example Use: To use this tool, all you need is a GOSUB 30 statement in your program. If the sole function of your program is to draw a picture, this GOSUB statement should be one of the last statements in your program. As you will later see, your main routine will be changed so that this subroutine is only executed when the **SPACE BAR** is pressed.

Technical Description: POKE is a BASIC command that places a code in one of the computer's memory locations. Memory location 53265 always controls whether the computer should display graphics or text. The code 27 that is placed in this location (POKE 53265,27) tells the computer to display text.

Memory location 53272 always controls where the computer stores and finds the letters, symbols, and colors used to display text. The computer always expects text information to be in one place. If you try to tell it that your text information is located elsewhere, it will not understand. Therefore, you *must* replace the appropriate code in memory location 53272 in order to again display text. This code is 21 (POKE 53272,21).

Memory location 56576 controls which "bank" the computer should use to store and retrieve your text. The Commodore 64 has 4 available banks, each holding 16K worth of memory. Placing 199 into memory location 56576 (POKE 56576,199) tells the computer to store and retrieve text in Bank 0.

Finally, we come to the subroutine at lines 40-44. This subroutine is used to change the foreground and background colors of the entire graphics screen.

Your color codes are currently being stored in 1,000 different memory boxes. These memory boxes each have a specific address, starting at 17408 and continuing to 18407. Each memory box controls one block of tiny lights on your screen. If you change the color code in memory box 17408, the first block of lights will change to a different color. If you change the color code in memory box 17409, the second block of lights will change to a different color. Now, if you want to change the color of all blocks of lights on the screen, you have to put a new color code in every memory box from 17408 to 18407. This is what this subroutine does.

Line 41 says `FOR I= 17408 TO 18407`. This assigns a number to the letter "I" each time the computer uses this line. The first time the line is used, I becomes the number 17408, the second time it is used, I becomes 17409, the third time it becomes 17410, and so on, until I finally becomes 18407. As you can see, these numbers are the memory locations that store your graphic color codes.

By using the `POKE` statement in Line 42, you can change the color code in each of those memory locations. The colors displayed on your screen are determined by the `C` in `POKE I,C`. If you look at line 1005, you will see that `C = 14`. 14 is a color code that changes the background color to blue and the foreground color to black. At line 43 you find the word `NEXT`. This tells the computer to go back and get the next number for I and continue through the subroutine again. Thus, you have found a very quick way of writing 1000 `POKE` statements (`POKE 17408,14` and `POKE 17409,14` and `POKE 17410,14`, etc).

To see how the foreground/background colors are determined, we have provided a color chart below. (Another copy of this chart can be found in the Appendices for quick referral.) Notice that the colors listed at the top of each column determine the screen's background color. The colors listed down the left side of the chart determine the foreground color. The numbers that are shown at the intersection of each foreground and background color combination can be used to change the screen's color. Find the number 14 on this chart. It is at the far right-hand side of the first row. Just above 14 you see the background color `BLU2` (meaning Blue #2). The foreground color shown on this line is `BLACK`. These are the colors that were displayed on your screen.

HIGH RESOLUTION COLOR CHART

BACKGROUND COLORS

FOREGROUND COLORS	B	W	R	C	P						R	G	G	G	B	G
	L	H	E	Y	U	G	B	Y	O	B	E	R	R	R	L	R
	K	T	D	N	P	N	U	L	G	N	2	1	2	2	2	3
Black	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
White	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Red	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Cyan	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Purple	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Green	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
Blue	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
Yellow	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
Orange	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
Brown	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
Lt. Red	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
Gray 1	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
Gray 2	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
Lt. Green	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
Lt. Blue	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
Gray 3	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

To change the program so that a different color combination is used, you would make C equal to a different number on this chart. Suppose you want the background color to be red and the foreground color to be yellow. Which number would you make C equal to? If you picked 114, you are right. Change line 1005 so that C = 114. RUN the program again. Remember, you will need to press G to get into high resolution graphics, and then press C in order to see the colors change.

Now that you know how to change the screen's color, you can have a lot of fun experimenting. Be sure to change line 1005 back to 1005 C = 14 when you are done.

TOOL 40 ::::::::::: COLORS

```
40 REM:::::::::::COLORS
41 FOR I = 17408 TO 18407
42 POKE I,C
43 NEXT
44 RETURN
```

What it Does: This tool colors in a background and foreground color on the entire graphics screen. Line 1005 (C=14) determines which color combination will be displayed.

Example Use: To use this tool, you need both a GOSUB 40 statement in your main routine and a line that defines C's code (for example, 1005 C = 14). C can be set to equal any combination of foreground/background colors shown on the color chart given in this chapter.

Technical Description: Each memory location from 17408 to 18407 controls the background/foreground colors of one 8 x 8 block of pixels on your graphics screen. These blocks start at the top left-hand side of the graphics screen, and continue left to right down through the entire screen. The foreground and background colors of each block are determined by the color code placed in each memory location from 17408 to 18407. Memory location 17408 controls the color of the block in the top left corner of the screen. Memory location 17409 controls the next block to the right of this corner. The last block on your screen (lower right-hand corner) is controlled by memory location 18407.

Line 41 increments I each time the NEXT statement in line 43 is read by the computer. I begins at 17408 the first time line 41 is read. The POKE statement places a corresponding color code into the memory location specified by I's current value. C is the color code that is placed in each memory location. By changing the value of C in line 1005, the foreground and background colors of the graphics screen will change.

Choosing Colors for Your Picture

You just learned that information about colors is kept in memory locations 17408 to 18407. Keep in mind that these 1,000 locations say nothing about the image being drawn. They only tell us about the colors that will be used in the image.

In each 8 x 8 pixel block, you can use two colors. One color is used to paint in the background of the picture, such as blue for the sky. The second color is used to draw the image, such as the ship or the water. Colors used as a backdrop are called *background* colors. Colors used to draw images are called *foreground* colors because they are used for drawing shapes in front of or on top of the background color.

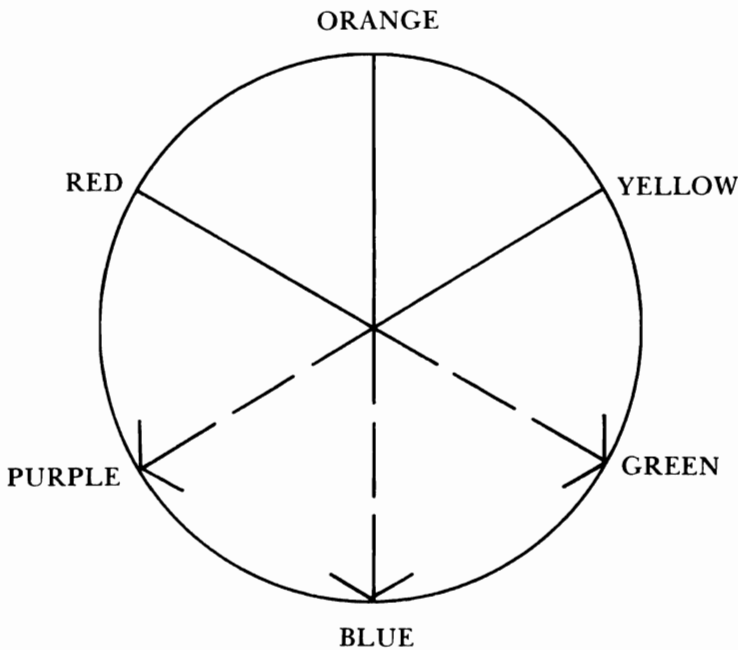
Some foreground/background color combinations produce blurred images on the Commodore 64. For example, the color code 37 (red and green), which is unshaded on our color chart, is a color pair that does not work particularly well together. Whereas, the color code 54 (cyan and blue), which is shaded, will produce a sharp, distinct picture. Depending on the type of picture you are drawing, you may want to have a sharp image or a blurred image.

Our example of the ship on water against the blue sky is more of a real life picture. This type of real imagery is called a "figurative" picture, because the figures and images look like they are real. Because it is supposed to be real looking, we chose a color combination that produces a sharp, distinct look (black against blue.)

Another way to use colors is to create a sense of depth; that is, make some shapes look far away, while others appear near and close. If a dark color is used for the background, and a light color is used for the foreground images, then the background will generally appear like it is far away. Think of a space ship and planets drawn in light colors against a dark background. The space ship and planets look relatively closer to you than the dark background color of outer space.

Certain color combinations can also create a sharp difference between the two colors. This difference is called "contrast," and an example of two contrasting colors is black and white. Another contrast can be seen between purple and yellow, or the contrast created when you use green and red. Opposite contrasting colors are called "complements." The use of complementing colors create contrast. Some of the pairs of complementing colors are:

Warm and Cool Complements



Red and Green
Orange and Blue
Yellow and Purple

You can see in the chart above that some colors (like red, orange and yellow) are called “warm” colors. Warm colors can be associated with things that in reality are warm—like the sun, a sunny day, and fire. Cool colors (like green, blue and purple) are used for objects that are usually thought of as cool or cold—like ice or cold water. Artists use colors in a very intentional way so that the viewer feels a certain way about the picture. The use of hot, fiery colors in a picture could result in an intense, emotional response to the picture. The use of cool colors in a picture could create a relaxed and calming effect.

The selection of colors for your pictures is often just as important as the images themselves. Plan in advance how you would like the pictures to appear to the viewer (true to life or hazy and foggy). Think about the images themselves. Should one image appear calm and peaceful like a cool summer lake, and another hot and fiery like the setting of a red sun? Using some of the color techniques described above, you will have more control over how a viewer feels when studying your art.

Entering the ZAP Subroutine

You are now going to enter the ZAP routine. This tool will be useful each time you want to start drawing a new picture. Its only function is to erase your existing main routine, leaving all subroutine tools in place.

Before you begin typing, a word of warning: Type this routine very carefully and doublecheck it thoroughly! There are a great many things that can go wrong if this routine is entered incorrectly and the program is run. Be especially careful of line 11 (A = 256: B = 2049: C = 1003). Make sure you make C = 1003, and no other number. As you type, each line that is too long to fit across your screen will automatically “wrap around” to the next typing line for your convenience.

```
10 REM::::::::::ZAP!
11 A = 256: B = 2049: C = 1003
12 IF PEEK(B+2) + A * PEEK(B+3) >= C THEN 15
13 B = PEEK (B) + A * PEEK (B+1): ON ABS(B<>0) GOTO 12: END
14 A = 256: B = PEEK(251) + A * PEEK (252)
15 IF PEEK(B+1) = 0 THEN END
16 PRINT CHR$(147) PEEK (B+2) + A * PEEK (B+3): PRINT "GOTO 14"
17 POKE 251, B - INT (B/A) * A: POKE 252,B/A
18 POKE 631,19: POKE 632,13: POKE 633,13: POKE 198,3: END
```

After you have typed and doublechecked each line in this routine, you can test it. To use this routine, you need to **RUN** it by itself—not your entire program. Remember when you listed a portion of your program, starting at line 100? You can also have the computer run portions of your program, starting at a specific line number. You want the computer to run lines 10 through 18. If you look at line 18, you will see that it ends with the word **END**. This word has the same effect as pressing **RUN/STOP**.

Type **RUN 10** and press **RETURN**. The 10 told the computer where to start running the program. The **END** in line 18 will stop the computer.

The first thing that happens is that your program is cleared from the screen. In the top left-hand corner you will see several flashing items. First, the line numbers you are erasing will flash by (lines 1005, 1110, 1120, 1130 and 1140). Beneath this, “**GOTO 14**” and “**READY.**” will flash on and off. Finally, the cursor will be flashing. When the computer has completed the routine, your screen will show:

```
1140
GOTO 14
READY.
```

Type **LIST** and press **RETURN** to see what happened. You should find that all of your subroutines still remain. What will be missing is the main routine (lines 1005 and higher). Check the program lines below to make sure your ZAP routine did not erase any of the necessary subroutine tools.

```

1 GOTO 1000
10 REM::::::::::ZAP!
11 A = 256: B = 2049: C = 1003
12 IF PEEK(B+2) + A * PEEK(B+3) >= C THEN 15
13 B = PEEK (B) + A * PEEK (B+1): ON ABS(B<>0) GOTO 12: END
14 A = 256: B = PEEK(251) + A * PEEK (252)
15 IF PEEK(B+1) = 0 THEN END
16 PRINT CHR$(147) PEEK (B+2) + A * PEEK (B+3): PRINT "GOTO 14"
17 POKE 251, B - INT (B/A) * A: POKE 252,B/A
18 POKE 631,19: POKE 632,13: POKE 633,13: POKE 198,3: END
20 REM::::::::::GRAPHICS
21 POKE 53265,59
22 POKE 53272,29
23 POKE 56576,198
24 RETURN
30 REM::::::::::TEXT
31 POKE 53265,27
32 POKE 53272,21
33 POKE 56576,199
34 RETURN
40 REM::::::::::COLORS
41 FOR I = 17408 TO 18407
42 POKE I,C
43 NEXT
44 RETURN
1000 REM:::::::::::
1001      MAIN ROUTINE
1002 REM:::::::::::

```

If the computer did not respond as described above, or some of your subroutine lines are now missing, check each line in the ZAP routine. Re-type any of the ZAP routine lines that have mistakes in them. If your ZAP routine erased one or more other subroutine lines, re-type them as well.

This tool is discussed in more detail in the tool box below. It is only important for you to understand two things in order to use this tool. First, it is a quick and handy way to save your tools when you want to draw a new picture. Second, line 1 tells the computer to go directly to line 1000 whenever you RUN the program. This GOTO line is especially important now. If this line were omitted, your subroutines would automatically be the first thing performed when the program is running. Since the ZAP routine starts at line 10, the first thing to happen is your main routine would be erased!

<p>TOOL 10 :::::::::: ZAP!</p> <pre> 1 GOTO 1000 10 REM::::::::::ZAP! 11 A = 256: B = 2049: C = 1003 12 IF PEEK(B+2) + A * PEEK(B+3) >= C THEN 15 </pre>
--

```

13 B = PEEK (B) + A * PEEK (B+1): ON ABS(B<>0) GOTO 12: END
14 A = 256: B = PEEK(251) + A * PEEK (252)
15 IF PEEK(B+1) = 0 THEN END
16 PRINT CHR$(147) PEEK (B+2) + A * PEEK (B+3): PRINT
   "GOTO 14"
17 POKE 251, B - INT (B/A) * A: POKE 252,B/A
18 POKE 631,19: POKE 632,13: POKE 633,13: POKE 198,3: END

```

What it Does: This routine will erase all the lines of your main routine. It will, however, leave each of your subroutine tools untouched so that you can use them to draw a different picture.

Example Use: To use this routine in a program, you need to have a GOTO statement positioned somewhere before the ZAP routine begins. This is so you can run your program whenever necessary, without having the ZAP routine used. When you *do* want to erase the main routine, type RUN and the line number the ZAP routine starts on (for example, RUN 10 RETURN).

Technical Description: This is a very unusual routine. In this section we will explain *what* the routine does. *How* this routine works will not be discussed in this beginner's graphics book, mainly because of the complexity of the routine. Its value outweighed omitting it entirely from the book, however, so it is included with only the following general explanation of its workings.

Normally, to delete a set of lines from your program, it is necessary to type each line number and then press RETURN. If there are many lines to be deleted (as there will eventually be in your main routine), this could take some time to do.

One alternative would be to have a program that prints each line number on the screen, pausing to let you press RETURN after each one. Even better, why not have the program press RETURN for you after each line number is printed? If it could do all that, all you would need to do is sit back and watch. This is exactly what happens when you use the ZAP routine. It should be noted here that this routine is actually a small program within your larger program. That is why it is executed by typing RUN.

Drawing the Lighthouse

As promised, you are going to draw something on the screen before this chapter ends. Because you have ZAPPED your main routine, a new one has to be entered. It is shown below. Before you begin typing, look at line 5010. It contains IF A\$ = " " THEN 6000. Be sure to type a space between the quotation marks, (" ").

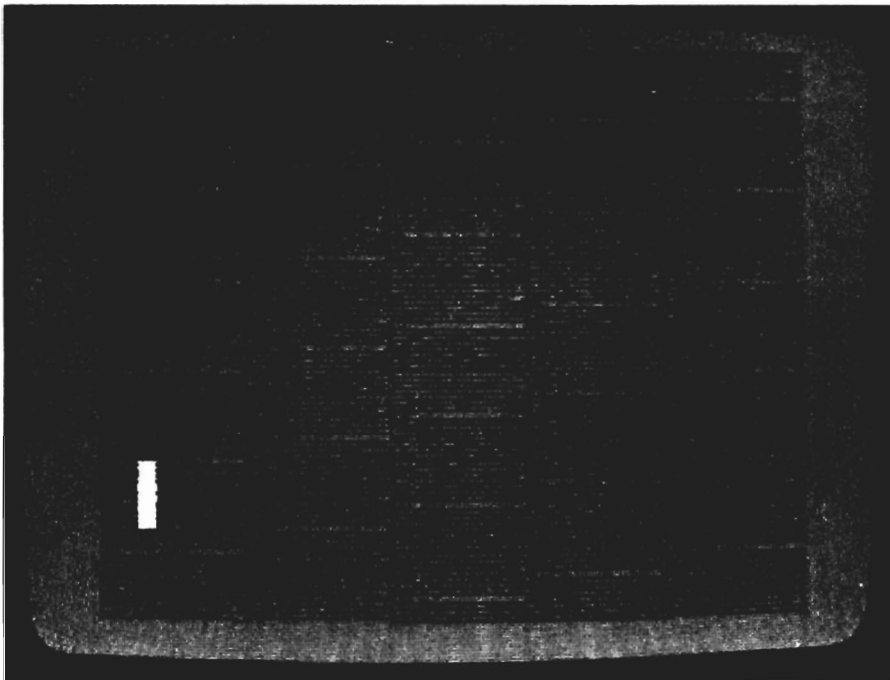
```

1100 GOSUB 20          : REM GRAPHICS
1110 C = 14: GOSUB 40: REM COLORS
1200 REM::::::::::LIGHTHOUSE
1210 POKE 18090,0: POKE 18130,17
1220 POKE 18170,17: POKE 18210,17
5000 GET A$
5010 IF A$ = " " THEN 6000
5020 GOTO 5000
6000 GOSUB 30
6010 END

```

Check the program before you run it, especially lines 1210 and 1220. If the numbers are correct, **RUN** the new program.

You should immediately see the high resolution screen. In addition, the screen will be colored in with the blue and black foreground/background colors (even if it already is). Finally, the lighthouse will appear. The lighthouse will be in the shape of a rectangle standing on end. Its tip will be black, and the rest of it will be white.



If you try pressing **G**, **T**, or **C**, you will quickly find that they no longer do anything to the screen. Try pressing the **SPACE BAR**. You are now back to your text screen, and the program has quit running.

If you had any problems with this program, check lines 1100 through 6010 again. If you never saw the high resolution screen, check line 1100. If the high resolution

To display color graphics, your screen is divided up into 1000 blocks, as shown above. Each block contains 64 of the tiny lights (pixels) you see on the screen. The blocks are numbered from 0 to 999, starting at the top left-hand corner and moving right. Each row is 40 blocks across, and each column is 25 blocks down.

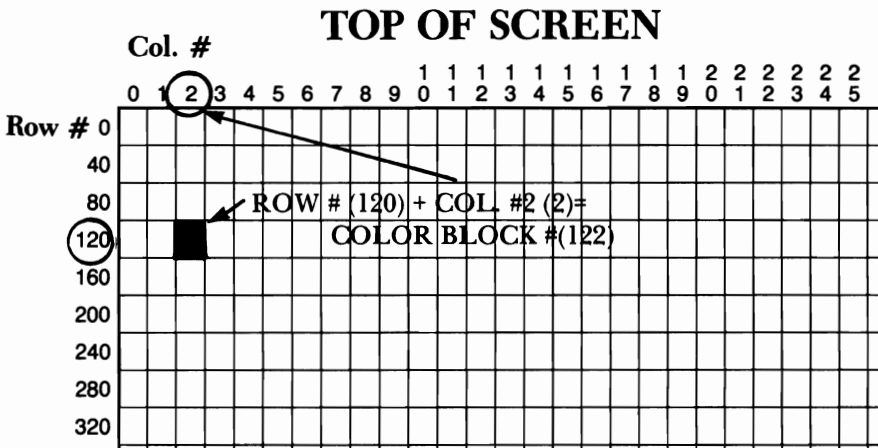
Using special numbers, you can change the foreground and background colors for *each* of the blocks. Thus, although each block can only display two colors, all of the blocks do not have to be displaying the same two colors. This is how the lighthouse was drawn. The program tells the computer to go to one of the blocks and change its foreground color to black and its background color to black (making the entire block of lights all white). Three other blocks are used to display the bottom part of the lighthouse. The program has the computer display each of these three blocks with white as both the foreground and background colors (again, giving the blocks a solid color).

Now, how did the computer know which blocks you wanted to change to these colors? Look at line 1210 and 1220. Notice that these statements POKE (place) color codes (0 and 17) into memory boxes 18090, 18130, 18170 and 18210. These memory boxes should look familiar. They are within the 17408 and 18407 memory box range that holds all of your color codes. For your lighthouse, you changed four of them to different colors.

By using the chart shown above, along with some graph paper, you can easily POKE colors into any box you want to. On your graph paper, draw a large box that is 40 small blocks across, and 25 small blocks high. Take the time to label and number your paper as shown on the above chart.

Think of the graph paper as your screen and locate the block or blocks on it that you want to color differently.

Suppose you want to change the colors for the block located in the third column, fourth row (see diagram below). By adding the block's column number (2) to its row number (120), you will get the exact *block number* for that block (122).



Since you know that the first block is controlled by memory location 17408, you *must add 17408* to each block number you decide to change. $122 + 17408 = 17530$. To change the colors, you use a **POKE** statement and a color code (for example, **POKE 17530,0** would change this block to black).

Look at lines 1210 and 1220 in your program. Notice that the computer puts (POKES) a color code into locations 18090, 18130, 18170, and 18210. These locations were found using a piece of graph paper, locating the block numbers to change, and adding 17408 to each of them.

In line 1210, the color code 0 is used with block 18090. If you look back at the color chart, you will see that the color code 0 makes that block black against black (or totally black). The other location in line 1210, as well as the two locations in line 1220, all have a color code of 17. The chart shows us that this produces white foreground pixels and black background pixels (or a block that is totally white).

To test this new idea out, try changing lines 1210 and 1220. Locate four blocks on your graph paper and find out what number between 0 and 999 they use. *Add 17408* to all four of these block numbers. Insert these numbers in the old **POKE** statements (replacing 18090, 18130, 18170, and 18210). To change the colors, pick different ones from the color chart and insert them after the comma in the **POKE** statements. **RUN** the program to see what happens.

Remember, the formula for changing the colors in a block always requires:

POKE 17408 + Block Number,Color Code

You can either add 17408 to the Block Number yourself, or you can have the computer do it (for example, **POKE 17408+12,0**).

When you are through, make sure you change the lines back to:

1210 **POKE 18090,0: POKE 18130,17**
1220 **POKE 18170,17: POKE 18210,17**

Now to the final lines in your new program, lines 5000 through 6010. Line 5000 again has the computer **GET A\$**. When the program is running, the computer not only displays the high resolution screen, it also sits there and calmly waits for **A\$**. When a key is pressed, two things can happen. Line 5010 tells the computer to go to line 6000 if the **SPACE BAR** is pressed. Line 5020 says to go back and **GET A\$** (line 5000) if anything else is pressed. Thus, the computer will continue to display your picture and wait for **A\$** to be the **SPACE BAR** before it ever reads line 6000. When you finally press the **SPACE BAR**, the computer reads line 6000, goes up to the subroutine which takes you back to text mode, and then **RETURNS**. When it returns to the main routine, it reads line 6010, which ends the program.

Summary

Save your new program under "CHAPTER 2." (Did you remember to change lines 1210 and 1220 back to the lighthouse? Does **C=14** in line 1005?) If you've

forgotten how to save your program, refer to the appropriate section in Chapter 1.

In this chapter you have learned several new things about Commodore 64 graphics. Using your graphics subroutine, you can easily get to the high resolution screen from within any program. Using your color tool, you can change the foreground/background colors used for the entire screen. If, instead, you only want to change the colors in one of the screen blocks, you locate the block number on your graph paper (it will be between 0 and 999), *add* 17408 to this number, and **POKE** a color code into it. Finally, you have a subroutine tool which takes you back to text mode whenever necessary.

We recommend that you take advantage of these tools each time you draw a picture. Each of them is necessary, so there is no use in re-typing them over and over again. Just **RUN 10 (ZAP)** and you're set!

Below are two exercises on poking colors into screen blocks. For those of you who like a challenge, go ahead and try them. (The solution immediately follows if you need to peek.) Don't try these exercises until Chapter 2's program is safely stored on your disk or tape.

Exercise #1

First, change the entire screen to blue against blue. Next, locate blocks 2, 41, 42, 43, 82, 121, and 123 on your graph paper. Using 7 **POKE** statements, change:

- (a) block 2 to white against white
- (b) blocks 41, 42, 43, and 82 to red against red
- (c) blocks 121 and 123 to black against black

RUN the program to see what happens.

Solution

To change the screen to solid blue, line 1110 should be **C = 102: GOSUB 40: REM COLORS**. To change the blocks, you can use any new line numbers you want, but the 7 **POKE** statements must be in the form of:

```
POKE 17410, 17
POKE 17449, 34
POKE 17450, 34
POKE 17451, 34
POKE 17490, 34
POKE 17529, 0
POKE 17531, 0
```

Exercise #2

In addition to the lines you typed in Exercise #1, add lines that will change: blocks 49,85, 86, 87, 88, 126, and 128 to brown

Solution

Again, using any new line numbers you want, the new POKE statements should have been:

```
POKE 17457, 153  
POKE 17493, 153  
POKE 17494, 153  
POKE 17495, 153  
POKE 17496, 153  
POKE 17534, 153  
POKE 17536, 153
```

If you completed the exercises correctly, RUN the program to see a somewhat primitive drawing of "Man and His Dog." If you had any problems, just remember that 17408 needs to be added to each block number. This gives you the memory location which controls the colors of that block. By poking that location with a color code, the block will change to the desired colors.

In the next chapter, you will learn how to find and plot points (pixels).

Chapter 3

FINDING AND PLOTTING POINTS

In this chapter you will learn how to paint the blue background sky, and how to draw in the water. To paint the sky, you could do one of two things. First, you could use the tool at subroutine 40, making C equal to a blue against blue color code. The problem with this approach is that you would never know which pixels are foreground colored and which are background colored. If you were to later change one of your blocks to different colors, those unknown foreground/background pixels would show up in a pattern. So, instead you will use a different approach that involves a new tool. This way, you always know that each pixel in a block will display the block's background color, unless you specify otherwise.

To paint the water, you will need to find and then plot certain points. A "point" is just another term for a pixel on your screen. To "plot" it, you have the computer change it to the foreground color—in your case, the color black.

There are many uses for plotting points. For the picture you are drawing, the speckled look of the water is achieved by plotting many points very close together. What you will see on the screen is the wholeness of the shape (the water), rather than each of the individual points. This illusion is like the fullness of leaves on a tree. Many people paint a tree's leaves as a whole, solid colored shape instead of painting each individual leaf. The same thought can be applied to a beach scene. You see the wholeness of the beach rather than each grain of sand.

On the other hand, if you were to plot your points farther apart, the viewer would see each point as an individual shape. An example would be plotting many white points far apart on a dark background to create the look of stars at night.

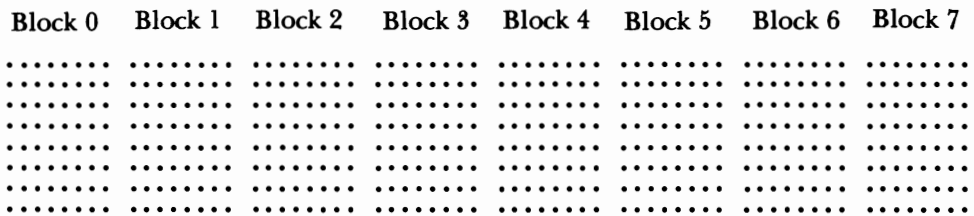
As you continue through this book, we will provide you with more design and color techniques that you will use later to draw your own pictures. For now, you need to learn how to pinpoint one of those 64,000 pixels on the screen so that the computer can then plot it.

Locating A Point

Look at your screen. If you look very closely, you might be able to see the individual dots of light that total 64,000 in all. Now, suppose you wanted someone to change 5 of those dots, all in a row, to the color black. How would you go about telling them which dots to change (without using your hands)? You would have to say something like: "Change the 5 dots that are in the 100th column, rows 50-54."

This is close to how you will tell the computer which pixels to plot. For exact precision, you could count each column across the screen (there are 320 of them), and each row down the screen (there are 200 of them). Or, instead, you could make a good guess—within 2 columns and 2 rows. Since each column and each row is only a difference of 1 tiny dot, you will learn how to make a good guess.

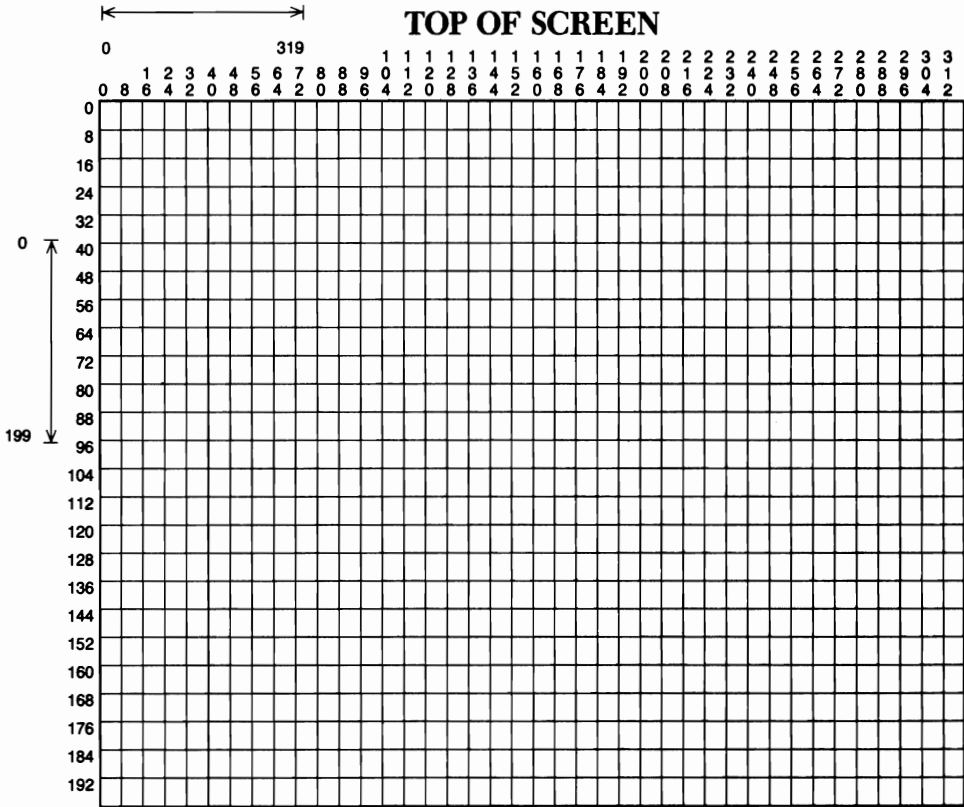
To follow along, get out your graph paper again. Draw a new box that is 40 columns (small boxes) across, and 25 rows (small boxes) down. Label this sheet of paper "X,Y PIXEL POINTS." This sheet should remind you of your "Color Memory" paper. It is, in fact, similar. Each memory location on the previous graph sheet controlled the colors for one block of 64 pixels. This new sheet will also represent those blocks of pixels. Look at a blow up of the first 8 blocks to see how the 64 pixels are arranged:



(Note that your screen does *not* separate the blocks with a space.)

Each pixel on your screen can be located by the computer if you specify the pixel's column number and row number. For example, the pixel located in the top left corner of Block 0 (computer stuff *always* starts with zero) is said to be in column 0, row 0. The next pixel to its right is said to be in column 1, row 0. Thus, each pixel in the first row is in row 0. The first pixel in the second row is said to be in column 0, row 1. The first pixel in the third row is said to be in column 0, row 2. Thus, each pixel in the first column (down to the 200th row) is in column 0. To plot a point, *all* you have to know is the exact column number (0-319) and the exact row number (0-199) that the pixel falls in. Sound too difficult? It is. There is an easier way.

Instead of numbering 320 columns and 200 rows on your graph paper (if that were possible), you just number the top, left pixel in each block. To illustrate, number across the top row and down the left column of your blocks as follows:



Add the labels on this chart to your X,Y PIXEL POINTS graph sheet.

Now you can approximate the location of any pixel. For example, suppose you drew a picture on this graph sheet. When done, you found that you needed to plot a pixel that was just a little past column 80, and just a little below row 144. You would make a close guess of maybe column 83, row 146. You would find that the guess was probably within 2 pixels either way.

The last thing to learn about locating a pixel is that the computer will not understand the words "column" and "row." The way your new subroutine works, however, allows the computer to understand that X represents the *column*, and Y represents the *row*. To have it plot a pixel in column 83, row 146, you tell it that X = 83 and Y = 146. To plot the black pixels in your picture's water, you will give the computer a whole set of X,Y numbers to use. These X,Y number sets are called the *coordinates* of a point.

Painting the Background

To get started, LOAD and LIST Chapter 2's program. Below are the new lines you will type to paint your background color. Notice that all you need is a subroutine tool and a GOSUB statement.

```
50 REM:::::PAINT BACKGROUND
51 FOR I = 24576 to 32575
52 POKE I,0
53 NEXT I
54 RETURN
1120 GOSUB 50      : REM PAINT BKGROUND
```

Type in these new lines now. Be aware that the comma in line 52 is followed by a zero (0) and not an "oh" (O). In line 1120 you can space the colon (:) and REM statement over as far as you like. Spacing it over makes it stand out more in your program.

When you have checked your typing and corrected any mistakes, RUN the program. It will take several seconds for anything visual to happen. This is because the computer is busy changing each color block to blue and black. It will always do this regardless of whether the screen already is blue and black, because the program instructs it to.

You will then see each row of blocks on your screen change to all blue. This may not seem like much, but it accomplishes two things. First, the screen is cleared of any "mish-mash" of colors, leaving a nice, solid background color. Second, you are now fully aware of which pixels are foreground colored (0), and which pixels are background colored (64,000). Later, you can change the colors used in any block with no surprises (foreground pixels) showing up on the screen.

Press the **SPACE BAR** to return to your program. If you had any problems, check the screen for "SYNTAX ERROR" or any other message. Error messages give the first line number the computer had problems with. (You can enter a program incorrectly and not get a message, so don't count on them every time.)

NOTE: If your program is not running properly, and pressing the **SPACE BAR** does not return you to text mode, carefully type: **GOSUB 30 RETURN**. You won't be able to see the typing as you type, so go slowly. This is a handy way to use a subroutine outside of the program. You should be returned to text mode and can check for error messages. Pressing **RUN/STOP** and **RESTORE** returns you to text mode, but at the same time erases the text screen.

Breakdown of the Paint Background Routine

Make sure you have lines 50-54 (LIST 50-54 **RETURN**) and line 1120 (LIST 1120 **RETURN**) on your screen. You may have noticed that this subroutine is almost identical to the subroutine at line 40. They are both closely related. The subroutine at line 40 gives the computer the background/foreground colors for the screen's 1000 blocks of pixels. This new subroutine tells the computer the current picture pattern to use when displaying the background/foreground colors. The picture pattern is determined by 8000 memory locations (24576 through 32575), each of which controls 8 pixels. Poking a pattern code into each of these memory locations tells the computer which pixels should be foreground colored.

Recall that each of the screen's 1000 blocks is made up of 8 rows of 8 pixels. For each row of 8 pixels, there is one memory location that controls how many of the 8 pixels are foreground colored. Thus, although there is only one memory location which controls the colors of each block, there are 8 memory locations which control the picture patterns within each block. Since there are 1000 blocks, there are 8000 memory locations controlling the entire screen's picture pattern. By poking a code of 0 into each of these blocks, you tell the computer that none (0) of the pixels in each row of 8 should be foreground colored (black). That leaves only the background color (blue), which the computer appropriately uses.

The importance of this subroutine tool is two-fold. First, most pictures have a background color that needs to be painted. This tool accomplishes this for you. Second, the process of changing all foreground pixels to background pixels automatically clears any previous image off of the high resolution screen. Once a foreground image is on the high resolution screen, it stays there. The only time it is erased is when you change the foreground pixels to background pixels, or you turn the computer off. Inserting this tool in all of your picture-drawing programs rids you of having to turn the computer off and on again each time you want to draw a new picture.

TOOL 50 :::::::::: PAINT BACKGROUND

```
50 REM::::::::::PAINT BACKGROUND
51 FOR I = 24576 TO 32575
52 POKE I,0
53 NEXT I
54 RETURN
```

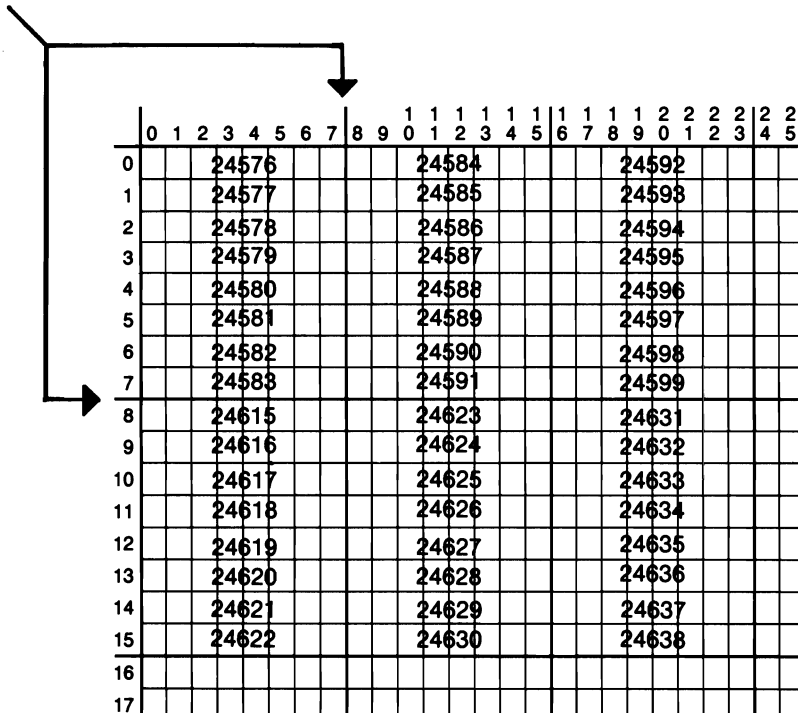
What it Does: This tool clears the high resolution screen of any previously drawn image, while painting in your background color.

Example Use: To use this tool, all you need is a GOSUB 50 statement in your main routine.

Technical Description: Memory locations 24576 through 32575 control the foreground/background pixel pattern for each byte of pixels on your screen. A byte, as used here, is one row of 8 pixels in a block of pixels. The first 8 memory locations (24576-24583) control the 8 bytes of pixels in the top left block on your screen. The next 8 locations control the 8 bytes of pixels in the second block on the first row. Each set of 8 memory locations controls 8 bytes in each block, across each row of blocks, down through the screen.

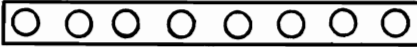
The diagram below shows a portion of the top left blocks on your screen. In each block, the memory locations controlling the pixel pattern for each byte is given. Again, a byte is a row of 8 pixels within a block. Notice that the memory locations do not start at the top left and then continue down the entire column. Instead, they continue down one block, and then move to the top of the next block in that row of blocks.

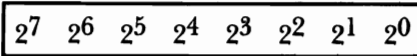
8 x 8 PIXEL BLOCK

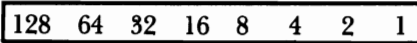


TOP, LEFT CORNER OF SCREEN

To change the pixel pattern within a byte, you **POKE** a code into the memory location which controls the pattern for that byte. This code tells the computer exactly which pixels are to be foreground colored. The code is determined by adding up powers of 2. Each pixel in a particular byte is assigned a power of 2, beginning with the right-most pixel and moving left, as shown below:

Byte of
Pixels: 

Powers of 2: 

Which is
the same as: 

Select the pixels you want foreground colored, add up their assigned numbers, and **POKE** the resulting number into the appropriate memory location. This subroutine **POKEs** a 0 into each memory location, meaning 0 (no) pixels should be foreground colored in any of the 8000 bytes.

Note that these powers of 2 (128, 64, 32...1) do *not* change from pixel to pixel. As long as you **POKE** the correct memory location, the computer does not need different powers of 2 for each byte of pixels.

As an example, suppose you wanted the first pixel, and every other pixel in the top left byte changed to the foreground color. The **POKE** statement would be: **POKE 24576,85**.

Drawing the Water

To draw the water, you will need to type several new lines. The new lines to type for the subroutine are:

```

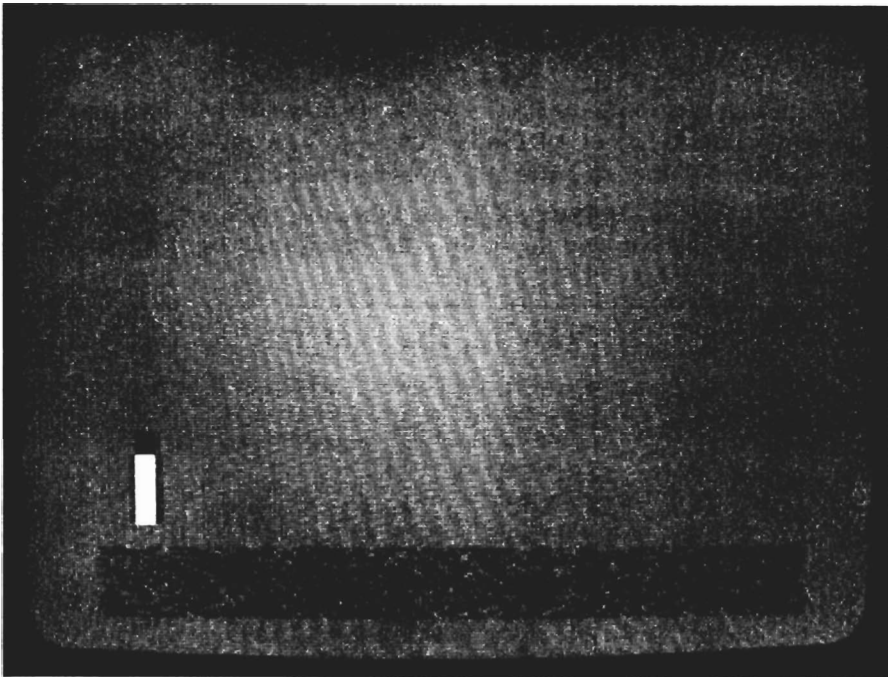
60 REM::::::::::FIND A POINT
61 ROW = INT(Y/8)
62 COL = INT(X/8)
63 LINE = Y AND 7
64 BIT = 7 - (X AND 7)
65 BYTE = 24576 + ROW*320 + COL*8 + LINE
66 CBYTE = 17408 + ROW * 40 + COL
67 RETURN
70 REM::::::::::PLOT A POINT
71 GOSUB 60
72 POKE BYTE,PEEK(BYTE) OR 2 ↑ BIT
73 POKE CBYTE,C
74 RETURN

```

Carefully type the lines shown above. Line 72 contains a “†” character. This key is located directly to the left of the **RESTORE** key. When you have typed these lines correctly, continue by typing the new main routine lines shown below (they are separated from the subroutine lines to give your eyes a break):

```
1300 REM:::::WATER
1310 FOR Y = 176 TO 199
1320 FOR X = 0 TO 319
1330 IF RND(1) < .3 THEN GOSUB 70
1340 NEXT X: NEXT Y
```

RUN this program to see this chapter’s finished product. The lighthouse will be erased from the screen as the computer fills in the background color (this will take awhile). Next, the lighthouse will be drawn again as the computer comes to those program lines. Finally, you will begin to slowly see the water plotted near the bottom of your screen. For each row of pixels, the computer changes about 1/3 of them to black. This gives the visual effect of the shadows created by the blue waves.



From now on, you will need to be patient with your Commodore 64 as it plots the points necessary to draw your picture. When the water is complete, take a moment to look at your screen. It's beginning to look like a picture!

RUN the program again if you like. When you are through, be sure to press the **SPACE BAR** to return to your program listing.

Breakdown of the Water Routine

List lines 1300-1340 on your screen. In addition, have your "X,Y PIXEL POINTS" graph sheet on hand.

Line 1310 tells the computer that the Y (row) locations to plot begin at 176 and continue through 199. Looking at our graph sheet, you will see that this involves the bottom three rows of blocks on the screen. This area was pre-determined by our artist.

Line 1320 tells the computer that the X (column) locations to plot begin at 0 and continue through 319. This is the entire width of each row. Thus, the water fills the bottom three blocks across the entire width of the screen.

As the computer reads this section of the program, it first comes to line 1310. It begins by setting Y equal to 176. Next, it comes to line 1320 and sets X equal to 0. This gives the computer the first X,Y coordinates to use in plotting (0,176).

Line 1330 says `IF RND(1) < .3 THEN GOSUB 70`. Whenever the computer reads a `RND(1)` statement, it internally generates a random number between 0 and 1 (*not* including 0 or 1). This results in some fractional number like .539, .686, .311, etc. Line 1330 tells the computer to generate the random number and `IF` it is less than (`<`) .3 `THEN GO` to the `SUB`routine at line 70 (which will plot the X,Y point found in lines 1310 and 1320). Statistically, the random number will be less than .3 about 1/3 of the time. Thus, about 1/3 of the time the computer goes to the subroutine at line 70 and plots the X,Y point it found in lines 1310 and 1320. If the random number is greater than .3, the computer does *not* plot the point. In either event, the computer plots or doesn't plot, and then continues to line 1340.

Line 1340 first says to get the `NEXT X`. This sends the computer back to line 1320, where X is then changed to its next value (1). Y is still 176, so the two coordinates now are 1,176. Again, line 1330 is read and the point 1,176 is plotted (changed to black) if the random number generated this time is less than .3.

The `NEXT Y` statement is never read until all of the X's are used up by the `NEXT X` statement in line 1340. This keeps Y stable, while every possible X (0-319) is plotted 1/3 of the time. The result is that the points in row 176 are plotted first.

When X finally becomes 319, and the point is plotted (or not), the `NEXT Y` statement is read. This takes the computer back to line 1310, where Y is set to 177. The computer comes to line 1320, sets X back to 0, and begins with the first coordinate of the next row (0,177). X again increases from 0 to 319 while Y remains stable at 177.

What you have here is a “FOR..NEXT statement” within a “FOR..NEXT statement.” Y stays at one value until all of the X values have been processed. Then Y changes to its next value, and all X values are again processed. As each new coordinate is established, line 1330 determines if the subroutine tools are used to plot it. Again, the plotting will occur randomly because you are using a random number as the determining factor. Because the random number needs to be less than .3 in order for the point to be plotted, only 1/3 of the X,Y coordinates ever actually get plotted.

Now let’s see what happens when the random number *is* less than .3 and the computer is sent to the subroutine at line 70. List lines 60-74 on your screen.

Curiously, even though the main routine sends the computer to line 70, line 71 immediately sends the computer to line 60. This is because the computer always needs to *find* the point on the screen before it can plot it. The subroutine beginning on line 60 is a tool that finds the point on the screen. You may still wonder why line 1330 doesn’t just say GOSUB 60 if that is where the computer needs to go first. The reason the program is written like this is to make sure you never forget to “find the point” before you try to “plot the point.” Back up to the subroutine at line 60.

The first thing the computer needs to know is the general location of the pixel to plot. To do this, the screen is divided up into the imaginary 40 x 25 blocks as shown on your graph paper. Given a Y coordinate, line 61 will figure out which row of blocks the pixel is in. Given an X coordinate, line 62 will figure out which column of blocks the pixel is in. This narrows down the location considerably—from 64,000 pixels to only one out of 64 pixels. The subroutine still hasn’t determined which of these 64 pixels is to be plotted, however. Line 63 will figure out which row in the *block* the pixel falls into. Line 64 will figure out which column in the *block* the pixel falls into. Line 65 sorts the information found in lines 61 through 63 to come up with the exact block and row within that block that the pixel is located in.

Line 66 has a separate purpose from locating the point on the screen. This line will determine the memory location which controls the 2 colors of the block the point is located in. It stores the number of this memory location in a place-holder called CBYTE. You will see the importance of this step in a moment.

Once all of this is done, the computer can then RETURN to the subroutine at line 70 to actually plot the point. Line 72 plots the point. Line 73 POKES CBYTE with C (POKE CBYTE,C). So, if you ever change the value of C before plotting a point, this line will POKE the new color code into the appropriate memory location. This will change the block’s background color, as well as the color of the newly plotted point. Keep in mind that any other pixels which were previously plotted in that block will also change to the new foreground color.

TOOL 60 ::::::: FIND A POINT

```
60 REM::::::::::FIND A POINT
61 ROW = INT(Y/8)
62 COL = INT(X/8)
63 LINE = Y AND 7
64 BIT = 7 - (X AND 7)
65 BYTE = 24576 + ROW*320 + COL*8 + LINE
66 CBYTE = 17408 + ROW * 40 + COL
67 RETURN
```

What it Does: This tool enables the computer to locate any point on the screen in order to plot it (see Tool 70.)

Example Use: To use this tool, the main routine must specify the X (column) and Y (row) locations of the point to find. For example:

```
1310 Y = 180
1320 X = 10
```

In addition, you will need a GOSUB 60 statement within your program. Usually, GOSUB 60 will appear in the PLOT A POINT tool, which requires the point to be found before it can be plotted (see Tool 70).

Technical Description: In this chapter we discussed how the program takes X,Y coordinates and carefully narrows down where that point is located. Now you can follow through the technical discussion step-by-step.

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

Y can range between 0-199. The computer groups these pixel rows into groups of 8. This means there will be a total of 25 block rows ($200/8 = 25$). $Y/8$ will yield a number between 0 and 24, with a remainder between 0 and 7. We are not interested in the remainder right now, so the BASIC statement "INT" ("integer") removes the remainder and leaves the whole number. This is stored in the variable called "ROW" until needed later.

```
62 COL = INT(X/8)
```

X can range between 0-319. The computer groups these pixel columns into groups of 8. That means there will be a total of 40 block columns ($320/8 = 40$). $X/8$ produces a number between 0 and 39, with a remainder between 0 and 7. This remainder is discarded by the use of INT (integer), and the final whole number is stored in the variable called "COL" until needed later. Now the location is narrowed down to one 8 x 8 block of pixels.

63 LINE = Y AND 7 Line

61 discarded the remainder gotten by $Y/8$. However, this remainder is very important because it tells us in which line to find the pixel within the 8 x 8 block. Line 63 uses a technique called "Boolean Logic" to retrieve the remainder discarded earlier. This number will be between 0 and 7, and will be the block row (0-7) that the pixel to find is in. Now the location has been narrowed down to 1 particular 8-pixel row.

64 BIT = 7 -(X AND 7)

In line 62 you threw away the remainder from $X/8$. This remainder tells us which pixel column within the 8 pixel row the pixel to plot is in. Again using Boolean Logic, this remainder is retrieved. The remainder will be between 0 and 7, and will be the pixel column needed. There is a slight problem, however, that you didn't have before. You see the pixels numbered like this:

```
0 1 2 3 4 5 6 7
O O O O O O O O
```

The computer thinks the pixels are numbered like this:

```
7 6 5 4 3 2 1 0
O O O O O O O O
```

This is not a big problem, however, because line 64 subtracts the column number (0-7) from 7, and all the pixels are then numbered the way the computer likes them. The program now has everything it needs to know in order to plot the point. This information, though, is scattered in four different variables: ROW, COL, LINE and BIT.

65 BYTE = 24576 + ROW*320 + COL*8 + LINE.

This line combines the ROW, COL and LINE variables that the pixel is located in, and adds the beginning memory location (24576) to this total. The result is the byte that controls the pixel pattern at your X,Y coordinate. BIT still contains the pixel number within the byte, which the subroutine at 70 needs to plot the pixel on the screen.

66 CBYTE = 17408 + ROW * 40 + COL.

This line is an added bonus the computer gets from the variables ROW and COL. By adding the beginning color memory location (17408) to these variables, CBYTE will contain the color memory location that controls the colors for the pixel to plot. Don't forget that it also controls the colors for the other 63 pixels in the block being plotted in.

TOOL 70 :::::::::: PLOT A POINT

```
70 REM::::::::::PLOT A POINT
71 GOSUB 60
72 POKE BYTE, PEEK(BYTE) OR 2 ↑ BIT
73 POKE CBYTE, C
74 RETURN
```

What it Does: This tool plots the point or points specified by the X,Y coordinates in the main routine. In addition, this tool POKES the memory location controlling the colors of the blocks being plotted in. The memory locations are poked with "C", which is a variable holding a color code value.

Example Use: To use this tool, the main routine must specify the X (column) and Y (row) locations of the point to plot. For example:

```
1310 Y = 180
1320 X = 10
```

In addition, you will need to "find the point" before it can be plotted. This can be handled by a GOSUB 60 statement before your GOSUB 70 statement, or a GOSUB 60 statement within the plotting tool (as we have done here). Finally, "C" should be set to any new color code necessary for the block being plotted in (for example, line 1305 could have been inserted with C = 94, had we wanted the water to be plotted green against blue).

Technical Description: Line 71 uses the subroutine at line 60 to convert your X,Y coordinates into terms the computer can understand. On RETURN from GOSUB 60, BYTE contains the byte number that controls the pixel's pattern, and BIT holds the pixel number controlling the pixel. CBYTE contains the memory location that controls the colors for the 8 x 8 block that the pixel is in.

72 POKE BYTE, PEEK(BYTE) or 2↑BIT
Line 72 plots the pixel found in Tool 60.

You could POKE BIT into BYTE to plot the pixel. Unfortunately, doing this will cause any other pixels controlled by that byte to be changed to the background color. The way to get around this is to first look (PEEK) to see which pixels within that byte are already foreground colored. PEEK(BYTE) does this for you. The next step is to "OR" PEEK(BYTE) with BIT. Look at an example to help follow along.

Suppose the "x"'s in the pixel pattern below are foreground pixels within a byte:

```
Pixel #:          76543210
Pixel pattern:    x0000x00
```

The computer stores this information using 1's and 0's:

Pixel #: 76543210
Pixel pattern: 10000100

The "x" below is the pixel we want to change to the foreground color:

Pixel #: 76543210
Pixel pattern: 0000x000

The computer stores this as:

Pixel #: 76543210
Pixel pattern: 00001000

The first thing to do is to look at the #0 pixels. If either one of them has a pixel pattern of 1 (meaning foreground colored), then the resulting pixel #0 will have a pixel pattern of 1. This comparison is done for all pixels (#0-#7), until the final pixel pattern is arrived at:

Pixel #: 76543210
Original pattern: 10000100
Pixel to change: 00001000
Final pattern: 10001100

Fortunately, you don't have to go through this process each time you want to plot a point. PEEK(BYTE) OR 21BIT does this for you.

73 POKE CBYTE,C

Line 73 simply changes the color block containing the plotted pixel to the color code in variable C. (Don't forget that 63 other pixels are affected whenever C's value is changed.)

Summary

In this chapter you have learned enough information to actually draw any picture. You can paint the background color (memory locations 24576-32575), and can find and plot any points on the screen. To save time, you can have the computer randomly plot points (if random plotting creates the effect desired). To plot a specific point, you can specify the X (column) and Y (row) locations of the point. In addition, you now know that adding a new program line that changes C's value will change the colors of the block or blocks being plotted in.

Three important things to remember about plotting are:

- (1) X always represents the column location; Y always represents the row location.
- (2) Once a point is plotted (changed to the foreground color), you can only get it back to the background color by changing the program and running it again. There is a subroutine which will “unplot” points, but we are not introducing it in this book.
- (3) You should never try to plot a point using negative (-) X,Y coordinates. You should never try to plot a point greater than 319 for X or greater than 199 for Y.

The next chapter will show you how to save even more time. Instead of finding and plotting each point in a line, you will learn how to plot a line by using only two of its coordinates.

Below are two exercises that use the new information taught in this chapter. Before you try them, be *sure* to first save this chapter's program under “CHAPTER 3”.

Exercise #1

Change line 1310 so that *only* the bottom row of blocks is painted in with the water. Change line 1320 so that only the first 20 columns of blocks is painted in with water. Change line 1330 so that your pixels are plotted 70% of the time instead of 30% of the time.

Solution

To have only the bottom row of blocks plotted, line 1310 should say FOR Y = 192 TO 199. If you look at your graph sheet, you will see that 192 is the first Y coordinate of the last row of blocks.

To have the water stop before the 21st column of blocks, line 1320 should say FOR X = 0 TO 159. Looking at your graph sheet, you will see that the 21st block column has a beginning X coordinate of 160. To stop the water before that point, the next lowest X coordinate of 159 is used.

To have the points plotted 70% (.7) of the time, line 1330 should say IF RND(1) < .7 THEN GOSUB 70.

Exercise #2

This exercise is a little harder, but it produces a nice picture for those of you who can tackle it. Be aware that it will take the computer 30-45 minutes to run through the entire program after you have inserted the exercise lines. If you don't have that much time, wait to try this exercise later.

Begin by loading Chapter 3's program back into memory, and then deleting lines 1210 and 1220 (this saves you the headache of plotting around your lighthouse). Next, make C in line 1110 equal to the color code representing a light blue (BLU2) *foreground* against a black background.

Finally, enter 5 new programming lines (1350, 1360, 1370, 1380 and 1390) which do the following:

Change C's color code again, this time to a white foreground and a black background.

Set Y and X so that they will eventually form the coordinates for all pixels *above* your water.

Plot the points every 100 times (.01) instead of every 40 (.4) times.

Again, this is not an easy one. After you try it, RUN the program. As the program is running, you may want to take a nap, watch TV, or visit a friend. In 30-45 minutes, return to your Commodore to see the surprise!

Solution

Line 1110 should have C = 224. The new lines to add are as follows:

```
1350 C = 16
1360 FOR Y = 0 TO 175
1370 FOR X = 0 TO 319
1380 IF RND(1) < .01 THEN GOSUB 70
1390 NEXT X: NEXT Y
```

Leaving lines 1300-1340 untouched, your water will be drawn in at the bottom of your screen. However, because C was changed in line 1110, the water will be mainly black—because black is now the background color.

Line 1350 again changes C, this time to white against black. As each new point is plotted, line 73 POKE's C's color code into the memory location controlling the colors of the block that the point is plotted in.

Your water begins in row 176. To keep from plotting new points in your water, line 1360 has Y stop at 175. Line 1370 keeps X at 0 through 319.

Line 1380 has the X,Y coordinates plotted 1/100 of the time instead of every 1/3.

Chapter 4

PLOTTING LINES AND PAINTING SHAPES

So far, you have learned how to:

- turn on high resolution graphics
- change the color codes for each block on the screen
- turn all of the screen's pixels to their background color
- plot specific points anywhere on the screen
- plot random points anywhere on the screen
- change the colors of a block being plotted in
- return to text mode at any time
- ZAP the main routine to draw a new picture

That's quite an accomplishment! Using your subroutine tools, you have even created part of a picture—the sky, a lighthouse, and an open sea. But what about that lighthouse? In a true-to-life situation, the lighthouse would never be suspended in air. In this chapter, you will correct this situation. Using a PLOT A LINE tool, you will draw the outline of some land and waves. Using a PAINT A SHAPE tool, you will paint in the colors for the land and waves. Before beginning, let's first discuss some principles of designing foreground images for your Commodore 64.

Designing Foreground Images

Whenever you want to paint a picture on your Commodore 64, you should always sketch the picture on graph paper first. Draw your large box that is 40 small blocks across and 25 small blocks high. Sketch your picture within this large box. Next, using highlighting pens or colored pencils, shade in the colors to be used in the picture. It is important that you do not use heavy markers, as they will cover up the small graph blocks. Finally, look at each block in the picture and check for any which shows more than two colors. Remember, each block can have one foreground color and one background color. If any of the blocks show more than two colors, you will need to adjust the picture in some way. This can be done by moving a figure up, down, right or left; redrawing a small portion of a figure; or deciding to use fewer colors in that area of your picture.

Another consideration is that two foreground colors cannot share the same block. To paint foreground shapes, every pixel in the shape is plotted. Plotting a pixel changes it to the foreground color of the block the pixel is located in. Now, suppose you have two images (or two colors within one image) which need to be plotted side-by-side. If those foreground colors fall within the same block, there is no way they can both be plotted. You would have to paint that block by *itself*—as if

it were a whole shape. In the main routine, you would make C = to the two colors needed for that block (e.g., C = 94). Then, only the portion of the block which used the foreground color would be painted (plotted). This will become clear when you draw your own pictures later.

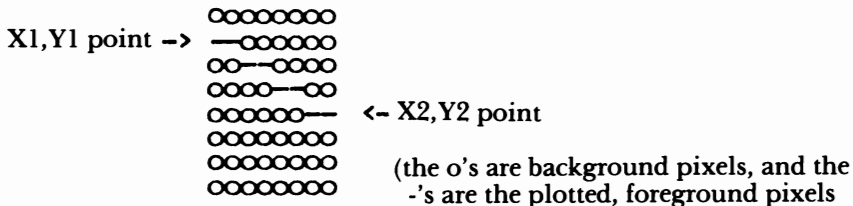
Once you have a finished, usable sketch, you can use the subroutine tools to enter high resolution graphics, change the screen's colors, and paint the background color of your picture. At that point, you will be ready to begin plotting points, lines and shapes.

Plotting Lines

In Chapter 3 you learned that to plot a single point, you have to identify its X,Y coordinates in the main routine. For example, the X,Y coordinates of the pixel in the top right corner of your screen are 319,0. When points are identified by an X,Y notation, the X coordinate is *always* given before the Y coordinate (X,Y). It would be incorrect to identify the top right pixel as 0,319. This would be read as the point in column (X) 0, row (Y) 319. Looking at your X,Y PIXEL POINTS sheet, you will see that there is no Y coordinate of 319. Keep this idea of "X before Y" in mind as we discuss plotting lines.

To plot lines, all you need to know is the starting point of the line (X1,Y1), and the ending point of the line (X2,Y2). For example, suppose you wanted to draw a line from point 10,20 to point 50,70. This line is easily handled by a PLOT A LINE tool and a main routine line stating that X1=10: Y1=20: X2=50: Y2=70. The 1's and 2's are very important. They tell the computer which X coordinate goes with which Y coordinate to form the starting or ending point of the line.

When the computer draws the line, it *always* begins at the X1,Y1 point and plots it (turns it to the foreground color). Next, it very quickly determines the shortest, straightest route to the X2,Y2 point. In many cases, this route will not be a truly straight line. To see why, let's blow up an 8 x 8 pixel block and look at a line plotted through it:



Because there was no straight path from point X1,Y1 to point X2,Y2, this line had to be plotted in a zig-zag pattern. The pattern used was "plot two, down a pixel, over a pixel; plot two, down a pixel, over a pixel, etc." The PLOT A LINE tool will figure out the necessary pattern and plot the line for you.

To learn more about plotting `linex`, load Chapter 3's program into the computer's memory. Run the ZAP routine (type `RUN 10` and press **RETURN**) so you can enter a practice program. Now, type the `PLOT A LINE` tool as shown below:

```
80 REM::::::::::PLOT A LINE
81 DX = X2 - X1: DY = Y2 - Y1
82 L = ABS(DX): IF ABS(DY) > L THEN L = ABS(DY)
83 IF L > 0 THEN XI = DX/L: YI = DY/L
84 X = X1 + .5 : Y = Y1 + .5
85 FOR I = 0 TO L
86 GOSUB 70 : REM PLOT POINT
87 X = X + XI: Y = Y + YI
88 NEXT I
89 RETURN
```

Look over the lines you just typed. As always, carefully compare your typed lines to those shown above. When a program doesn't work properly, it is often difficult to determine if the problem lies in the main routine or in a subroutine. Taking the extra time to go over each line now can save you a headache or two down the road.

To start out simple, you will plot a line from the top, left corner of the screen down to the bottom, right corner of the screen. This diagonal line will be easy to see, and it will be easy to determine if you've plotted it off course. As mentioned above, plotting a line involves an `X1,Y1` coordinate, and an `X2,Y2` coordinate. Using your `X,Y PIXEL POINTS` graph sheet, see if you can figure out the coordinates necessary to plot this diagonal line. You will find that `X1,Y1` should be `0,0` (`X1=0,Y1=0`) and `X2,Y2` should be `319,199` (`X2=319,Y2=199`). Type the main routine lines shown below, noting how these coordinates have been used:

```
1100 GOSUB 20      : REM GRAPHICS
1110 C = 1: GOSUB 40: REM COLORS
1120 GOSUB 50      : REM PAINT BKGROUND
2100 REM::::::::::LINE TEST
2110 X1 = 0: Y1 = 0
2120 X2 = 319: Y2 = 199: GOSUB 80
2150 END
```

Before running this program, check lines 2110 and 2120. Be sure that the X's (`X1` and `X2`) are 0 and 319. Be sure the Y's (`Y1` and `Y2`) are 0 and 199. Plotting an X coordinate outside of the 0-319 range, or plotting a Y coordinate outside of the 0-199 range, could cause you problems. By "problems," we mean the computer could stop operating. You would then have to turn it off and on again to clear its memory of the badly plotted point. This, of course, also clears your program from memory. If you want to memorize any of this book's material, *memorize the X,Y coordinate ranges*.

RUN the new program.

When the program starts, all background pixels will change to white. Next, all foreground pixels will be changed to background pixels (making them white also). Finally, your diagonal line will be plotted from the top, left corner to the bottom, right corner. Watch the zig-zag pattern which appears as the line is plotted. There is no straight path from point 0,0 to point 319,199. Isn't it nice, though, that the subroutine figures out the straightest, quickest path for you?

A consequence of changing the screen's colors in line 1110 is that you now have a light blue border surrounding the high resolution picture area. You may not have known it, but this border has *always* been on your screen. It surrounds the high resolution picture area, but is not a part of it. In fact, there is no way you can draw on it or get rid of it. The border has not shown up in the past because your screen's background color has always been light blue also. Thus, the picture area and the border blended together to appear as one solid screen. You can change the color of this border by poking a color code into memory location 53280. Note, however, that the background color of the color code is used on the border, and the foreground color is ignored.

Watch as the computer plots the large diagonal line from the top, right corner of the white box to the bottom, right corner. If the line is being plotted anywhere else, or if the line is not being plotted at all (wait a few minutes to be sure), press **RUN/STOP** and tap **RESTORE** at the same time. Check to make sure that C = 1 in line 1110. Sometimes, you can accidentally make the screen's foreground and background color the same. This would mean that even though the computer is plotting your line, it won't be visible on the screen. For example, had you typed C = 17 in line 1110, it would have *appeared* as though the line never got plotted. This is because 17 is a color code representing white against white. Plotting a white line on a white background color makes everything blend together. The line would not stand out against the background color of the screen.

Make sure your program is running properly before continuing. If the main routine lines are correct, but you did not get a plotted line, check the subroutine at line 80. This subroutine has to be entered correctly before you can go on to the next practice session.

When everything checks out, and the diagonal line has been plotted, press **RUN/STOP** and tap **RESTORE**. Look at your main routine. Line 1100 turns on the high resolution screen so you can see the line being drawn. Line 1110 paints in the foreground/background colors on the screen (black against white). Line 1120 changes all the pixels to the background color of white. (You've seen these lines before.)

Line 2110 gives the starting coordinate for the line (0,0). Line 2120 gives the ending coordinate for the line (319,199) and also sends the computer to the PLOT A LINE tool. Once you have given the computer the X1,Y1 and X2,Y2 coordinates of a line to be plotted, all you have to do is send it to the new tool. Line 2150 "ends" the program. This line is necessary only if you want the computer to stop processing the program at a certain line number. If there are no other line numbers below the END statement, the END statement is not necessary. Right now, there are no other

line numbers below the END statement. You will be adding some more in a moment, though, and the purpose of this END statement will then be explained.

For your next practice, we will tell you where to plot two lines. *You* will figure out the main routine lines necessary to plot them. Ready? Change lines 2110 and 2120 to plot a line across the very top row of your screen. In addition, add lines 2130 and 2140 to plot a line down the far, right edge of your screen. Make sure you do not try to plot these lines outside of the 0-319 range for X's, and 0-199 range for Y's.

When you think you have it right, run the program again. The two lines should be plotted in black across the top and down the right side of the screen. Use **RUN/STOP** and **RESTORE** to leave high resolution if you are having any problems. If the computer "freezes" up on you, and **RUN/STOP RESTORE** does not help, you probably plotted outside of the X and Y ranges. The only solution is to turn the computer off and on, load Chapter 3's program, and re-enter the PLOT A LINE tool and main routine lines.

If the lines were plotted correctly, you correctly entered the new program lines as:

```
2110 X1 = 0: Y1 = 0
2120 X2 = 319: Y2 = 0: GOSUB 80
2130 X1 = 319: Y1 = 0
2140 X2 = 319: Y2 = 199: GOSUB 80
```

Remember that after each new set of X1,Y1 and X2,Y2 coordinates, a GOSUB 80 must be inserted to have the line plotted. Since you needed to plot two lines, you needed two GOSUB 80 statements.

Line 2110 says that X1 = 0 and Y1 = 0. This is point 0,0, or the top, left pixel on your screen. Line 2120 says that X2 = 319 and Y2 = 0. This is the top, right pixel on the screen. The GOSUB 80 causes a line to be plotted between these two pixel points. Lines 2130 and 2140 plot the line between the top, right corner and the bottom, right corner. Plotting lines is really that simple.

Notice that in lines 2120 and 2130, the X,Y coordinates stayed the same. The only difference is that in line 2120, you were using X2,Y2, and in line 2130, you were using X1,Y1. In the next practice session, you will learn how to use the same coordinate more than once, without re-typing it.

Take some time now to plot your own lines. Begin by drawing a line on your X,Y PIXEL POINTS sheet. At one end of the line (it doesn't matter which end), write down the X and Y locations of that starting point (X1=?,Y1=?). At the other end, write down the X and Y locations of the ending point (X2=?,Y2=?). Enter these new coordinates in the LINE TEST routine, keeping within the X and Y ranges. RUN the program again. Compare the new line on your screen to the line you drew on the graph sheet.

One way to plot the lines in this outline shape would be:

(Don't type these program lines)

```

3100 REM:::::SHAPE TEST
3110 X1 = 72: Y1 = 71
3120 X2 = 88: Y2 = 40: GOSUB 80
3130 X1 = 88: Y1 = 40
3140 X2 = 103: Y2 = 71: GOSUB 80
3150 X1 = 103: Y1 = 71
3160 X2 = 72: Y2 = 71: GOSUB 80
    
```

This would work, but look how often you use a previous coordinate to draw the next line. A shorter way to accomplish the same thing is:

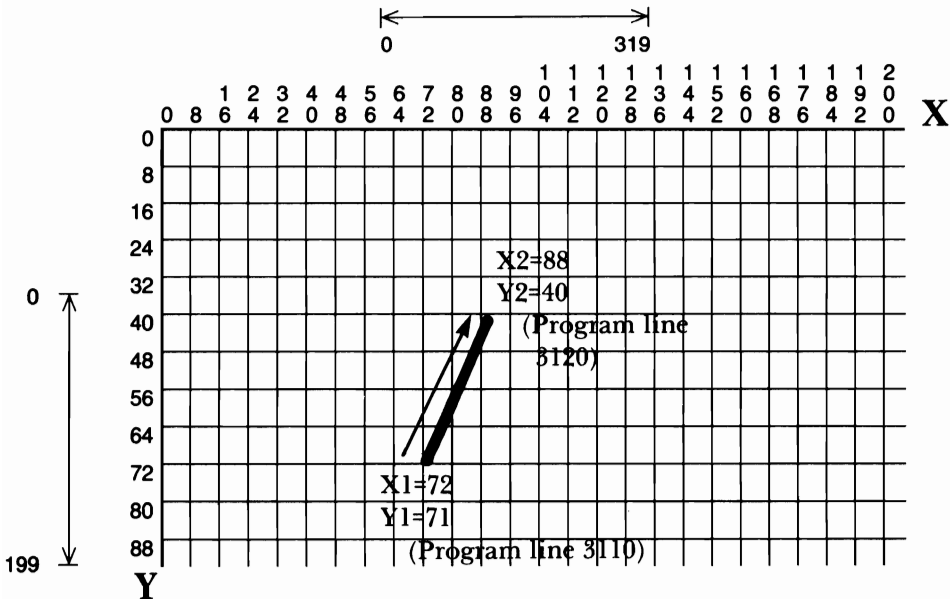
```

3100 REM:::::SHAPE TEST
3110 X1 = 72: Y1 = 71
3120 X2 = 88: Y2 = 40: GOSUB 80
3130 X1 = 103: Y1 = 71: GOSUB 80
3140 X2 = 72: Y2 = 71: GOSUB 80
    
```

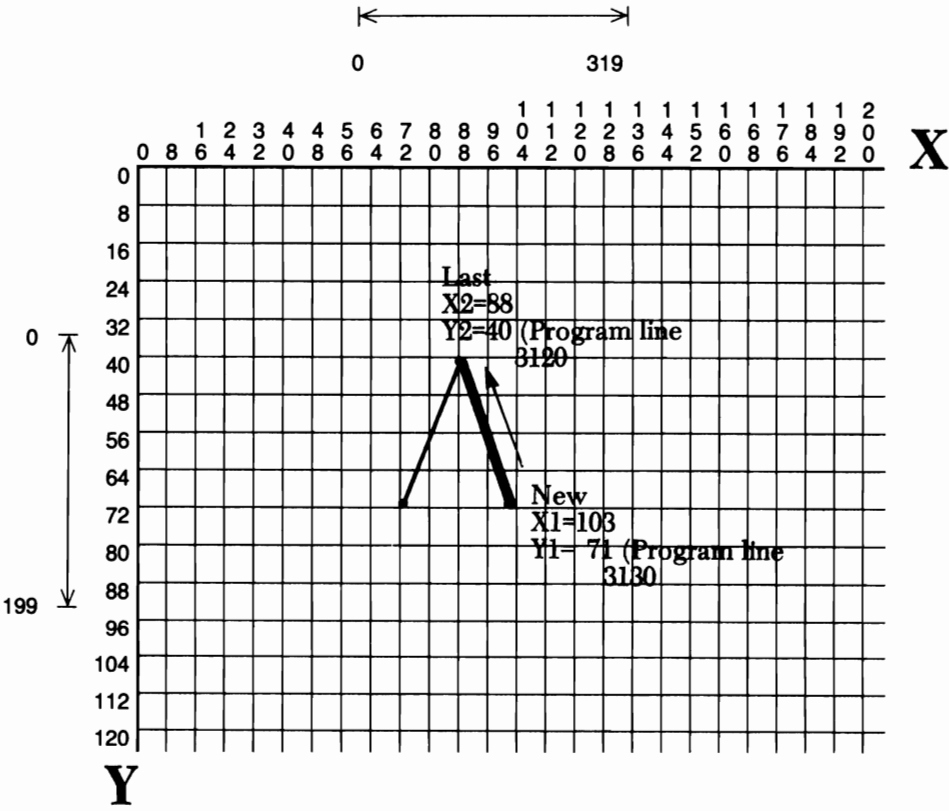
This second method requires only 5 program lines instead of 8, but how does it work?

When the computer is sent to the PLOT A LINE tool, it always plots a line between the *last* X1,Y1 coordinate it was given and the *last* X2,Y2 coordinate it was given. It doesn't matter if one of those coordinates has already been used in the past.

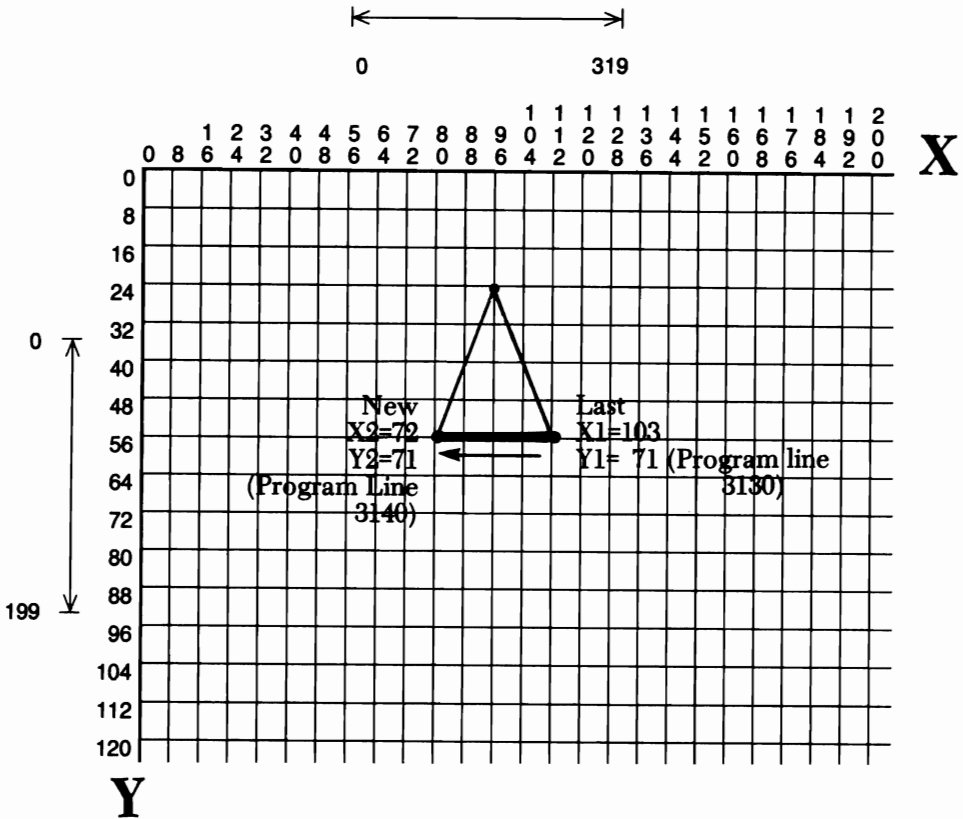
Looking at the lines above, lines 3110 and 3120 plot the first line:



When the computer reaches line 3130, it sees that only the X1,Y1 coordinates have changed. It goes to the PLOT A LINE tool and plots from the new X1,Y1 coordinate (103,71) to the old X2,Y2 coordinate (88,40):



Line 3140 changes X2 and Y2. Using the last X1,Y1 coordinate given in the program (103,71) and the new X2,Y2 coordinate in the program (72,71), the final line is plotted:

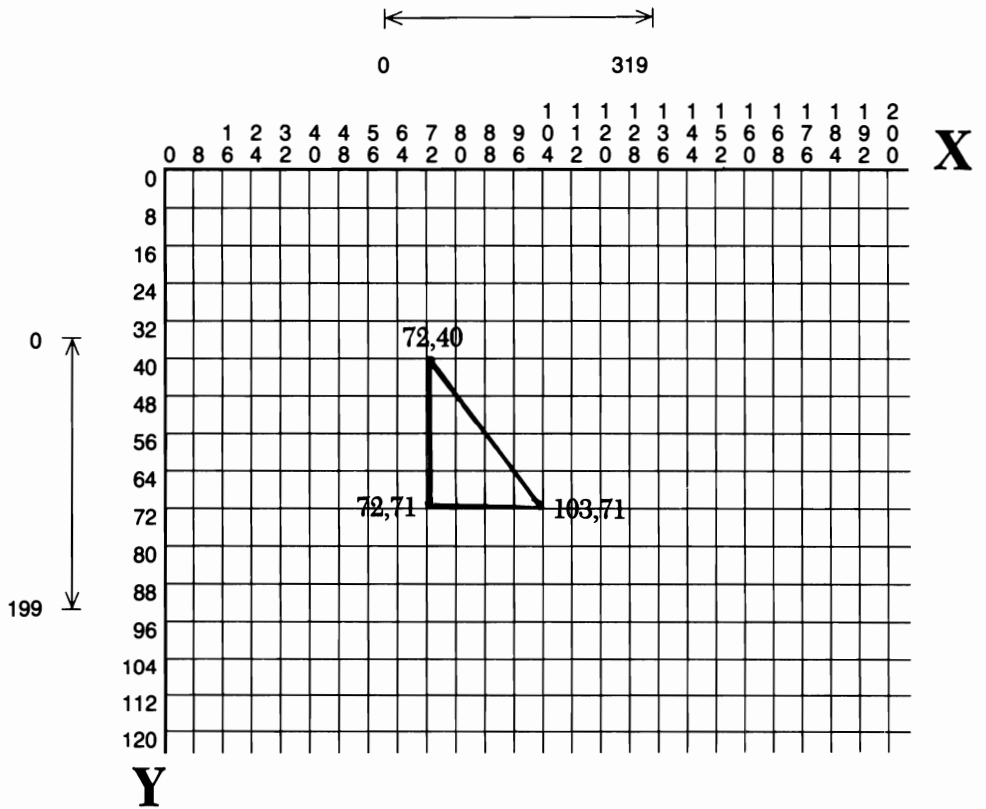


As long as the computer has an X1,Y1 and an X2,Y2 pair, it will plot a line. The line will be plotted from the last X1,Y1 point given, to the last X2,Y2 point given (provided, of course, that you follow this with GOSUB 80).

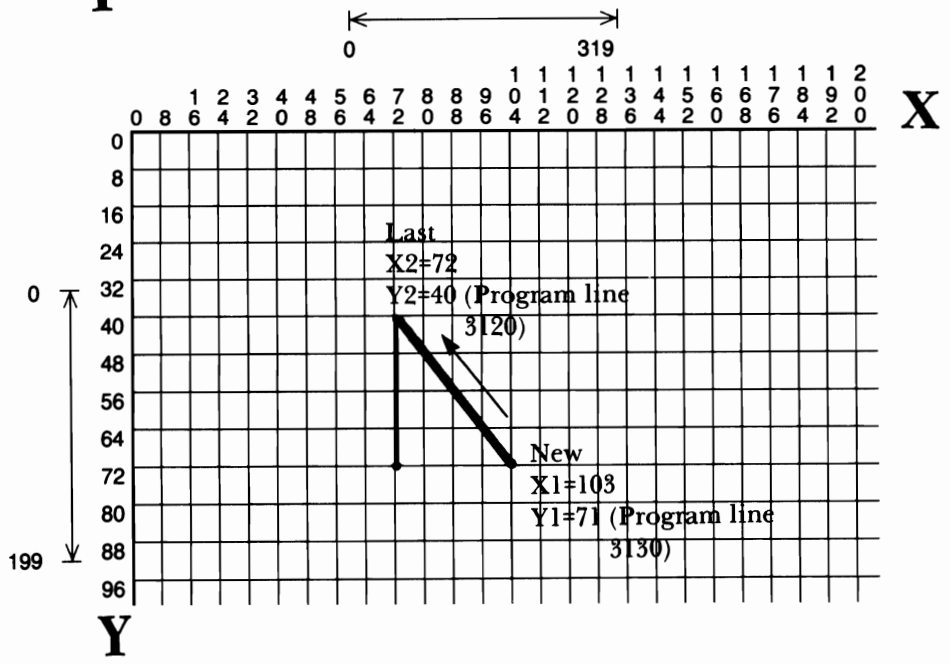
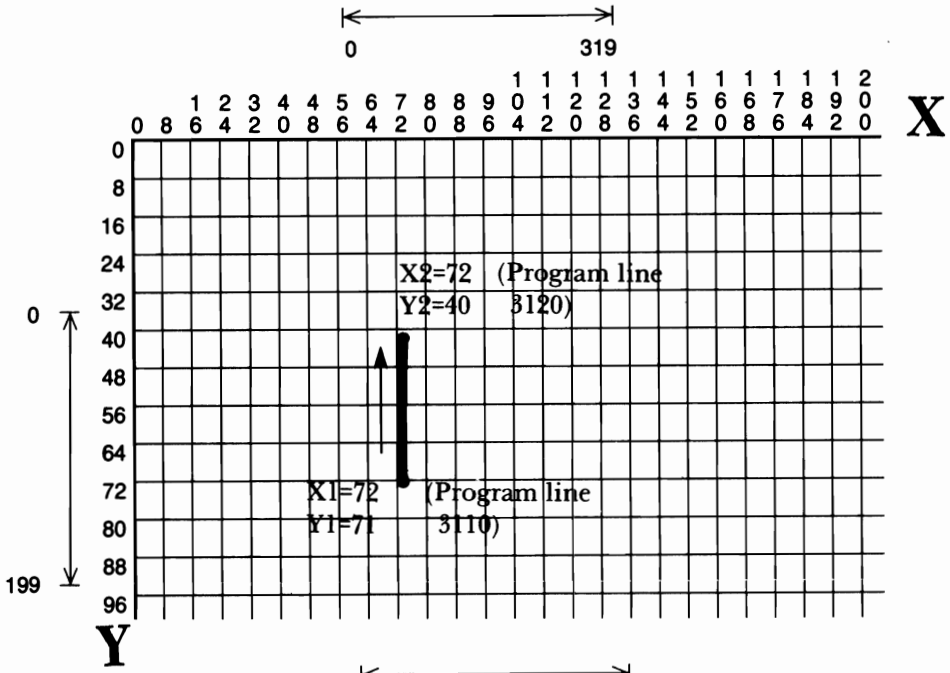
Type lines 3100 through 3140, as previously listed here. Add line 1130 GOTO 3100 (this has the computer by-pass the LINE EST when the program is running). RUN the program. Notice that the triangle is not drawn in one continuous motion. This is because each line is plotted *from* point X1,Y1 *to* point X2,Y2. Look back at the diagrams on this triangle to see these X,Y locations.

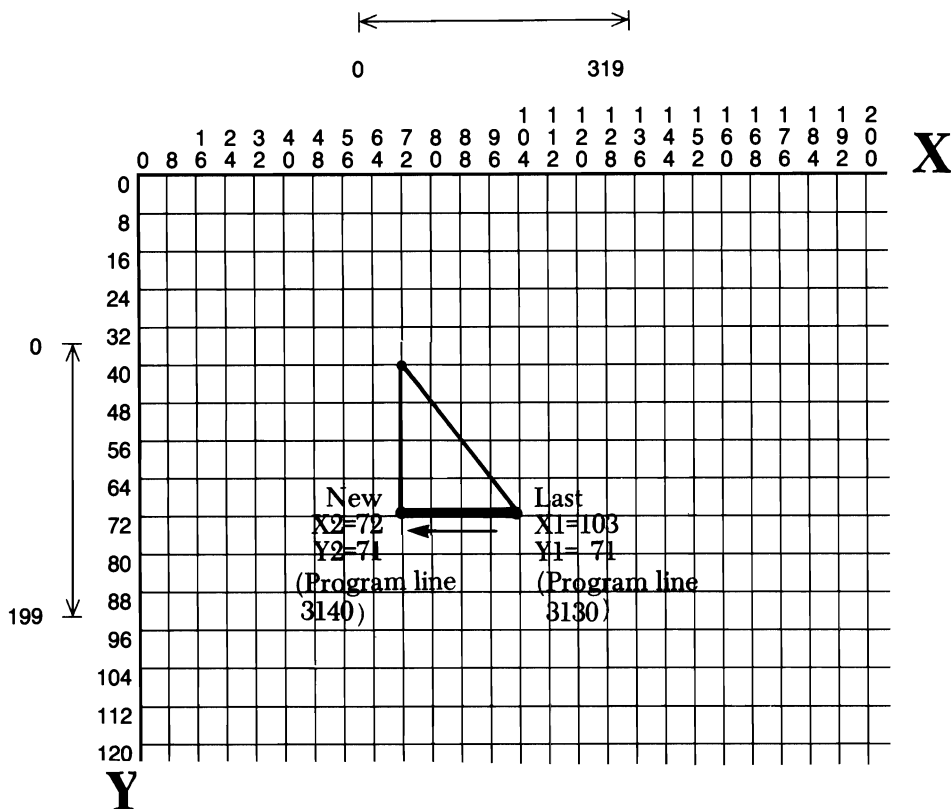
When the triangle is finished, stop the program and return to the program listing. Try changing the main routine to plot the right (Δ) triangle shown below.

The solution is discussed in the next paragraph, but try it on your own first. RUN your solution to see how it worked.



To make the right triangle, only line 3120 needed to be changed. It should now have 3120 X2 = 72: Y2 = 40. This adjustment changes the ending point of the first plotted line, and the beginning point of the second plotted line. Look at the three diagrams below to see how this worked:





Keep these new program lines in your main routine. The next section deals with painting shapes, and you will use this triangle as the first shape to be painted.

To practice drawing shapes on your own, change line 1130 to GOTO 4100. Beginning at line 4100, enter the new program lines for your new shapes. If a line gets plotted incorrectly in a shape, take the following steps:

- (1) List the program on your screen;
- (2) Begin at the GOSUB 80 statement which caused the incorrect line to be plotted;
- (3) *Backtrack* through your program, writing down the *last* X1, Y1, X2, and Y2 coordinates given before this GOSUB 80;
- (4) Compare these coordinates (X1,Y1 and X2,Y2) to the coordinates your graph sheet shows for the line.

To practice drawing a single line instead, delete line 1130 entirely. This causes the computer to go directly to the LINE TEST routine when you type RUN. The END statement in line 2150 stops the computer from going through the SHAPE TEST routine again. The use of END and GOTO statements provides an easy way to test certain portions of your program without re-running the program in its entirety.

Painting Shapes

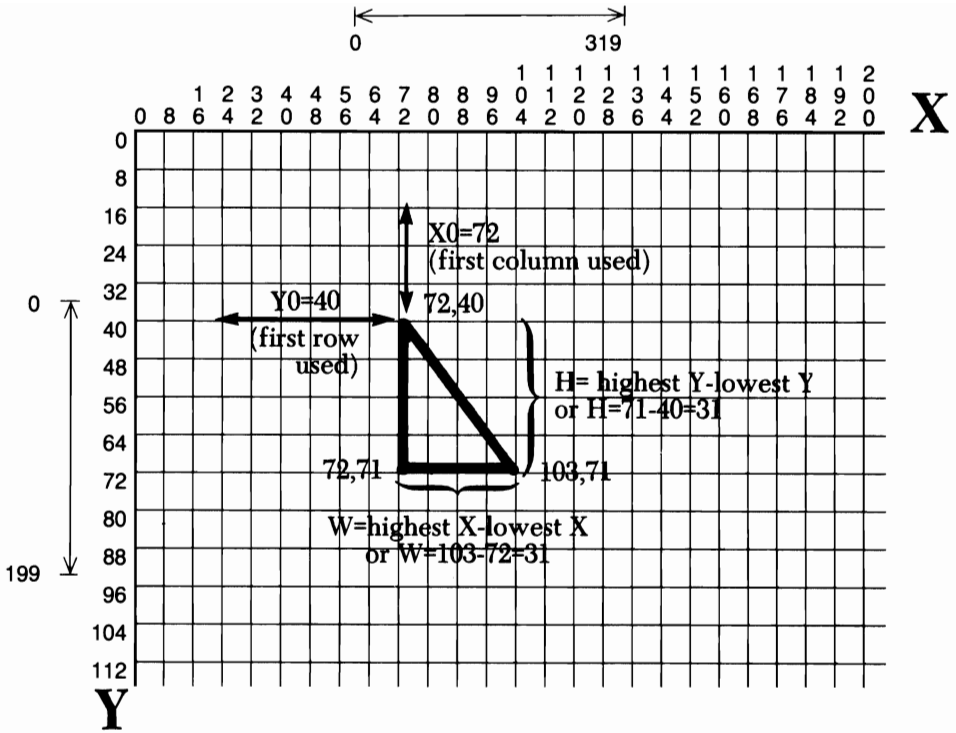
Painting shapes is one of the more rewarding parts of computer graphics. As colors are painted in, your picture will become more life-like, as well as more attractive, to anyone viewing it. Type the PAINT A SHAPE tool shown below.

```
90 REM:::::PAINT A SHAPE
91 PC = PC + ABS(PC=0): FOR X = X0 TO X0 + W: FL$ = "F": PR = 0
92 FOR YC = Y0 TO Y0 + H: Y = YC: GOSUB 60
93 ON ABS((PEEK(BYTE) AND 2 ^ BIT) <> 0) GOTO 97: IF PR=0 THEN 96
94 PR = 0: IF FL$ = "F" THEN Y1 = YC: FL$ = "T": GOTO 96
95 GOSUB 99: FL$ = "F"
96 NEXT YC: GOTO 98
97 PR = 1: NEXT YC: IF FL$ = "T" THEN GOSUB 99
98 NEXT X: RETURN
99 FOR Y = Y1 TO YC -1: ON ABS(RND(1) < PC) GOSUB 70: NEXT Y: RETURN
```

To use this new tool, you need a GOSUB 90 statement. Before GOSUB 90, though, you also need main routine lines which contain the following:

- (1) The *first* X column your shape falls in (X0=?).
- (2) The *first* Y row your shape falls in (Y0=?).
- (3) The width of your shape (W=?). Note: The width starts at 0. So, if your shape takes up 8 pixel columns, then W=7. You will later find that a 0-based width is easier to figure out than the true width.
- (4) The height of your shape (H=?). Note: The height starts at 0. So, if your shape takes up 10 pixel rows, the H=9. You will later find that a 0-based height is easier to figure out than the true height.
- (5) The percentage of pixels you want painted within the shape (PC=?). Note: This percentage is expressed in a fraction. Thus, make PC=.9 to randomly paint 90% of the shape; make PC=.6 to randomly paint 60% of the shape; and make PC=1 to paint 100% of the shape. All unpainted pixels will be background colored.

Let's see how this would work with your right triangle:



- (1) The first column your triangle uses is 72. Thus, $X0=72$.
- (2) The first row your triangle uses is 40. Thus, $Y0=40$.
- (3) To determine the width, subtract the *lowest X* coordinate the shape uses *from* the *highest X* coordinate the shape uses. In your triangle, this is $103 - 72$, or $W=31$.
- (4) To determine the height of your shape, subtract the *lowest Y* coordinate the shape uses *from* the *highest Y* coordinate the shape uses. In your triangle, this is $71 - 40$, or $H=31$.
- (5) We will have $PC=1$ so that the triangle is completely painted in.

Type the program lines below that will paint your outline shape:

```

4100 REM:::::PAINT SHAPE TEST
4110 X0 = 72: Y0 = 40
4120 W = 31: H = 31
4130 PC = 1: GOSUB 90

```

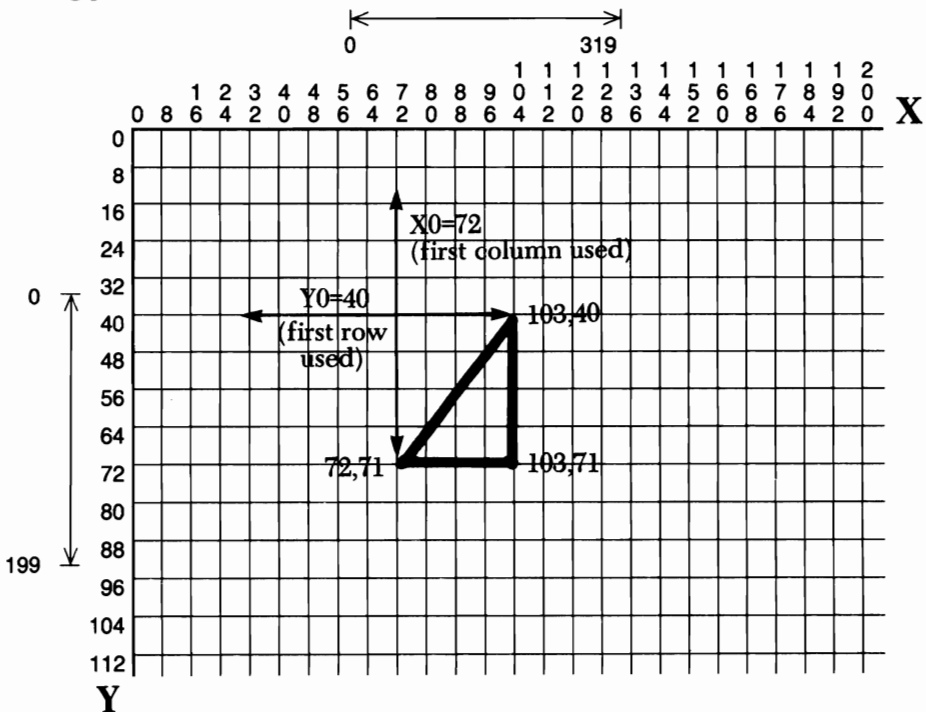
That's all it takes. Before running the program, change the color the triangle will be plotted in. The best way to do this is to change *C* *before* the outline of the triangle is plotted. Add line 3000 $C = 33$ (this is red plotted on white). Changing *C* at line 3000 ensures that every pixel plotted from line 3000 on will be plotted in the new foreground color. Finally, add line 1130 $GOTO 3000$ so that this new color is used by the computer. **RUN** the program.

Wait while the screen is cleared and changed to white background pixels. (This will take a few moments.) Next, the triangle will be outlined in red. Finally, the entire shape is painted in. Watch carefully how the triangle is painted. Starting at the triangle's first row ($Y0=40$) and first column ($X0=72$), the computer plots down the column within the outline shape. When this column is completed, the next column (73) is plotted from top to bottom within the outline. This process is continued down each column, across the shape.

If your triangle is not being painted properly, stop the program and check the new program lines. The problem will probably be in the PAINT A SHAPE tool, so give particular attention to those lines.

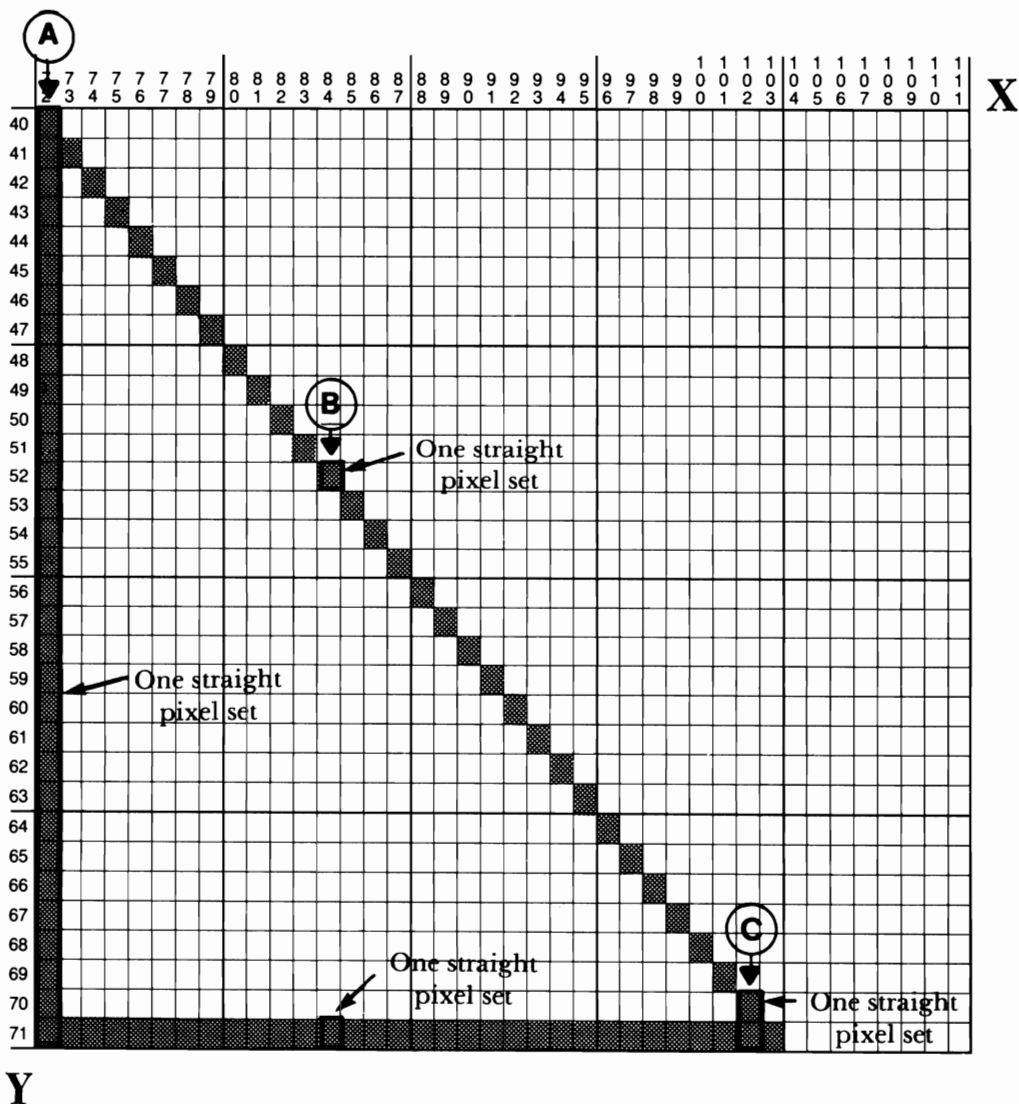
It is very important to understand how the PAINT A SHAPE subroutine works, primarily because it cannot properly paint every shape of every kind. Read the rest of this section carefully, or you could be in for some unpleasant surprises when you later try to paint your own shapes.

To paint a shape, the computer begins at column $X0$ and row $Y0$ (or point $X0,Y0$) established in your main routine. This pixel location tells the computer the first column and first row that your outline shape uses. Note, however, that this pixel point is *not* necessarily a part of the outline shape. For example, suppose your right triangle were reversed, as shown below. This reversed triangle still uses the same first column ($X0=72$) and the same first row ($Y0=40$), but the point at $72,40$ is not a part of this triangle's outline. The computer uses the $X0,Y0$ point merely as a starting position.



The shape is painted column-by-column, starting at point X0,Y0 down the length of your shape (H=31), and extending across the columns that make up the width of your shape (W=31). However, plotting only occurs in the columns that have *at least two* straight sets of outline pixels.

“Outline pixels” are the plotted, foreground pixels which make up the outline shape. Each plotted line in the shape is made up of these outline pixels. A “straight set” is one or more outline pixels appearing *one right after the other* down a column within your shape. Look at a blow up of your triangle’s outline to see what we mean:

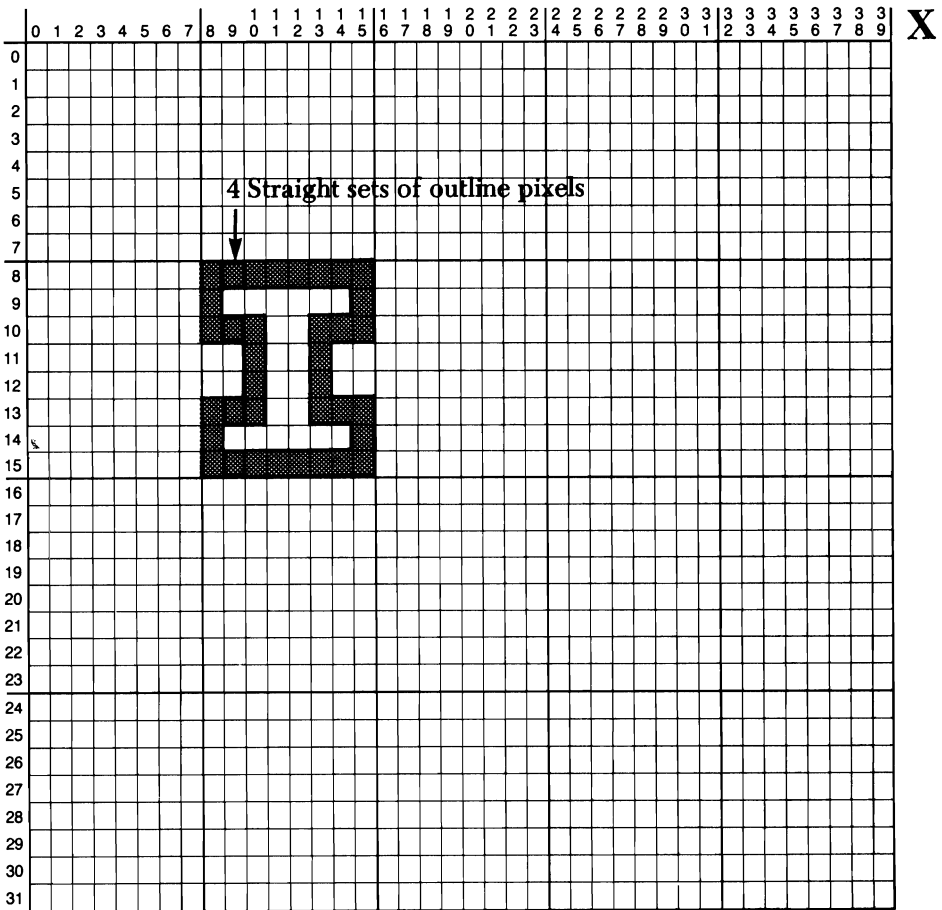


The first column in your triangle (A) has only *one* straight set of outline pixels, and so it is skipped over by the subroutine. (Because columns can be skipped over by the subroutine, it is important to plot your outlines in the same color you plan to later paint them in with.)

Column B in the diagram has two straight sets of outline pixels. (Again, a "straight set" can be made up of only one outline pixel.) When this happens, the subroutine plots each pixel between the two sets.

Column C contains only one straight set of outline pixels and is thus skipped over the subroutine.

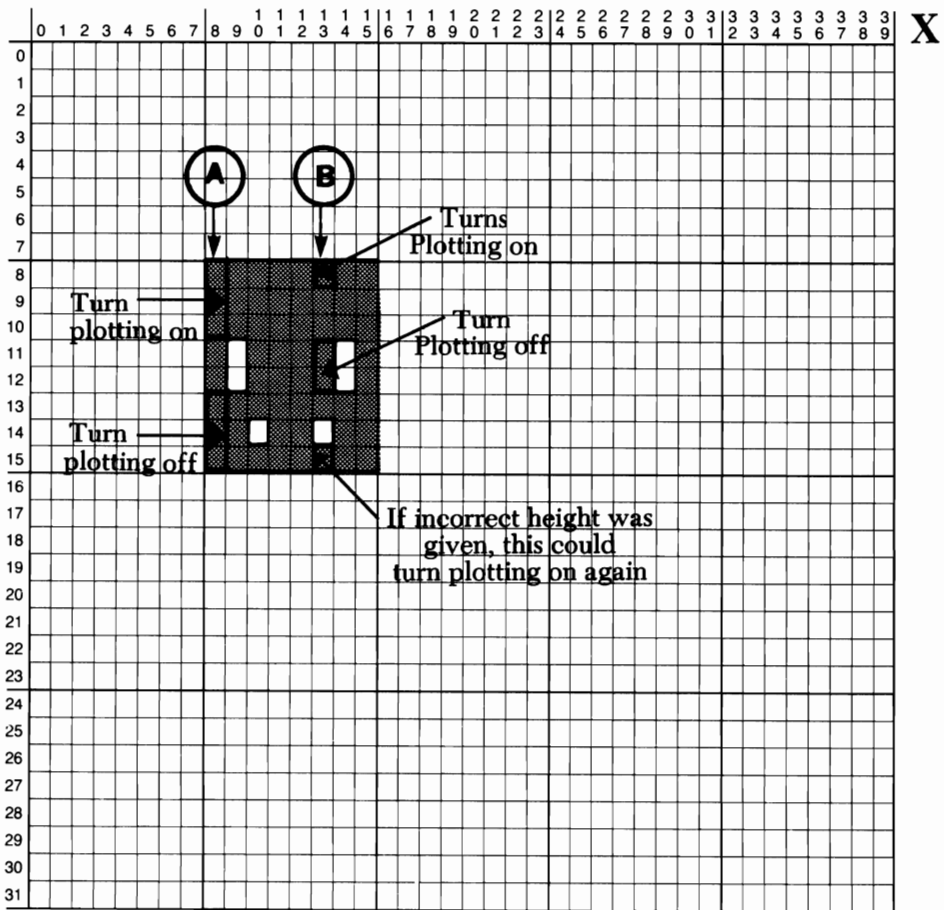
Sometimes, a column within a shape will have more than two straight sets of outline pixels. Look at the shape shown below for example:



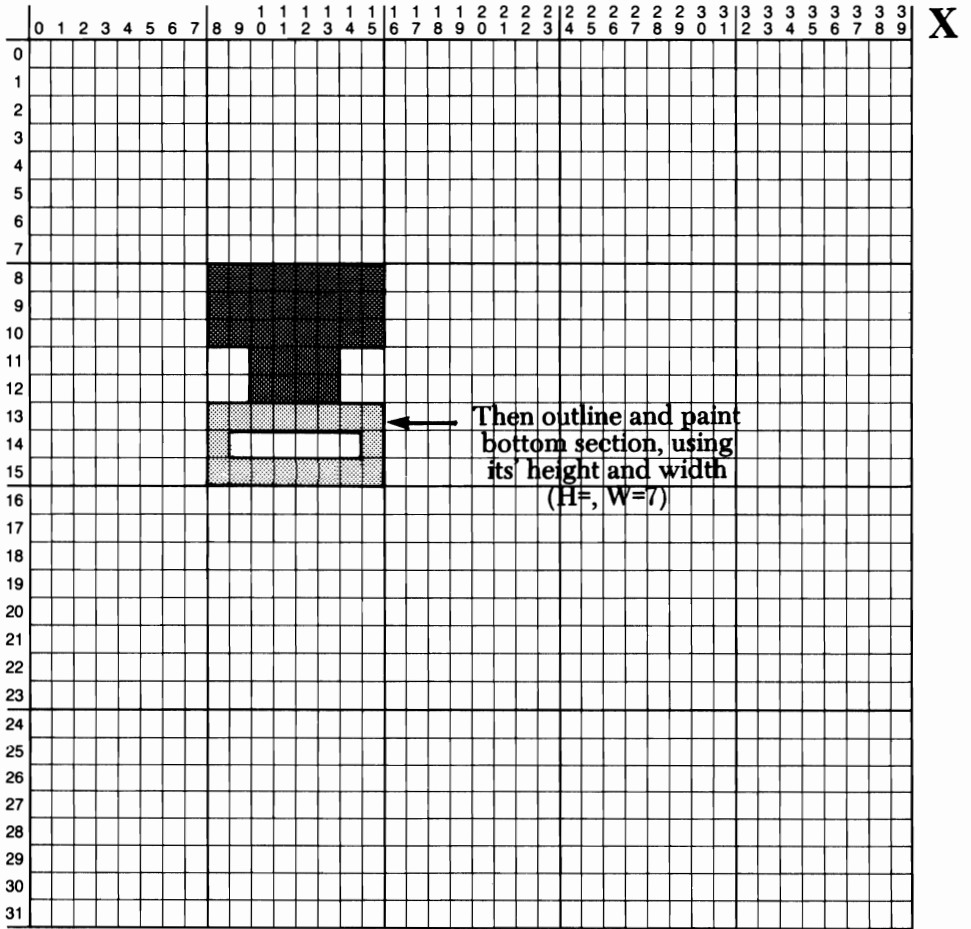
In the diagram above, we have pointed out a column which contains four straight sets of outline pixels. The subroutine uses each straight set as an ON/OFF plotting switch. When the first set is passed through, the computer starts plotting the pixels beneath it. When the second set is passed through, the computer stops plotting. When the third set is passed through, plotting begins again, and at the fourth set, plotting stops. If there were no fourth set, plotting would stop at the bottom row of the shape as determined by H=? in the main routine.

This ON/OFF plotting switch is used in every column. Even in the columns containing two straight sets of outline pixels. The computer begins plotting beneath the first set, and quits plotting when it reaches the second set. This is the most important, and perhaps hardest, idea to learn about this new tool. *Each straight set of outline pixels in a column is a plotting switch.*

Were we to plot and paint the above shape, the outcome would be:







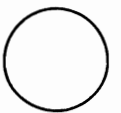

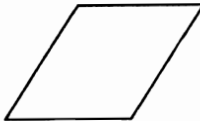





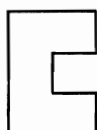

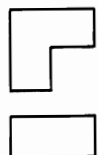



Y

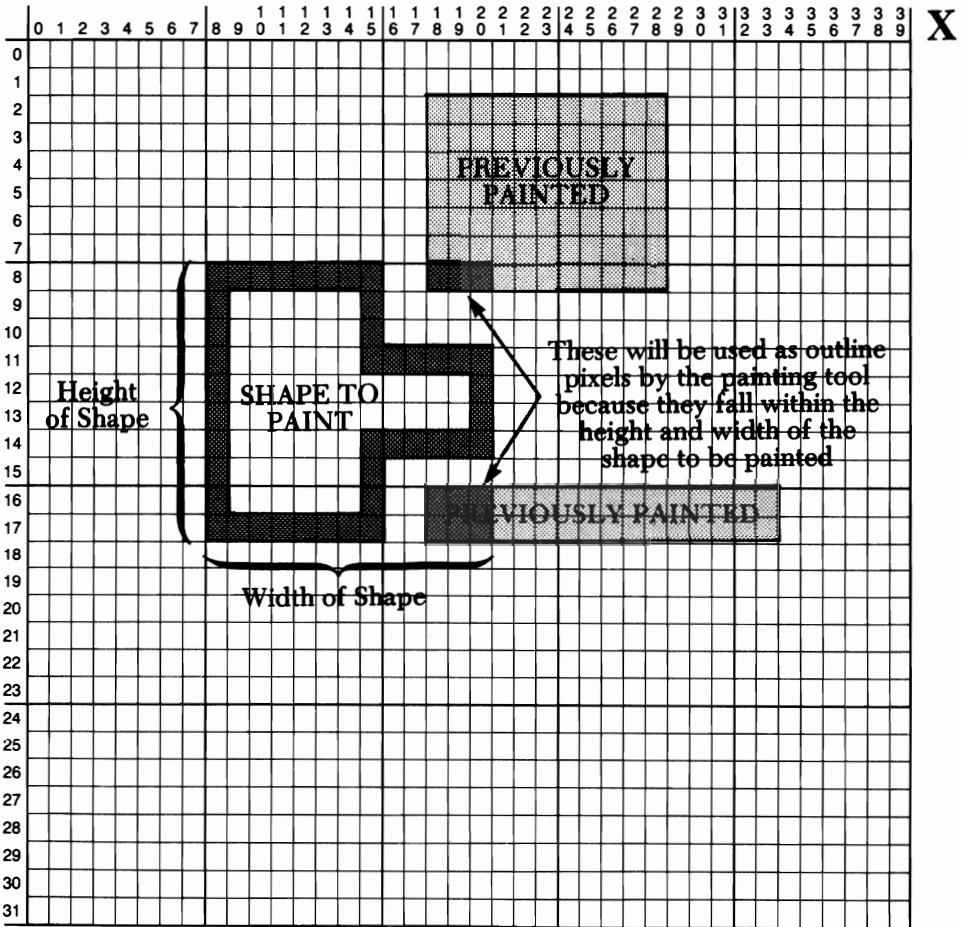


Y

Below is a chart showing common shapes that will work “as is” with the PAINT A SHAPE tool. Also shown are shapes that need to be painted in sections in order for the PAINT A SHAPE tool to work.

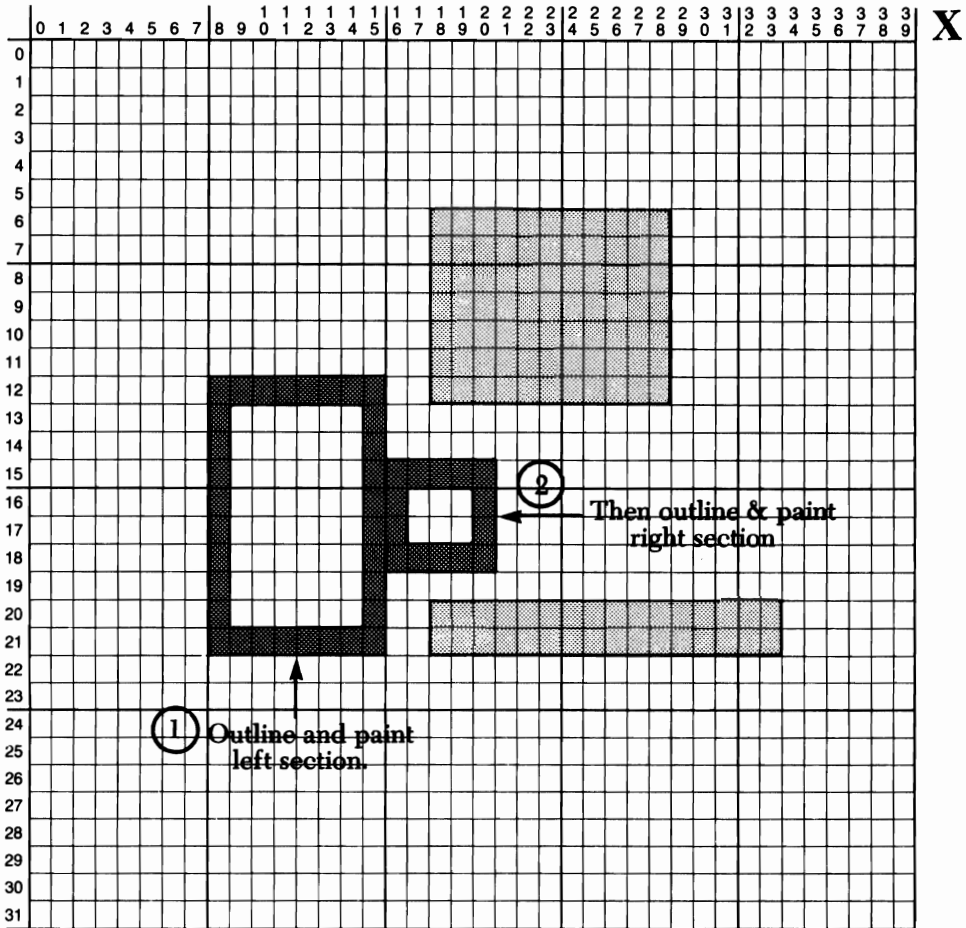
Outline Shape	Painted Shape	Alternative Way to Paint		
		Not necessary.		
		Not necessary.		
		Not necessary.		
		Not necessary.		
		Not necessary.		
		Plot & paint top half: then plot & paint bottom half:		
		Plot & paint top half: then plot & paint bottom half:		

As a final comment on painting shapes, remember that this tool starts at row Y0 when plotting each column. It moves down the column, and begins plotting beneath the first straight set of outline pixels it comes to. Consider the diagram shown below. On it, there is the shape to be painted, as well as two previously painted shapes.



Y

The two previously painted shapes fall within the columns of the shape to be painted. Their foreground pixels will be considered outline pixels by the PAINT A SHAPE tool. As outline pixels, they turn plotting on and off in each column they fall in. To tackle this problem, you again divide up the shape to be painted:



As a checklist, search your sketched pictures for:

- (a) 3 colors within one block;
- (b) 2 foreground colors within one block; and
- (c) Straight sets of outline pixels which will start or stop plotting when painting a shape.

Any shape can be outlined and painted using your new tools. By carefully studying your sketched picture, you can determine ahead of time those shapes which need to be outlined and painted in sections. When you enter the program, be sure to use the color you plan to paint with to *first* outline each shape.

If you have the time, try outlining and painting some of your own shapes. This effort will be well worth your time. Understanding how the PAINT A SHAPE tool uses the foreground pixels in each column of each shape is an important concern in designing your own pictures.

Outlining the Land and Waves

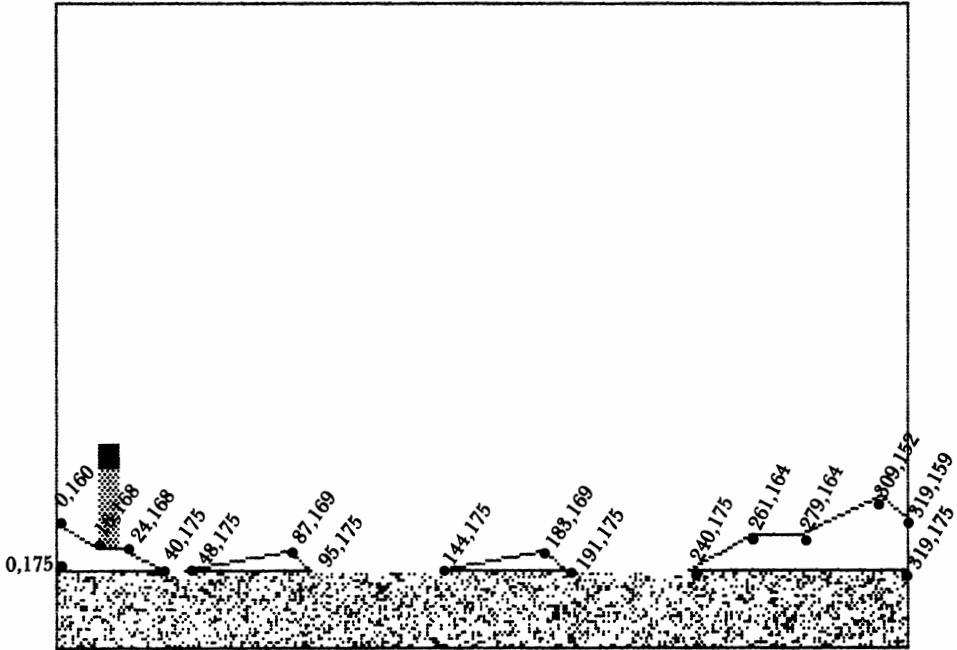
To add an outline of some land and waves, you need to first re-type Chapter 3's main routine. (If you loaded Chapter 3's program, the two new subroutine tools would be lost.)

Type **RUN 10 RETURN**. Wait while your practice main routine lines are erased. Then type in Chapter 3's main routine lines as follows:

```
1100 GOSUB 20           :REM GRAPHICS
1110 C = 14: GOSUB 40:REM COLORS
1120 GOSUB 50           :REM PAINT BKGROUND
1200 REM::::::::::LIGHTHOUSE
1210 POKE 18090,0: POKE 18130,17
1220 POKE 18170,17: POKE 18210,17
1300 REM::::::::::WATER
1310 FOR Y = 176 TO 199
1320 FOR X = 0 TO 319
1330 IF RND(1) < .3 THEN GOSUB 70
1340 NEXT X: NEXT Y
5000 GET A$
5010 IF A$ = " " THEN 6000
5020 GOTO 5000
6000 GOSUB 30
6010 END
```

Run the program to make sure it was typed correctly. Plotting the water will take some time, so now would be a good time to take a break. Check your screen again in 15-20 minutes. When the sky, water and lighthouse are complete, press the **SPACE BAR** to return to the program listing.

Below are the outlines for the land and waves drawn on graph paper. The X,Y coordinates for each pixel point that begins or ends a line is shown. These coordinates will be used in the new program to plot the outlines.



Type the lines shown below that will outline the left and right areas of land. Compare these program lines to the X,Y coordinates shown above.

```

1400 REM:::::LEFT LAND
1410 C = 94: REM COLOR = GREEN ON BLUE
1420 X1 = 0: Y1 = 160
1430 X2 = 16: Y2 = 168: GOSUB 80
1440 X1 = 24: Y1 = 168: GOSUB 80
1450 X2 = 40: Y2 = 175: GOSUB 80
1460 X1 = 0: Y1 = 175: GOSUB 80
1470 X2 = 0: Y2 = 160: GOSUB 80
1500 REM:::::RIGHT LAND
1510 X1 = 240: Y1 = 175
1520 X2 = 261: Y2 = 164: GOSUB 80
1530 X1 = 279: Y1 = 164: GOSUB 80
1540 X2 = 309: Y2 = 152: GOSUB 80
1550 X1 = 319: Y1 = 159: GOSUB 80
1560 X2 = 319: Y2 = 175: GOSUB 80
1570 X1 = 240: Y1 = 175: GOSUB 80

```

Check each line and correct any errors you find. When you are ready to see the outlines plotted, move the cursor to a *free, blank line* somewhere below your program. Carefully type the following:

```
GOSUB 20: RUN 1400 RETURN
```

This is a new method of by-passing portions of your program. GOSUB 20 turns on your high resolution screen. RUN 1400 tells the computer to start running the program at line 1400. When you type statements without line numbers, they are performed but *not* inserted in the program.

Compare what is happening on your screen to the previous diagram. The land should be outlined in green foreground pixels, in a form similar to the diagram.

When the outline is complete, press the **SPACE BAR**. If you had any problems, check each line carefully. Look at all X and Y coordinates. Common mistakes include reversing the X's and Y's, or typing X1 instead of X2. Correct all typing mistakes. To try again, you will have to re-run the entire program. Line 1120 changes all foreground pixels to background pixels, which would now be necessary if you plotted the outlines improperly.

Let's move on to the outlines for the left and right waves:

```
1600 REM:::::LEFT WAVE
1610 C = 14: REM COLOR = BLACK ON BLUE
1620 X1 = 48: Y1 = 175
1630 X2 = 87: Y2 = 169: GOSUB 80
1640 X1 = 95: Y1 = 175: GOSUB 80
1650 X2 = 48: Y2 = 175: GOSUB 80
1700 REM:::::RIGHT WAVE
1710 X1 = 144: Y1 = 175
1720 X2 = 183: Y2 = 169: GOSUB 80
1730 X1 = 191: Y1 = 175: GOSUB 80
1740 X2 = 144: Y2 = 175: GOSUB 80
```

To add these outlines to your picture, move the cursor to a free line and type:

```
GOSUB 20: RUN 1600 RETURN
```

Flip back to the diagram showing the outline shapes. Make sure the program is plotting your outlines correctly. When both waves are complete, press the **SPACE BAR**.

Look at lines 1700 through 1740. Recall that a shape's outline is made of several connecting lines. To plot each line, the X1, Y1 coordinate and X2, Y2 coordinate are necessary. GOSUB 80 plots a line between the two points. To plot lines, the computer will use the *last* X1, Y1 and X2, Y2 coordinates given in the program. This allows you to re-use coordinates without re-typing them.

TOOL 80:::PLOT A LINE

```

80 REM:::PLOT A LINE
81 DX = X2 - X1: DY = Y2 - Y1
82 L = ABS(DX): IF ABS(DY) > L THEN L = ABS(DY)
83 IF L > 0 THEN XI = DX/L: YI = DY/L
84 X = X1 + .5: Y = Y1 + .5
85 FOR I = 0 TO L
86 GOSUB 70 : REM PLOT POINT
87 X = X + XI: Y = Y + YI
88 NEXT I
89 RETURN
    
```

What It Does: This routine plots a “straight” line, in the foreground color specified by C’s current value, from coordinates X1,Y1 to X2,Y2.

Example Use: To plot a line, all you need is one or more main routine lines to give the starting point (X1,Y1) and ending point (X2,Y2) of the line to be plotted. To have the line plotted, a GOSUB 80 is necessary. For example:

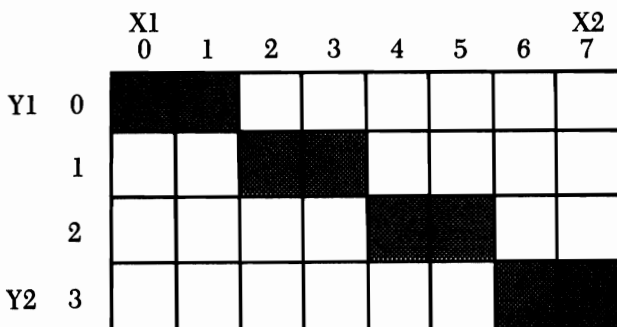
```

1110 X1 = 1st X coordinate: Y1 = 1st Y coordinate
1120 X2 = last X coordinate: Y2 = last Y coordinate
1130 GOSUB 80
    
```

When plotting several lines to make an outline shape, C’s value should be set to the color code that will later be used to paint the shape. This should be done somewhere before the GOSUB 80 statement. For example:

```
1100 C = 14
```

Technical Description: To understand how a line is drawn, imagine a dot that travels from point X1,Y1 to point X2,Y2, changing each pixel in its path to the foreground color. To get to point X2,Y2, the dot must first figure out the most direct pattern of motion, and the number of times it must repeat that pattern to reach its destination. We will use these terms “patterns” and “repetitions” to explain this subroutine.



The “pattern” in the above example is: <plot>; 1 step right; ½ step down. There are 8 “repetitions” of this pattern. In other words, the pattern is performed 8 times.

Two more terms that will help explain this subroutine are “major axis” and “minor axis.” In the above example, it is farther from X1 to X2 than it is from Y1 to Y2, so X is the “major axis” and Y is the “minor axis.” If the distance from Y1 to Y2 were greater than the distance from X1 to X2, then Y would be the major axis.

The dot always follows a standard pattern of:

- (a) <PLOT>
- (b) 1 step along the major axis
- (c) 1 step *or less* along the minor axis

To plot a line, the dot must:

- (a) Decide which axis is the major axis and which is the minor;
- (b) Decide how many pattern *repetitions* it must perform;
- (c) Decide *how far* to move along the minor axis for each whole step along the major axis in each repetition. (This is the only variable, other than the major and minor axis, which affects the pattern.)

$$81 \text{ DX} = \text{X2} - \text{X1}; \text{ DY} = \text{Y2} - \text{Y1}$$

This line calculates the distance from X1 to X2 (DX), and the distance from Y1 to Y2 (DY). The larger of these two numbers becomes the major axis, and the smaller becomes the minor axis.

$$82 \text{ L} = \text{ABS}(\text{DX}); \text{ IF } \text{ABS}(\text{DY}) > \text{L} \text{ THEN } \text{L} = \text{ABS}(\text{DY})$$

This line calculates the number of pattern repetitions necessary to complete the line. Since the dot will step once along the major axis for each repetition, the length of the major axis can be used as the number of repetitions. The number of repetitions is stored in the variable L. Line 82 begins by assuming X is the major axis, and therefore $\text{ABS}(\text{DX}) =$ “number of repetitions.” It then checks to make sure it was correct in assuming that X is the major axis. If it finds that Y is actually the major axis, it changes the number of repetitions to equal $\text{ABS}(\text{DY})$.

$$83 \text{ IF } \text{L} > 0 \text{ THEN } \text{XI} = \text{DX}/\text{L}; \text{ YI} = \text{DY}/\text{L}$$

This line calculates the pattern the dot will be repeating to reach its destination. First, this line checks to make sure the number of repetitions is greater than (<) zero. If it’s not, the dot won’t go anywhere. It will just plot the point at X1,Y1 and stop. Otherwise, it figures out how far it must move along the X axis (XI) and along the Y axis (YI) for each repetition. Since L = either $\text{ABS}(\text{DX})$ or $\text{ABS}(\text{DY})$, then one of these

numbers (XI or YI) will be 1 (-1 if it is moving towards the axis). The other number will be less than or equal to (\geq) 1. By adding these numbers (increments) to the current dot position after each plot, the dot will move in the desired pattern.

```
84 X = XI + .5: Y = YI + .5
```

Line 84 places the dot at the starting point of the line to be plotted. Adding .5 to each coordinate provides a more accurate starting position. Since the point plotting routine only uses the integer portion of the X,Y coordinates, an X coordinate of 4.99 would become 4. By adding .5 to the X coordinate, it becomes 5.49, or 5 by the point plotter, which is closer to the intended value of 4.99. This process is called "rounding."

```
85 FOR I = 0 TO L
86 GOSUB 70: REM PLOT A POINT
87 X = X + XI: Y = Y + YI
88 NEXT I
```

This loop does all of the work. Line 85 loops once for each pattern repetition. Line 86 plots the point the dot is currently on. Line 87 adds the increments which move the dot to the next position. Line 88 returns to the start of the loop for another repetition.

```
89 RETURN
```

This line has the computer return to the main routine after the line has been plotted.

Painting the Land and Waves

Once you have the outlines plotted, you can paint the area within the outline. To paint the waves, enter the following program lines:

```
1800 REM:::::PAINT LEFT WAVE
1810 C = 14: REM COLOR = BLACK ON BLUE
1820 X0 = 48: Y0 = 169
1830 W = 47: H = 6
1840 PC = .3: GOSUB 90
1900 REM:::::PAINT RIGHT WAVE
1910 X0 = 144: Y0 = 169
1920 W = 47: H = 6
1930 PC = .3: GOSUB 90
```

Painting is done within the height and width of the shape, starting at the X0,Y0 point listed in the program. The height and width of a shape is determined on a 0-based scale. Since the left wave is 48 columns across and 7 rows high, W=47 and H=6. PC is the percent of pixels to be plotted in the shape. The water was randomly plotted 30% (.3) of the time, and this must also be done to the waves (PC=.3). Move the cursor to a free line and type:

```
GOSUB 20: RUN 1800 RETURN
```

Watch as each wave is randomly plotted within the outline shape. Plotting is done column-by-column, between the straight sets of outline pixels. When the bottom of the shape has been reached, the next column is plotted. Plotting stops when the last column used by the shape has been completed.

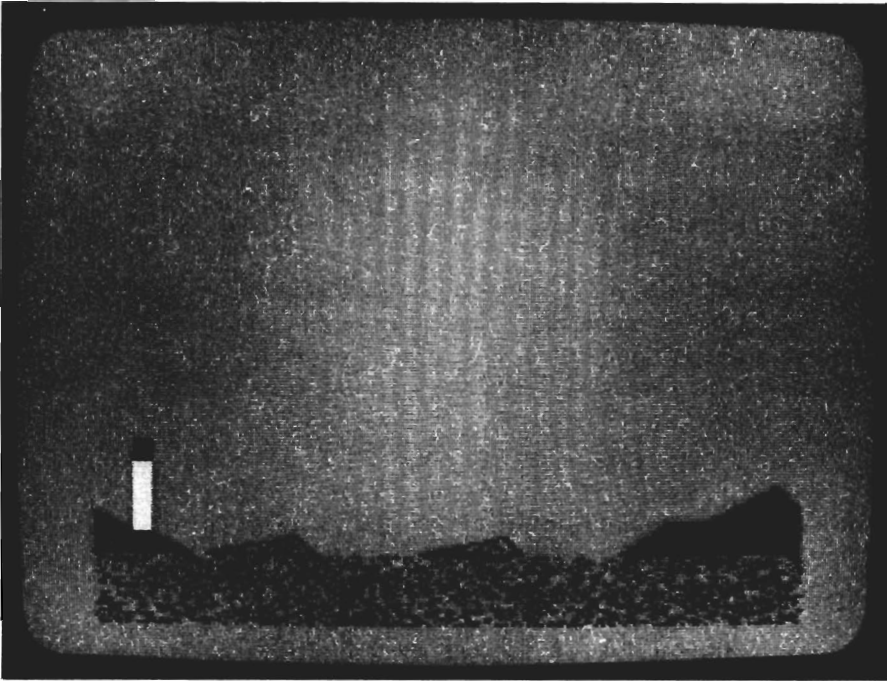
After the waves are painted, press the **SPACE BAR**. All that remains is to paint the land. Type the following program lines:

```
2000 REM:::::PAINT LEFT LAND
2010 C = 94: REM COLOR = GREEN ON BLUE
2020 X0 = 0: Y0 = 160
2030 W = 40: H = 15
2040 PC = 1: GOSUB 90
2100 REM:::::PAINT RIGHT LAND
2110 X0 = 237: Y0 = 152
2120 W = 82: H = 23
2130 PC = 1: GOSUB 90
```

PC is now equal to 1. This completely paints each land area with a solid color. Type GOSUB 20: RUN 2000 **RETURN**.

Watch what is happening on your screen carefully. As each block is plotted in, foreground pixels which are already present in the block are changed to green. This is how the left-most vertical outline of the left land gets changed to green. The subroutine itself actually ignores this column because it only contains one straight set of outline pixels. After a short time lapse, the right land area will be painted.

Compare your screen to the picture shown below. If you are not sure why the new main routine lines produced this portion of your picture, review the sections on "Plotting Lines" and/or "Painting Shapes."



TOOL 90:.....PAINT A SHAPE

```

90 REM:.....:PAINT A SHAPE
91 PC=PC+ABS(PC=0): FOR X = X0 TO X0 + W: FL$ = "F": PR = 0
92 FOR YC = Y0 TO Y0 + H: Y = YC: GOSUB 60
93 ON ABS((PEEK(BYTE) AND 2 ↑ BIT) <> 0) GOTO 97: IF PR=0
  THEN 96
94 PR = 0: IF FL$ = "F" THEN Y1 = YC: FL$ = "T": GOTO 96
95 GOSUB 99: FL$ = "F"
96 NEXT YC: GOTO 98
97 PR = 1: NEXT YC: IF FL$ = "T" THEN GOSUB 99
98 NEXT X: RETURN
99 FOR Y = Y1 TO YC -1: ON ABS(RND(1) < PC) GOSUB 70:
  NEXT Y: RETURN

```

What It Does: This subroutine will fill in most outline shapes with the foreground color specified by C's current value.

Example Use: To use this tool, you need main routine lines that give the first column used by the shape (X0=?); the first row used by the shape (Y0=?); the height of the shape, using zero as the base number (H=?); and the width of the shape, using zero as the base number (W=?). In addition,

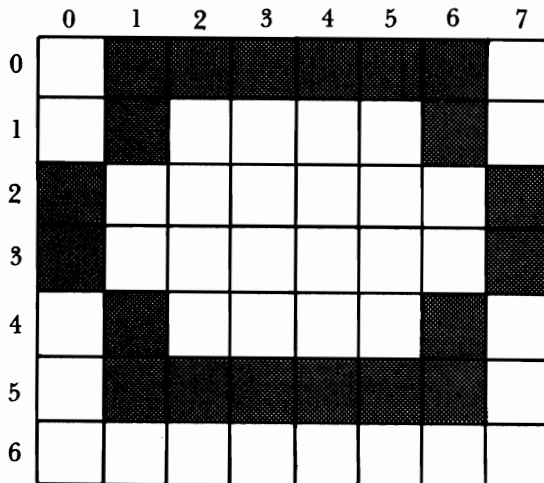
you need to specify the fractional percent of pixels within the shape that should be plotted (PC=?). This is done in the form of:

```
1100 X0 = left-most X coordinate: Y0 = upper-most Y coordinate
1110 W = highest X - lowest X: H = highest Y - lowest Y
1120 PC = percent of pixels to be plotted expressed in a fraction: C =
foreground/background color code for the shape
1130 GOSUB 90
```

Technical Description: This subroutine searches each column in a specified range, looking for columns containing 2 outline pixels. The upper outline of the column is the "initial" point, and the lower outline is the "terminal" point. When these two points have been found, line 99 plots the pixels between them.

In line 91, the statement $PC=PC+ABS(PC=0)$ checks to see if PC is set to zero. If it is, then PC is automatically set to 1. Otherwise, it is left alone. This statement was included in case the variable PC was not set before the subroutine was called. If you never set PC (percentage of pixels to paint), then PC would equal zero, and 0% of the pixels in the shape would be painted. This would cause the subroutine to *appear* not to work. Adding this statement to the subroutine makes it easier to locate the problem when PC is not set properly.

The variable FL\$ keeps track of whether the initial point has been found. FL\$ = "T" means it has, FL\$ = "F" means it hasn't. The variable PR keeps track of the previous pixel. If the previous pixel was background colored, then PR = 0. If the previous pixel was foreground colored, then PR = 1. The following example will show why this is important.



In column 1, we want to fill in the pixels between row 1 and row 4. Even though the upper part of the outline in this column is 2 pixels high, we still want to consider it as the initial point. We can do this by keeping track of the previous pixel. If the current pixel is 0 (background colored), but the previous pixel was 1 (foreground colored), then we know we just *passed* an outline. We want to wait until we pass an outline before we process the outline. When the outline has been passed, there are two possibilities. The first is that the outline is an initial point. The second is that the outline is a terminal point. If the outline is an initial point, then Y1 is set to the current value of Y. This will keep track of the initial point until it can be used. The FL\$ should be set to "T" since the initial point has been found. If the outline is a terminal point, then a GOSUB 99 will plot the pixels between Y1 and the current Y coordinate. FL\$ is set back to "F" to look for another initial point.

There is one special case we must also check for. If we have found a terminal point, but reach the bottom of the shape before we pass it, then we must still process it as if we had passed it.

Line 99 does the actual work of painting. GOSUB 70 directs the computer to the PLOT A POINT subroutine. ON ABS(RND(1)<PC) GOSUB 70 will plot a point the specified percent of the time.

More on Colors

Up to this point, we have talked about designing your picture with foreground/-background color blocks in mind. You have learned that certain color combinations work better together than others. Complementary colors produce a sharp contrast, while other combinations may produce blurred images. Using certain color combinations, you can even put emotional effects into a picture, such as using warm colors (red, orange, yellow) for hot feelings, and cool colors (blue, green, purple) for calm feelings. In this section, we will talk more about how colors can be used for visual effects. The topic will be shadows and highlights.

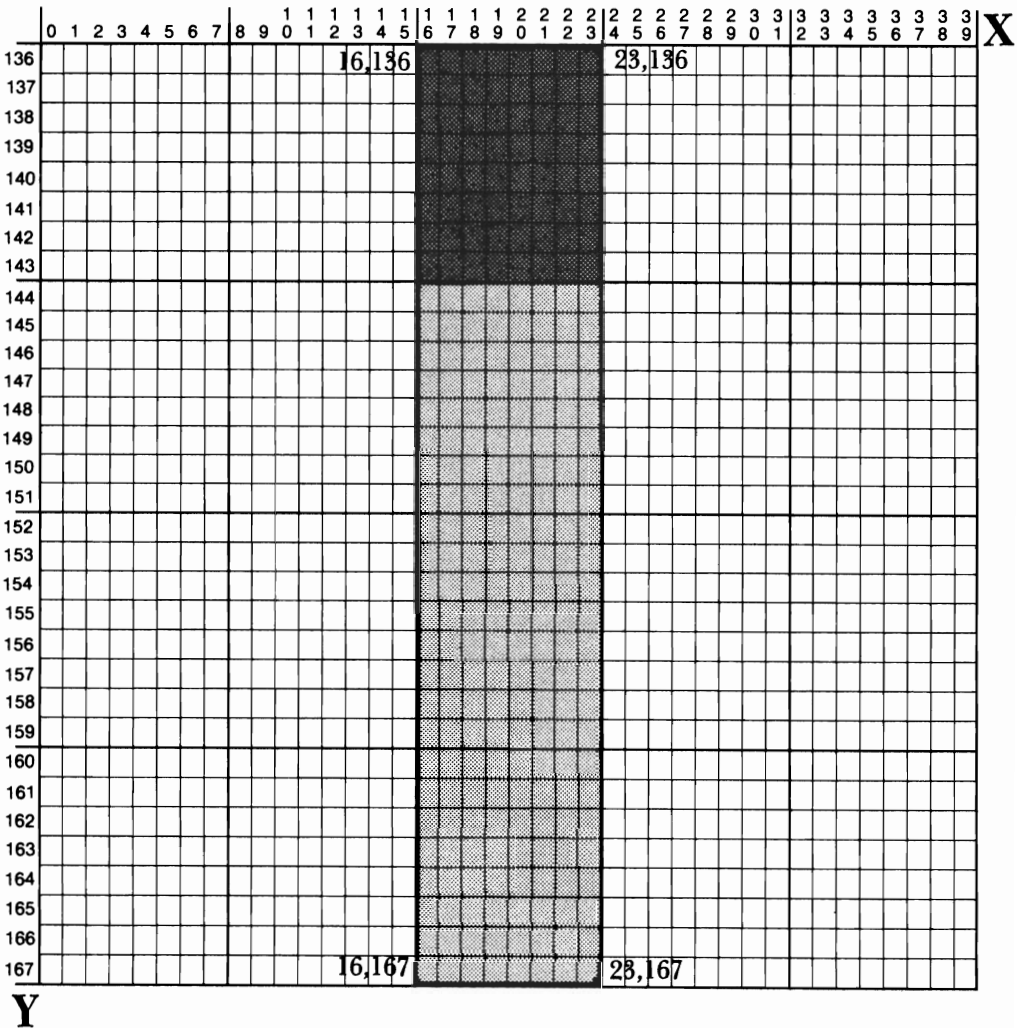
In painting, shadows are usually made by darkening a color. The original color is called the *base* color. This base color is darkened by adding a darker color to it—like purple or black. On the Commodore 64, you don't have to bother with mixing colors because they are already there. A shadow effect can be created by using two similar colors, where one color is slightly darker than the base. The darker color should, however, be related to the base color—for example, they should both be in the same family of colors (warm or cool).

You can also put highlights into your pictures. A highlight color is almost the same as the base color, but it is slightly lighter. Think of shining a flashlight on a red ball. The areas where the light is reflected off the ball would be considered the highlights, and would be light red. The highlight of a blue ball would be light blue. The highlight of green is light green. Another name for highlights is *tints*, which means to add white to the base color.

The use of shadows and highlights gives a picture a sense of depth. This is easily done by painting one side of an object with a highlight or shadow color. Let's try the shadow idea on the lighthouse. We'll assume that the left side of the lighthouse is away from the sun, and so it should be slightly darker than the right side. To shade this left side, we will plot three dark lines from the top of the lighthouse to the bottom of the lighthouse.

Shading the Lighthouse

The lighthouse is located on your screen as shown in the following diagram:



You want to draw three lines down the left side of this lighthouse. The coordinates for these lines are:

- 16,136 to 16,167
- 17,136 to 17,167
- 18,136 to 18,167

The program lines which will shade the lighthouse are shown below. Line 3010 changes the color code to a light grey plotted on white. Lines 3020-3040 plot the three lines. Notice that it does not matter in what order you list X1, Y1, X2, or Y2. As long as they are all given before GOSUB 80, a line can be plotted. Since each line will be plotted from the same beginning row (Y1=136) down to the same ending row (Y2=167), Y1 and Y2 only need to be given once. Finally, line 3050 pokes the colors grey on black in the top block of the lighthouse. Enter these new program lines now.

```

3000 REM:::::SHADE LIGHTHOUSE
3010 C = 241
3020 X1 = 16: X2 = 16: Y1 = 136: Y2 = 167: GOSUB 80
3030 X1 = 17: X2 = 17: GOSUB 80
3040 X1 = 18: X2 = 18: GOSUB 80
3050 POKE 18090, 11
    
```

Type GOSUB 20: RUN 3000 **RETURN**. Watch as the lighthouse is shaded, as if the sun is off the screen to the right. Now the picture really appears to be life-like. When the shading is finished, press the **SPACE BAR**. You probably need a better look at how this worked:

PROGRAM LINE #	START OF LINE X1,Y1	END OF LINE X2,Y2
3020	16,136	16,167
3040	17,136	17,167
3040	18,136	18,167

In the chart above, you can easily see that neither Y1 or Y2 ever change. Thus, they only need to be included once in the program. X1 and X2 change for each line, and so these new column locations have to be included before each GOSUB 80 statement.

Shaded and highlighted objects in a picture create a more dramatic effect because they give objects a sense of depth, thickness, and volume. It's a good idea to try out the shadows and highlights on your graph sheet before entering the program. By coloring in the shades and highlights on your graph sheet first, you can make quick changes and learn to control the colors for a more successful design.

Finishing Up The Program

With two more chapters to go, “Finishing Up The Program” might sound a bit premature. You would probably agree, however, that the current program is getting to be a bit cumbersome. To keep the program listing down to a manageable size, this book’s final picture will be drawn with three separate programs. You are about to finish up the first of those three.

Let’s discuss how this “3-program picture” idea will work. Your picture is currently being stored on the high resolution graphics screen. It will remain there until you: (a) turn the computer off, or (b) run another program that erases or changes what’s on this screen. As long as Chapter 5’s program does *not* erase or change anything you’ve already drawn, anything it draws will be *added* to your current picture. Thus, we can put Chapter 4’s picture on the high resolution screen, use Chapter 5’s program to add more detail to the picture, and then finish it up with Chapter 6’s program.

At the beginning of Chapter 5, you will first load this chapter’s picture onto the high resolution screen. Next, you will enter Chapter 5’s new main routine lines. Finally, you will run Chapter 5’s program to add a new object to your picture.

When combining pictures drawn by two or more programs, the program run *last* will dominate. By this, we mean it will control any overlapping areas. For example, suppose you run a program that draws a white house on a *light blue* background. Next, you run a program that draws a red sun on a *yellow* background. The combined picture would display a white house, a red sun, and a *yellow* background (the background determined by the program run last).

To finish up this program and move on, there are just a couple of loose ends to tie up. The first is the picture’s border. Have you noticed that the left and right edges of the water seem to begin and end in the middle of nowhere? This was done because of the border surrounding the high resolution screen—it can’t be drawn on. To improve the appearance of the picture, you will plot three lines to confine it. As a simple exercise, add program lines that will do the following:

- (1) Plot a line down the left edge of the screen, stopping just above the land (Y=159);
- (2) Plot a line across the entire top row of the screen;
- (3) Plot a line down the right edge of the screen, stopping just above the land (Y=166).

Draw these lines using green foreground pixels plotted on blue background pixels. Begin with line 4100 REM:::DRAW BORDER.

The solution is shown below. You can’t put border lines around the water because that would require 3 colors (green, black and blue) in some of the color blocks that the water uses. Knowing that only 2 colors can be used per block, the border lines are kept above the water.

```

4100 REM:::::DRAW BORDER
4110 C = 94: REM COLOR = GREEN ON BLUE
4120 X1 = 0: Y1 = 159
4130 X2 = 0: Y2 = 0: GOSUB 80
4140 X1 = 319: Y1 = 0: GOSUB 80
4150 X2 = 319: Y2 =166: GOSUB 80

```

Saving A Picture On Disk or Tape

In Chapters 5 and 6, you will be entering two new and independent programs. Combining the pictures from this chapter and Chapters 5 and 6 will complete the artwork that is covered in this book. Currently, the only way to combine the pictures of two or more programs is to load and run each program. Having come this far, you are painfully aware of the time this will take.

A SAVE PICTURE tool is in order, and thus supplied. This tool will save the *picture* created by a program. That picture can then be easily and quickly loaded into memory at any time. Before saving this chapter's program, *carefully* type the following new tool lines:

```

100 REM:::::SAVE PICTURE
101 INPUT "ENTER FILENAME"; FILE$
102 INPUT "ENTER 8 FOR DISK, OR 1 FOR CASSETTE"; DE
103 SYS 57812 FILE$ + ".PIC", DE
104 POKE 174,64: POKE 175,127: POKE 193,0: POKE 194,96
105 SYS 62954
106 SYS 57812 FILE$ + ".COL", DE
107 POKE 174,232: POKE 175,71: POKE 193,0: POKE 194,68
108 SYS 62954: END

```

This tool is a "routine," and is executed in almost the exact same manner as the ZAP routine. Because it is, take the time to save Chapter 4's program *now*. Never run this routine or the ZAP routine until the current form of your program is safely stored on disk or tape.

Look over this new tool one more time. When you test this tool, the only way to correct it and test it again is to re-run Chapter 4's program. That would take about 20 minutes of your time. Better to take an extra 5 minutes of proofreading time now.

To put this tool into action, make sure a disk is present in your disk drive. If you are using a cassette recorder, set the counter and tape in the same manner as when saving a program. Write down the filename "CHAPTER 4.PIC" on your list of program names. Write down the starting counter number. Storing a picture on tape is no different than storing a program on tape. The picture will take up space, and should not be recorded over.

To save the picture that currently resides in memory, type RUN 100 **RETURN**.

The computer should respond with the prompt "ENTER FILENAME". You are to enter the filename you want to attach to the picture. The name can be a maximum of 12 characters in length. An important thing to understand is that the computer will save your picture in two different files. The first file will contain the pixel patterns that make up the picture. The second file will contain the colors that make up the picture. Each will have a filename that begins with whatever you type now. However, one will have ".PIC" appended to the name, and the other will have ".COL" appended to the name. Enter CHAPTER 4 as the filename for this picture.

The computer now responds with "ENTER 8 FOR DISK, OR 1 FOR CASSETTE". If you are using a disk drive, enter 8. Otherwise, enter 1. From this point on, the computer will prompt you as if you were saving a program. The prompts will be familiar, so respond to each in the usual manner.

There is no way to directly verify that the picture was stored on your disk/tape. The only way to find out is to clear the high resolution screen and then load the picture. Move the cursor to a free line and type:

```
C=14: GOSUB 40: GOSUB 50 RETURN
```

This will clear the screen of any image, and set all color blocks to black on blue. When the cursor reappears, you can try loading the picture. To do so, you must load the following *two* files (one right after the other):

For Disk Drive Users

```
LOAD "CHAPTER 4.PIC",8,1
  then
LOAD "CHAPTER 4.COL",8,1
```

For Cassette Recorder Users

```
LOAD "CHAPTER 4.PIC",1,1
  then
LOAD "CHAPTER 4.COL",1,1
```

(If you are using a cassette player, be sure to rewind the tape to a point before the picture begins.)

The above files are loaded almost the same as program files, except that a ",1" needs to be appended to the end. This 1 tells the computer to replace the picture in Bank 1 instead of Bank 0, and should be typed regardless of whether you are using a disk drive or a cassette player.

After you enter the load command for one of these files, the computer will respond as if you were loading a program file. Answer all prompts as you would when loading any other file. (Occasionally, the computer may "freeze up" after you've loaded the first picture file. When this happens, press **RUN/STOP** and tap **RESTORE**. You should be returned to text mode, and can then load the second picture file.) When both files have been loaded, type GOSUB 20 and press **RETURN** to see the picture.

If Chapter 4's picture does not get displayed, press **RUN/STOP** and tap **RESTORE**. LOAD Chapter 4's program. Check each line in the SAVE PICTURE routine. Check 0's and 1's. Make sure you are not using oh's and small l's by mistake. Look at the line numbers. Have any lines been omitted? Make sure you use semi-colons (;) and colons (:) properly. When the tool is entered correctly, re-save this chapter's program (see Chapter 1 for instructions on re-saving a program). RUN the program. When the picture is restored on the high resolution screen, RUN the SAVE PICTURE routine again.

NOTE: Loading picture files has no effect on any program listing that may be in memory. It does, however, have effect on the high resolution screen. The newly loaded picture will completely *erase* anything already on the high resolution screen.

TOOL 100:.....:REM SAVE PICTURE

```
100 REM:.....:SAVE PICTURE
101 INPUT "ENTER FILENAME"; FILE$
102 INPUT "ENTER 8 FOR DISK, OR 1 FOR CASSETTE"; DE
103 SYS 57812 FILE$ + ".PIC", DE
104 POKE 174,64: POKE 175,127: POKE 193,0:
    POKE 194,96
105 SYS 62954
106 SYS 57812 FILE$ + ".COL", DE
107 POKE 174,232: POKE 175,71: POKE 193,0:
    POKE 194,68
108 SYS 62954: END
```

What It Does: This routine will save your picture to tape or disk so that it can be quickly restored to the high resolution screen. The picture is saved in two files. One file contains the picture pattern, and one file contains the color pattern.

Example Use: Type RUN 100 and press **RETURN** to save the picture that currently resides in memory. You will be prompted for a filename, which can be any name you desire (up to 12 characters in length). After entering a filename, you will be prompted for a storage device number. Type 8 if you are using a disk drive, or type 1 if you are using a cassette recorder. The picture will then be saved in the same manner as when saving a program. To load the picture back into memory, type:

```
LOAD "filename.PIC",8,1 (disk)
LOAD "filename.COL",8,1 or
LOAD "filename.PIC",1,1 (tape)
LOAD "filename.COL",1,1
```

Note that *filename* should be replaced with the filename assigned by you when the picture was originally saved.

Technical Description: This routine uses three pieces of the BASIC

“SAVE” command to save the picture. These pieces are used to:

- (1) Get the name in quotes and the storage device number;
- (2) Determine the area of memory that holds the information that is to be saved;
- (3) Save the information in the specified area.

Lines 101 through 103 collect the filename and storage device number. SYS 57812 uses a portion of the SAVE command to store the filename and device number entered. Line 104 determines the area of memory that holds the pixel patterns, and line 105 saves the information in that area. Lines 106 through 108 repeat this process to save the color information on the disk/tape.

Summary

You have reached the first major milestone in this book. With the first program completed, the last two chapters will take no time at all. Even if you quit reading this book right now, your current knowledge of computer graphics is extensive enough to:

- draw lines in any direction
- connect lines to form outline shapes
- paint any shape of any size
- use shadows and highlights to give pictures a more “true-to-life” appearance
- store and retrieve pictures to and from a disk/tape

Drawing lines and shapes is easy and fun. A line is simply a series of plotted pixels, one right after the other. Using the PLOT A LINE tool, all you need to know are the starting and ending points of each line.

Lines can be drawn in any direction, and can start anywhere *on the screen*. Incorrectly plotting a point off the screen could cause the computer to “freeze up.” Remember that X coordinates establish the *column* to start in, and can range from 0 to 319. Y coordinates establish the *row* to start in, and can range from 0 to 199.

By connecting several lines together, you can make outlines for shapes. Making outlines is the first step in painting shapes with colors. Outlines are painted from top to bottom (H=?), across the width of the shape (W=?). As each column is passed through, the straight sets of outline pixels turn plotting on and off. This idea must be kept in mind when designing any shape which will be painted later.

Colors can create different visual effects—such as shadows (darker than the base color) and highlights (lighter than the base color). On a graph sheet, you can quickly estimate which color combinations work well together, and which color blocks will be effected. Once you decide on the colors, you can change “C” to the appropriate foreground/background color code.

Chapter 5 teaches you how to use a new tool called DRAW A SHAPE. This tool allows you to draw a shape *once*, and then duplicate it anywhere and as many times on the screen as you like. If you plan to create detailed computer graphic designs and artwork, you will find this tool to be an invaluable addition to your tool kit.

Chapter 5

TEST PLOTTING AND DUPLICATING SHAPES

Now that you have plotted points and lines, and have painted shapes, you are ready to add the ship to your picture. To do this, you need to plot 68 lines. In addition, you will place several seagulls in the picture. You'll do all of this with a new tool called DRAW A SHAPE.

The DRAW A SHAPE tool has two very useful functions. First, it provides an easy way to "test plot" a shape on the screen. You can plot a shape, see where it falls in your picture, and then easily adjust the program if the shape needs to be moved in any direction. With last chapter's PLOT A LINE tool, the only way to move a misplotted shape was to change each X and Y coordinate for each line. This new tool only needs *one* X and *one* Y adjustment to re-plot an entire shape in a new location. The ship was test plotted several times before its final placement was established. You will find that being able to "test plot" shapes can be a great time-saver for shapes of any complexity.

The second important feature of this tool is its ability to duplicate a shape. Once a shape has been drawn with this tool, it can be duplicated as many times as you like anywhere on the screen. To duplicate the shape, all you need to do is specify *where* you want the duplicate copy(s) to appear. Think of the programming time that can be saved as:

- one man becomes a crowd of people
- one car becomes a traffic jam
- one building becomes a city scene
- one brick becomes an entire wall
- one flower becomes a brilliant bouquet

All of this is done by storing a written description of your shape in memory. Each time the shape needs to be moved or copied on the screen, the computer simply looks at the stored description.

Defining a Shape With Data Lists

The description of the shape is made up of lists stored in memory. One list describes the shape's "endpoints," and the other list describes the shape's lines. The first thing to do is gather the information which describes the shape to be drawn. By sketching the shape on graph paper, these two lists can easily be compiled.

On the first list, write down all X,Y coordinates which form endpoints in the shape. Any point which begins or ends a line in the shape is called an "endpoint." Thus, a square shape has four endpoints, and a triangle has three endpoints. The

You have only recorded the shape's endpoints, so another list is needed to finish the description. This second list will give a line-by-line description of the shape, and is set up in the following form:

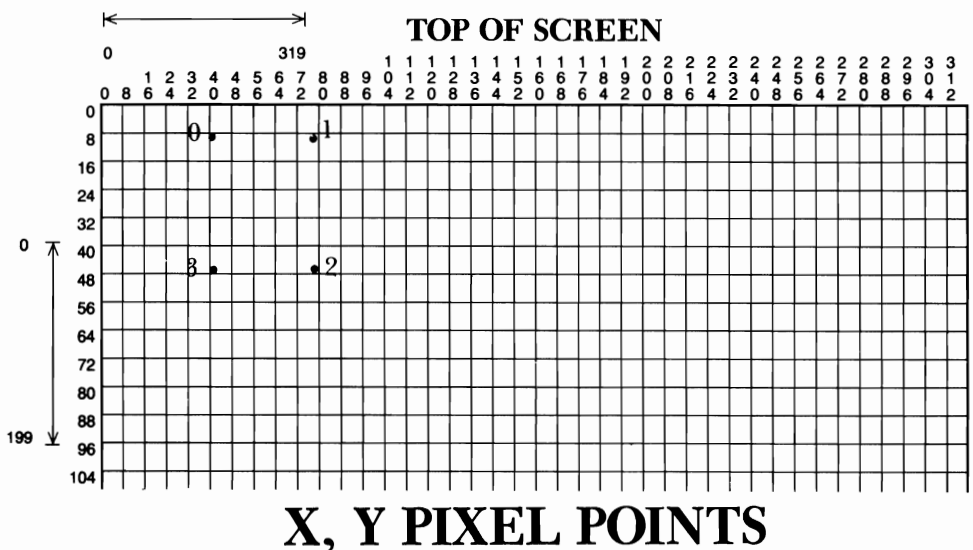
	line 0	line 1	line 2	line 3	... Line N
"FROM" Endpt. #					
"TO" Endpt. #					

For each line in the shape you will need a column to write down its "from" and "to" endpoints. All lines are plotted "from" one endpoint "to" another. In a shape that has 27 lines, for example, 27 columns will be needed (numbered 0 to 26). *Using the numbers you assigned to each endpoint*, one column is completed for each line in the shape. For the square, this would be:

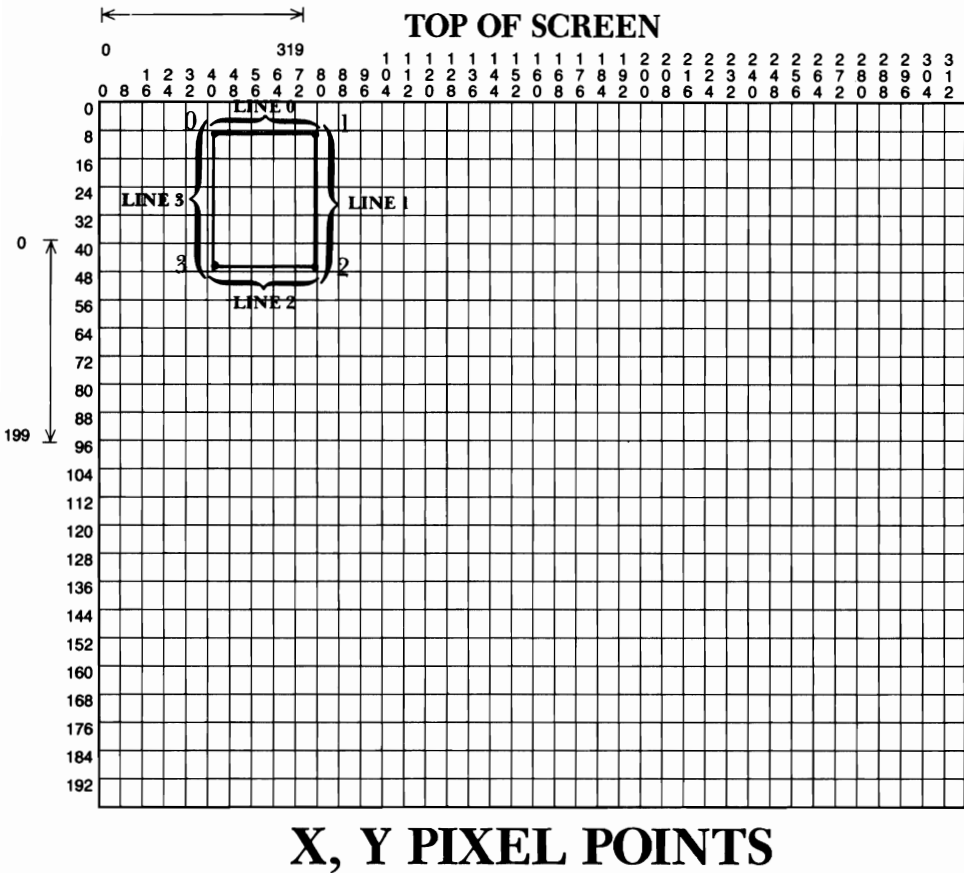
	line 0	line 1	line 2	line 3
"FROM" Endpt. #	9	1	2	3
"TO" Endpt. #	1	2	3	0

Again, it does not matter which line is assigned to column #0, or which to column #1. As long as the columns are numbered starting with 0, and no numbers are skipped, you will have no problems. What *is* important is that you use the *correct* endpoint numbers, as assigned on your first list.

That completes the second list needed by the DRAW A SHAPE tool. This second list is your "line data." Let's see how the computer can use these lists to draw or re-draw any shape. In the example of our square, the endpoint data list gives all the endpoints of the shape:



This empty “connect-the-dots” board is easily filled in by looking at the line data list:



What would happen if we changed the line data list to contain the following:

	line 0	line 1	line 2	line 3
“FROM” Endpt. #	0	1	3	2
“TO” Endpt. #	1	3	2	0

Even though the endpoint data list remained the same, the line data list describes the following shape:

NOTE: During this discussion, do *not* type any program lines. Typing begins in the next section, “Test Plotting.”

A DIM statement does several things. First, it assigns a name to the list. We have chosen the name “E” for the *Endpoints* data list. The “%” tells the computer that the list will be used to store whole number (integers). It is important to remember that the DRAW A SHAPE tool expects you to use E% when storing your endpoints data list.

The numbers in parentheses (1,3) are called *subscripts*, and they establish the number of rows and columns the list will use. The first number gives the number of *rows* in the list. This is a zero-based number, so a 1 means the list uses 2 rows. The second number gives the number of *columns* in the list. This is also a zero-based number, so a 3 means the list uses 4 columns.

Given the size of your list, the computer sets aside enough memory space (*no more and no less*) to hold the data. If you say your list only uses 10 columns and then you try to use 11, you will get a “BAD SUBSCRIPT ERROR.”

It’s okay, however, to say the list is longer than it really is. In fact, it’s usually a good idea to have the computer set aside more memory space than your list actually uses. The E% list will never use more than two rows (one for X coordinates and one for Y coordinates), so the row dimension can be left at 1. However, the number of columns needed will depend on the number of endpoints in your shape. We suggest that you create E% with 100 columns, which will be sufficient for almost all shapes. To give E% this size, the program should read:

```
1100 DIM E%(1,99)
```

Once this line is entered into the program, you have an empty E% list that needs to be filled with endpoint data. Let’s compare the empty list in memory to the handwritten endpoint list from earlier in the chapter:

Empty E% List In Memory

	Column 0	Column 1	Column 2	Column 3	... Column 100
Row 0					
Row 1					

Written Endpoint Data List

	endpt. 0	endpt. 1	endpt. 2	endpt.3
X	40	79	40	79
Y	8	8	47	47

You need a way to copy your list directly into memory. Each “position” in memory can be filled with a data item by using the correct Row # and Column #. For example, the first position in E% is Row 0, Column 0. This is written as E%(0,0).

The next position *down* is Row 1, Column 0. This is written as E%(1,0). (Note that the Row notation always comes before the Column notation.) For the square data, we would like:

- 40 put in position (0,0)
- 8 put in position (1,0)
- 79 put in position (0,1)
- 8 put in position (1,1)
- 40 put in position (0,2)
- 47 put in position (1,2)
- 79 put in position (0,3)
- 47 put in position (1,3)

This would fill the empty E% list as follows:

	Column 0	Column 1	Column 2	Column 3	... Column 100
Row 0	40	79	40	79	
Row 1	8	8	47	47	

Look carefully at the charts above to see how the data and positions are being used here. If you do not understand why 40 has been placed in position (0,0), or what position (0,0) is, you will not get the full benefit of this chapter's tool. Study the charts hard. It might help to look back at the original square, with its coordinates and endpoint numbers shown.

To place each endpoint data item in the proper E% positions, a FOR/NEXT loop and a DATA statement are entered:

```

1110 FOR I = 0 TO 3
1120 READ E%(0,I), E%(1,I)
1130 NEXT I
1140 DATA 40, 8, 79, 8, 40, 47, 79, 47

```

Line 1110 begins the loop, setting I to 0. Thus, line 1120 starts by READING:

```
E%(0,0),E%(1,0)
```

These are the two positions in Column 0 of E%. Endpoint #0 data needs to be placed in these positions, so they are the first two coordinates given in line 1140. Each time the loop is gone through, the next two data items in line 1140 are placed in the list. Because I begins at 0 and continues up to 3, the loop is processed four times. Each time it is processed, line 1120 changes to specify another column in the list: E%(0,1),E%(1,1);E%(0,2),E%(1,2); etc. For each pass through the loop, another coordinate pairs is read from the DATA statement and placed in a column.

Several important things to remember about this FOR/NEXT loop and DATA statement are:

- (1) The loop (FOR I = 0 TO ?) must be set to be processed the *correct* number of

times. If there are 20 items on the DATA statement, the loop must read FOR I = 0 to 9. (Remember: The loop reads *two* data items at a time.) You can get incomplete and unexpected shapes if this loop is not set correctly.

- (2) The READ statement fills the list with DATA, *two* data items at a time (one X coordinate and one Y coordinate). It does this using the variable "I" to specify each column position in the list. As long as you do not change this READ statement in any of your programs, it will always fill the E% list.
- (3) To enter the DATA statement, *copy* the endpoint data list of your shape. Copy the coordinates column-by-column, starting with endpoint #0. The X coordinate of each endpoint should always be entered before the Y coordinate. This is how it was done for the square:

```

end.0   end.1   end.2   end.3
  X,Y,   X,Y,   X,Y,   X,Y
1140 DATA 40, 8, 79, 8, 40, 47, 79, 47
  
```

Since the endpoints are numbered from 0 to 3, the loop is set accordingly:

```
1110 FOR I = 0 TO 3
```

That's all there is to creating an E% list. Putting the line data list in memory is just as easy. Let's look at that list again:

	line 0	line 1	line 2	line 3
"FROM" Endpt. #	0	1	2	3
"TO" Endpt. #	1	2	3	0

The size (dimension) and name of this list must be entered in the program. The new tool expects this list to be called L% (for *Line* data). Can you figure out the minimum DIM statement that would work for this list? The list needs 2 rows and at least 4 columns, so the smallest dimensions we would enter would be:

```
1200 DIM L$(1,3)
```

To leave plenty of room for other shapes of varying sizes, this list will be given 100 columns:

```
1200 DIM L$(1,99)
```

Storing the line data in L% is done with the following main routine lines:

```

1210 FOR I = 0 TO 3
1220 READ L$(0,I), L$(1,I)
1230 NEXT I
1240 DATA 0, 1, 1, 2, 2, 3, 3, 0
  
```

These lines have the computer store your line data in the L% list. The loop is set to process once for each “from” and “to” set of data items—*no more and no less*. If your handwritten list contains line #0 through line #12, then the loop must be set at FOR I = 0 TO 12.

Line 1220 fills L% with the data items in line 1240. Each time the loop is processed, line 1220 reads *two* data items from the DATA statement. (Note: If a DATA statement ever has an uneven number of data items, something is wrong.) The data items are placed column-by-column in the L% list.

Line 1240 must be a *copy* of your line data list, set up in this form:

	line 0	line 1	line 2	line 3
	From, To,	From, To,	From, To,	From, To
1240 DATA	⏟	⏟	⏟	⏟
	0, 1,	1, 2,	2, 3,	3, 0

Of course, all of the program line numbers are arbitrary. You may use any program lines available in whatever program you work with. It is how you set up the program lines that is so important.

Enough reading. You need to put all of this into practice to fully understand any of it.

Test Plotting

To start, you will enter the DRAW A SHAPE tool and then test plot a shape. Since the new tool uses the PLOT A POINT and PLOT A LINE tools, load Chapter 4’s *program* into memory. List lines 1000 through 1400 on your screen.

The first three lines in the main routine are necessary to see the shape you plot. The rest of the main routine, though, is of no use right now. A quick adjustment to the ZAP routine will keep lines 1100, 1110 and 1120 from being zapped when you RUN 10. List lines 10 and 11.

These are the first two lines in the ZAP routine. Line 11 reads:

11 A = 256: B = 2049: C = 1003

The “C = 1003” specifies where to *start* zapping lines. Carefully re-type this line as follows:

11 A = 256: B = 2049: C = 1200

This minor change leaves all lines before line 1200 untouched by the ZAP routine. Type RUN 10 and press **RETURN**.

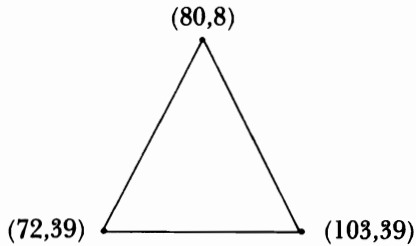
Wait as the main routine is zapped, and then list your program to check for tools. (There should be several.) Also, check to see if the main routine contains *only* lines 1100, 1110, and 1120. If anything is amiss, re-load Chapter 4’s program and try again. When everything is in order, type the DRAW A SHAPE tool below:

```

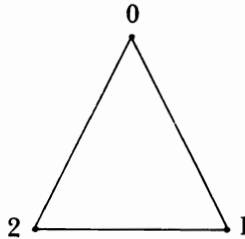
110 REM::::::::::DRAW A SHAPE
111 FOR J = 0 TO NL
112 E1 = L%(0,J): E2 = L%(1,J)
113 X1=E%(0,E1) + X0: Y1=E%(1,E1) + Y0
114 X2=E%(0,E2) + X0: Y2=E%(1,E2) + Y0
115 GOSUB 80
116 NEXT J
117 RETURN

```

Re-check each line before continuing. To make sure you entered all lines correctly, you need to draw a shape. Let's try this triangle:



To gather the two data lists, assign a number to each endpoint:



You can then easily fill in the data lists:

	endpt. 0	endpt. 1	endpt. 2
X	88	103	72
Y	8	39	39

(coordinates of each endpoint in the shape)

	line 0	line 1	line 2
"FROM" Endpt. #	0	1	2
"TO" Endpt. #	1	2	0

(lines made "from" one endpoint "to" another, using endpoint numbers assigned above)

To enter these lists into the program, a DIM statement is needed. Type the following:

```
1200 DIM E%(1,99), L%(1,99)
```

Notice that you can give the dimensions for both lists in the same program line. For each one, you have created 2 rows and 100 columns (1,99). To fill E% with the correct endpoint coordinates, type:

```
1300 REM::::::::::TRIANGLE ENDPOINTS
1310 FOR I = 0 TO 2
1320 READ E%(0,I), E%(1,I)
1330 NEXT I
1340 DATA 88, 8, 103, 39, 72, 39
```

The endpoints range from #0 to #2, so the loop is set from 0 TO 2. The DATA items from the endpoint list are typed in line 1340, starting with the X and Y coordinates of endpoint #0. Line 1320 READS the data items into the appropriate list positions.

Type the following to create the line data list:

```
1400 REM::::::::::TRIANGLE LINES
1410 FOR I = 0 TO 2
1420 READ L%(0,I), L%(1,I)
1430 NEXT I
1440 DATA 0, 1, 1, 2, 2, 0
```

The loop is set to process 3 times again. (Note: The E% and L% loops will frequently need to be *different* lengths.) Line 1420 has the computer READ data items into the L% list. The data items appear in line 1440, starting with the “from” and “to” endpoints of line #0.

To have the DRAW A SHAPE tool look at these lists, type:

```
1500 REM::::::::::DRAW TRIANGLE
1510 C = 14
1520 NL=2: X0=0: Y0=0
1530 GOSUB 110
```

Line 1520 is crucial to test plotting a shap. It will be explained after you test the program. RUN it now.

Wait as the screen is changed to black on blue, and all pixels are set to the background color. After a moment, the triangle should be plotted near the top, left-hand corner. When it has been plotted (or if you run into problems), press **RUN/STOP** and tap **RESTORE**.

If lines were plotted, but not in a triangular pattern, check the DATA statements for both E% and L%. If you can, ask someone else to read the program lines from this book as you check your screen. Check line 1520, making sure NL is equal to 2, and X0 and Y0 are equal to 0. Check the FOR/NEXT loops to be sure they are processed from 0 TO 2. If the main routine is correct, plow through the subroutine. Don't continue until you get the triangle plotted correctly.

List lines 1500 through 1530 on your screen. Line 1510 sets the color code to black against blue. The screen already was black on blue, but specifying C is a good habit to get into.

Line 1520 is of primary interest. First, it sets the value of NL. NL is a variable (place-holder) for the number of lines the shape to be drawn has. This is a 0-based number, and thus a 3-sided triangle is drawn with NL set equal to 2. The NL variable is used in subroutine 110, and so it *must* be set each time the subroutine is called. To make it easy on yourself, look at the last line # on your handwritten line list. Set NL equal to that line number.

Line 1520 also contains what are known as "offset values." These X0 and Y0 values will allow you to test plot and move your shape. Very simply, the value you enter for X0 will be *added* to all X coordinates in your endpoint data list. This will adjust the shape so that it is plotted to the right or to the left of its original screen location. For example, suppose you used this tool to plot a straight line from endpoint 3,3 to endpoint 20,50. If you make X0 equal 5 (X0=5), the computer will *add* 5 to each X coordinate *before* the line is plotted. If, instead, you want to move the line *left* 5 columns, X0 would have to equal -5. This is because you need 5 *subtracted* from each X coordinate to move the line left. Remember this rule:

—X0 moves the shape *left*; +X0 moves the shape *right*

Changing Y0 moves the shape up or down. If a line were plotted from 8,3 to 25,50, and you felt it should be moved *down* two rows, you would make Y0=2. This adds 2 to each Y coordinate, plotting the line from 8,5 to 25,52 (two rows down on the screen). For Y0, the rule is:

—Y0 moves the shape *up*; +Y0 moves the shape *down*

Let's test plot the triangle by moving it 32 columns right and 32 rows down. Type:

```
1520 NL=2: X0=32: Y0=32
```

RUN the program again. After the screen is cleared, the triangle should be plotted in its new location (over and down).

It is important to keep the X and Y coordinate ranges in mind when setting the X0 and Y0 values. Wherever you move your shape, all of its points *must* remain within these ranges. If your original shape has an X coordinate of 10, and you decide to move the shape to the left 12 columns (X0=-12), you will cause yourself problems. The original X coordinate of 10, when added to -12, becomes -2. This, of course, is outside the X coordinate range. This is not allowed.

Finally, remember to set the values of NL, X0 and Y0 *each* time you draw a shape with the DRAW A SHAPE tool. This is important. X0 and Y0 can be set to 0 (which does not move the shape in any direction), but they should be set. This is because as soon as you set these variables and call the DRAW A SHAPE tool, the variables stay set until you specifically change them again. If your program uses this tool to draw more than one shape, all of the shapes will be drawn according to the most recent values assigned to NL, X0 and Y0.

Duplicating Shapes

Understanding how to duplicate a shape will be very simple. All you need are *new* offset values, a new color code (if desired), and GOSUB 110 statements (one for each copy of the shape you want).

List your main routine on the screen to review what you've done so far. Briefly:

- line 1200 reserves space in memory for the two lists
- lines 1300-1340 fill E% with a description of the triangle's endpoints
- lines 1400-1440 fill L% with a description of the triangle's lines
- lines 1500-1530 give the number of lines in the shape, set the offset values, keep the color code at 14, and call the DRAW A SHAPE subroutine

The description of the triangle will remain in E% and L% until you fill the lists with another shape's description. By typing new offset values and calling the subroutine again, the shape can be duplicated as many times as you like. To see what we mean, add the following lines to your program:

```
1540 NL=2: X0 = 16: Y0 = 16
1550 GOSUB 110
1560 NL=2: X0 = 8: Y0 = 8
1570 GOSUB 110
```

RUN the program to see what happens. It will take a few moments, but you will end up with three triangles—one for each GOSUB 110 statement in the program. Each triangle is plotted at an offset location relative to the *original* triangle described in E% and L%. The original triangle is *not* plotted on your screen. It is important to remember that offset values do not offset a shape from the last copy of that shape that was plotted. Instead, offset values offset a shape from the shape stored in E% and L%. This will be an easy thing to forget.

As long as you have a shape stored in E% and L%, Tool 110 can draw it. The NL variable must be set equal to the number of lines in the shape, based on a 0-based scale. Offset values can be used to move the shape or duplicate it anywhere on the screen. To move the shape, set X0 equal to the number of columns right (+) or left (-) the shape is to be moved. Set Y0 equal to the number of rows down (+) or up (-) the shape is to be moved. Then RUN the program again.

To duplicate a shape, you need a GOSUB 110 statement for each copy of the shape desired. X0 and Y0 must be changed before each GOSUB to indicate where each new copy is to be placed. Set NL equal to the last line # in the shape.

You should spend some time practicing with this new tool. Try moving the triangle around, and duplicating it across the screen. Enter new program lines that will fill E% and L% with a new shape. Program lines can be no longer than 2 screen lines (80 characters) in length, so DATA statements often have to be broken up into more than one program line. This is fine, as long as each line starts with the word DATA.

If you store a new shape in E% and L% using new program lines, change X0 and Y0 back to 0. Otherwise, the new shape will be plotted 16 rows and columns off (16 is currently the last value given for X0 and Y0).

The rest of the chapter shows you how to plot the ship and seagulls using the DRAW A SHAPE tool. Because the ship is comprised of 68 plotted lines, you will learn a lot about this new tool before the end of this chapter. It is a good idea to try plotting the ship before using this tool for any serious work on your own.

Drawing and Placing the Ship's Hull

You need to do three things to get ready for this chapter's program. First you need to load Chapter 4's *picture* into memory. This involves LOADING "CHAPTER 4.PIC", and LOADING "CHAPTER 4.COL". Each LOAD command should end with an addition of ",1". Refer to the last chapter if you need help on loading these files.

When you think the picture has been loaded properly, type GOSUB 20 and press **RETURN**. You should find the land, water, waves, sky, and lighthouse in place. Do not continue until you actually see this on the screen. Press **RUN/STOP** and tap **RESTORE** to get back to text mode.

You need to run the ZAP routine to get ready for Chapter 5's program. Before running it, change line 11 to the following:

```
11 A=256: B=2049: C=1003
```

Now, type RUN 10 and press **RETURN**. Wait while the main routine is removed from the program. Next, add the following program lines to allow you to enter and exit high resolution graphics:

```
1010 GOSUB 20: REM GRAPHICS
5000 GET A$
5010 IF A$ = " " THEN 6000
5020 GOTO 5000
6000 GOSUB 30
6010 END
```

You will notice that we did *not* add a GOSUB 40 and GOSUB 50 statement to this program. These tools *clear* the high resolution screen of any image it may contain. Because you want Chapter 4's picture to stay on the screen, these GOSUB statements are intentionally left out.

Now you are ready to draw the hull of the ship. Start by typing the DIM statement for the E% and L% lists:

```
1020 DIM E%(1,99), L%(1,99)
```

Follow this with the FOR/NEXT loop to read and place the hull's endpoints into E%:

```
1100 REM::::::::::SHIP ENDPOINTS
1110 FOR I = 0 TO 16
1120 READ E%(0,I),E%(1,I)
1130 NEXT I
```

Before typing the DATA statements below, notice how they are neatly organized. All the commas from one DATA statement to the next are carefully lined up. This may seem a bit picky, but this arrangement helps locate typing errors. By keeping your DATA statements equal in length (all having the same number of data items), and keeping each data item lined up with those above it, it is easy to see if you added or left out a number. Try typing these DATA statements just as we have them listed here:

```
1140 DATA 114, 108, 93, 125, 89, 125
1150 DATA 93, 135, 84, 150, 78, 150
1160 DATA 56, 132, 54, 135, 92, 127
1170 DATA 90, 130, 18, 144, 17, 141
1180 DATA 4, 144, 7, 150, 6, 149
1190 DATA 88, 132, 92, 137
```

Check over your typing before continuing. A group of data statements, one right after the other, is called a "data block." The data block above provides the endpoints for the hull of the ship. These were found by plotting the hull on graph paper and marking down the coordinates of each endpoint.

You now need to type in the "from" and "to" data of the L% list. Type these lines:

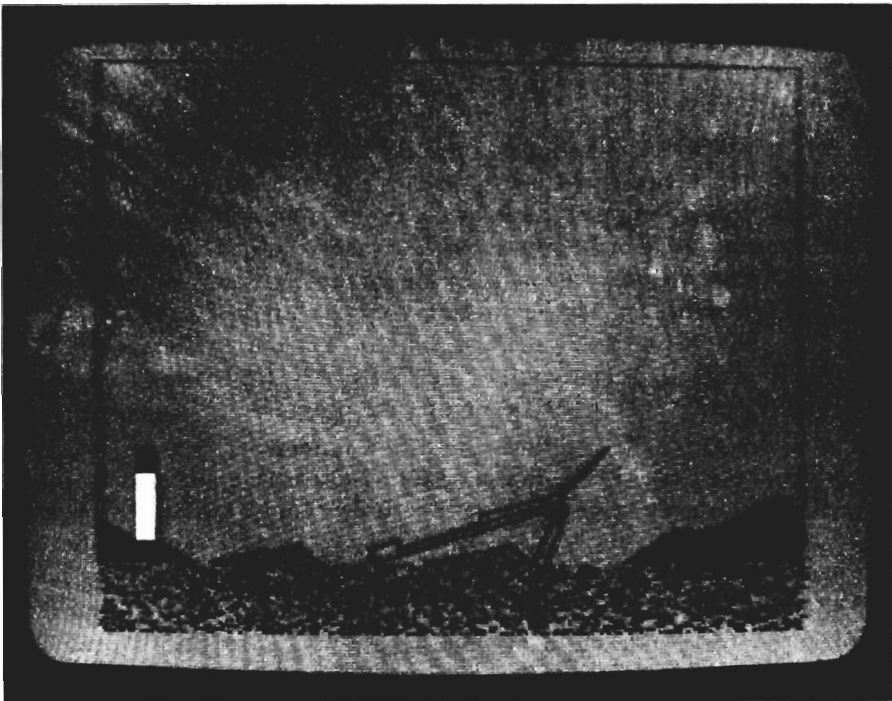
```
1200 REM::::::::::SHIP LINES
1210 FOR I = 0 to 13
1220 READ L%(0,I), L%(1,I)
1230 NEXT I
1240 DATA 0, 1, 0, 2, 1, 3, 1, 5
1250 DATA 2, 6, 3, 4, 3, 9, 6, 7
1260 DATA 8,10, 9,14,10,11,11,12
1270 DATA 12,13,15,16
```

Check over your DATA statements before continuing. Make sure line 1210 makes a loop from 0 to 13.

In addition to specifying the endpoints and lines for the ship, X0 and Y0 need to be set to place the ship appropriately on the screen. Finally, a GOSUB statement which calls the subroutine is required. Type these lines:

```
1300 REM::::::::::DRAW SHIP
1310 C = 14
1320 NL=13: X0=114: Y0=26
1330 GOSUB 110
```

This completes the information needed for the hull. Check each line carefully before running the program. If you can, have someone go over the DATA statements line-by-line with you. When ready, RUN the program. The hull should be plotted as shown below:



The hull will be centered on top of the water. If it gets plotted in the wrong place, check the X0 and Y0 offset values in the program. If only some lines are plotted in the wrong place, or if a line was not plotted at all, check the endpoints data block in lines 1140-1180. There should be a total of 34 numbers in this block. Finally, check

the line data entered for L%. You will have to re-load Chapter 4's picture before running a corrected program.

When the program is working properly, list lines 1020 through 1320. Now that you have used your new tool, you are ready to learn more about READ statements and FOR/NEXT loops. Understanding their "ins and outs" can save you from much bewilderment if your skillfully designed works of art ever become mish-mash on the screen.

READ causes the computer to put data into a specified list. If the statement says "READ L%...", data is placed into the L% list. If the statement says "READ E%...", data is placed in the E% list. How does the computer know *what* data to read and *when* to quit reading it? This may sound like an elementary question, but "read" on...

When the computer comes to a READ statement, it will begin placing the *first* unused data item it finds into the specified list. Thus, you may have a READ statement at line 35400 that reads a DATA statement at line 1. The computer quits reading data items when the loop containing the READ statement has been processed the specified number of times. If the computer runs out of data items to read *before* the loop has been completed, an "OUT OF DATA" error will occur and the program will stop.

In other words, when a program is running, a READ statement reads data items from DATA statements. As each data item is read and placed in a data list, it is "checked off" by the computer. This keeps it from being read more than once. This is why it is important to make your loops exact. For example, if you make the loop that reads E% data too long, data items meant for the L% list will be read into the E% list—regardless of the fact that they are not grouped with the E% DATA statements. Then, when the L% list gets filled, those first few data items will have already been "checked off" and will *not* be a part of the L% list. The L% list will come up short, an "OUT OF DATA" error will occur, and the program will stop.

READ statements will simply read DATA statements from the lowest line number up to the highest line number. How the DATA statements are grouped, where they are located, or what section they appear under has no bearing on how they are read. As each item is read and placed in a list, it is checked off—never to be read during the same program execution again. This is all key in making sure your loops are set right.

Look at the ship's endpoint data block again:

1140	DATA	114, 108,	93, 125,	89, 125
1150	DATA	93, 135,	84, 150,	78, 150
1160	DATA	56, 132,	54, 135,	92, 127
1170	DATA	90, 130,	18, 144,	7, 141
1180	DATA	4, 144,	7, 150,	6, 149
1190	DATA	88, 132,	92, 137	

TOOL 110:.....DRAW A SHAPE

```

110 REM:.....DRAW A SHAPE
111 FOR J = 0 TO NL
112 E1 = L%(0,J): E2 = L%(1,J)
113 X1=E%(0,E1) + X0: Y1=E%(1,E1) + Y0
114 X2=E%(0,E2) + X0: Y2=E%(1,E2) + Y0
115 GOSUB 80
116 NEXT J
117 RETURN
    
```

What it Does: This tool draws the shape described in lists E% and L%. It places the shape (or duplicates it) at the offset location given in variables X0 and Y0.

Example Use:

- (1) Draw your shape on graph paper.
- (2) Number all the endpoints in your shape, starting with endpoint #0.
- (3) Make a list (E%) of the X,Y coordinates that form each endpoint:

	E%		
	endpt 0	endpt 1	endpt 2 . . .
X	55	60	55
Y	3	10	15

- (4) Make a list (L%) which gives the "FROM" and "TO" data needed to describe the ship's lines:

	L%		
	line 0	line 1	line 2 . . .
"FROM" Endpt. #	3	1	2
"TO" Endpt. #	2	3	6

- (5) Type a DIM statement to reserve space for the E% and L% lists:
1020 DIM E%(1,#),L%(1#)
where # is the maximum number of columns each list will need
- (6) Fill E% with the endpoint data using the following format:

```

1110 FOR I = 0 TO #
where # is the total number of endpoints in the shape being drawn, including endpoint #0
1120 READ E%(1,I),E%(1,I)
1130 NEXT I
    
```

```
1140 DATA X,Y,X,Y
```

where X,Y are replaced with the coordinates of each endpoint, starting with endpoint #0

(7) Fill L% with the line data using the following format:

```
1210 FOR I = 0 TO #
```

where # is the total number of lines to be drawn, including line #0

```
1220 READ L%(0,I),L%(1,I)
```

```
1230 NEXT I
```

```
1240 DATA from,to,from,to
```

where "from,to" are replaced with the endpoint #'s for each line, starting with line #0

(8) Set the values of NL, X0 and Y0:

```
1310 NL=E: X0=F: Y0=G: GOSUB 110
```

E should be the last line # on your handwritten line data list; F is how many columns right the shape is to be move; G is how many rows down the shape is to be moved

Technical Description:

To DRAW A SHAPE, a loop is used:

```
101 FOR J = 0 TO NL
```

NL is the number of lines in the shape (0-based), and this loop will be performed once for each line. The first time through the loop, a line will be drawn between the points represented by the first entry in the L% list:

```
L%(0,0),L%(1,0)
```

The second time through the loop, the second entry in the L% list is used:

```
L%(0,1),L%(1,1)
```

Remember that L%(0,J) and L%(1,J) are the 2 endpoints that determine the line (where J is replaced with a number by the loop). These endpoints were previously stored in the E% list by the main routine.

```
102 E1 = L%(0,J) : E2 = L%(1,J)
```

This line retrieves the endpoint #'s of the first two endpoints of the current line to be drawn.

```
102 X1 = E%(0,E1) + X0 : Y1 = E%(1,E1) + Y0
```

This line looks at E% to find out the actual X,Y coordinates of the first endpoint in the line, and then adds the offset values to it.

```
103 X2 = E%(0,E2) + X0 : Y2 = E%(1,E2) + Y0
```

This line looks at E% to find the actual X,Y coordinates of the second endpoint in the line being drawn. The offset values are added to this endpoint also.

This is all the information necessary to draw a line with the line drawing routine at line 80.

```
104 GOSUB 80  
105 NEXT J
```

This is the bottom of the loop.

```
106 RETURN
```

Line 106 tells the computer to return to the main routine.

Drawing the Front Sails

You will now be able to work more quickly because you have already learned most of this chapter's material. The front sails are plotted in the same manner at the hull of the ship. Type in these DATA statements for the endpoints of the sail:

```
1400 REM::::::::::FRONT SAIL ENDPOINTS  
1410 FOR I = 0 TO 42  
1420 READ E%(0,I),E%(1,I)  
1430 NEXT I  
1440 DATA 96, 7, 91, 28, 92, 29  
1450 DATA 86, 17,105, 29, 78, 22  
1460 DATA 108, 38, 75, 30, 69, 39  
1470 DATA 67, 38,112, 62,111, 54  
1480 DATA 108, 60, 86, 48, 88, 50  
1490 DATA 85, 51, 88, 52, 66, 40  
1500 DATA 122, 72, 62, 54, 42, 75  
1510 DATA 39, 72,117,105,113,103  
1520 DATA 122, 90, 76, 89, 81, 90  
1530 DATA 75, 94, 79, 96, 60, 88  
1540 DATA 115,111, 60,109, 57,120  
1550 DATA 55,119, 81,126, 93,130  
1560 DATA 115,135,110,134,117,126  
1570 DATA 70,123, 76,124, 68,129  
1580 DATA 74,128
```


Now, enter the program lines that will connect these endpoints to form the proper lines:

```
1600 REM:::::FRONT SAIL LINES
1610 FOR I = 0 TO 27
1620 READ L%(0,I),L%(1,I)
1630 NEXT I
1640 DATA 0, 1, 0, 2, 3, 4, 5, 6
1650 DATA 5, 7, 6,11, 7, 8, 9,10
1660 DATA 11,12,13,15,14,16,17,18
1670 DATA 17,19,18,24,19,20,21,22
1680 DATA 23,24,25,27,26,28,29,30
1690 DATA 29,31,30,38,31,32,33,34
1695 DATA 35,36,37,38,39,41,40,42
```

Finally, set the color code and offset values, and then call the subroutine tool:

```
1700 REM:::::DRAW FRONT SAIL
1710 C = 14
1720 NL=42: X0=114: Y0=26
1730 GOSUB 110
```

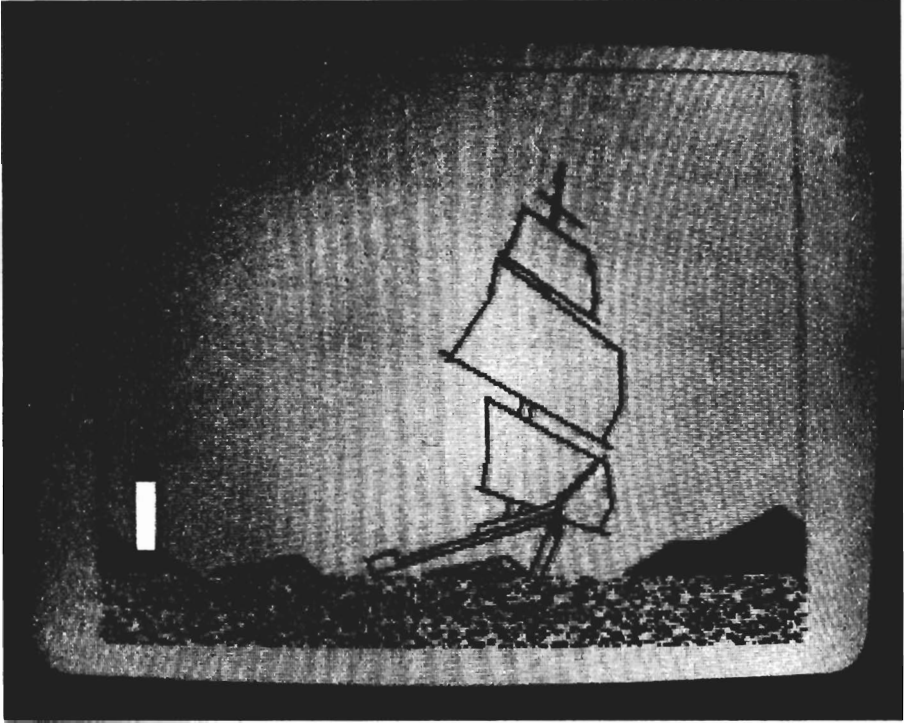
You have probably found out how easy it is to make mistakes in DATA statements. (Unfortunately, it is not quite as easy to find them.) If a program does not run properly, and it contains a long list of DATA statements, you usually know where to look to find the errors.

When running this program, you want the computer to start at line 1400 so you don't have to wait for the hull to be re-drawn. Unfortunately, typing RUN 1400 won't work with DATA statements. Why? Because a READ statement (like that in line 1420) will start reading the *first* un-read data item in the program. This occurs in line 1140 of your program. This DATA statement will be the first one read when the program is run, regardless of which line number the RUN statement sends the computer to.

There is a way around this problem, though. By changing line 1330 to a REM statement (temporarily), the hull shape is stored in E% and L% but *never* plotted. Then lines 1400 through 1695 erase the hull from E% and L%, replacing it with the front sail descriptions. The front sails get plotted by line 1730. Change line 1330 as follows:

```
1330 REM:::GOSUB 110 RETURN
```

RUN the program. The front sails should be plotted as shown in the following illustration:



If you are having any problems, go over the following checklist to locate errors:

Has the high resolution screen “frozen,” plotting *no* part of the sails?

- When this happens, carefully type **GOSUB 30 RETURN**. You will not see your typing on the screen as you type, but you should be returned to text mode. Look for an error message (e.g., “OUT OF DATA ERROR”, “BAD SUBSCRIPT ERROR”, “SYNTAX ERROR”, etc.)

Are the front sail lines being plotted, but in what appears to be random locations?

- Check to see if the loop in line 1210 is processing too many times, and thus reading some of the data items in line 1440 (line 1210 should have **FOR I=0 TO 13**).
- Check the loop in line 1410 (**FOR I=0 TO 42**). Make sure it is not processing too many times and reading too many data items into **E%**.
- Check the line data in lines 1640-1695. Omitting or adding an extra data item here will produce a messed up picture from that point on.

Was only one line plotted off course?

—Check the endpoints data block (lines 1440 through 1580). One number has probably been entered incorrectly.

Was only one line plotted?

—Check to make sure you set NL correctly before GOSUB 110.

Were extra lines added to your picture?

—Check to make sure you set NL correctly before GOSUB 110.

Were the front sails formed, but plotted too far right or left?

—Check the offset values in line 1720. X0 should be set to 14, and Y0 should be set to 26.

Color problems?

—Check line 1710.

If any program lines were entered incorrectly, you need to:

- (1) correct them;
- (2) reload Chapter 4's picture; and
- (3) re-run Chapter 5's program.

When the program is running properly, and the picture is displaying the hull and front sails of the ship, press the **SPACE BAR** to return to text mode.

Drawing the Rear Sails

Carefully type the endpoint data for the rear sails below:

```
1800 REM::::::::::REAR SAIL ENDPOINTS
1810 FOR I = 0 TO 40
1820 READ E%(0,I), E%(1,I)
1830 NEXT I
1840 DATA 78, 12, 75, 20, 73, 15
1850 DATA 80, 20, 61, 13, 78, 22
1860 DATA 60, 18, 48, 25, 44, 22
1870 DATA 69, 39, 52, 30, 50, 38
1880 DATA 39, 45, 37, 44, 60, 56
1890 DATA 99, 59, 99, 55, 59, 57
1900 DATA 37, 48, 33, 69, 20, 84
1910 DATA 13, 80, 60, 102, 102, 106
1920 DATA 100, 99, 28, 87, 30, 96
1930 DATA 24, 105, 21, 102, 58, 116
1940 DATA 15, 81, 25, 141, 41, 111
1950 DATA 49, 114, 33, 140, 42, 138
1960 DATA 54, 115, 39, 139, 48, 137
1970 DATA 12, 105, 63, 127
```

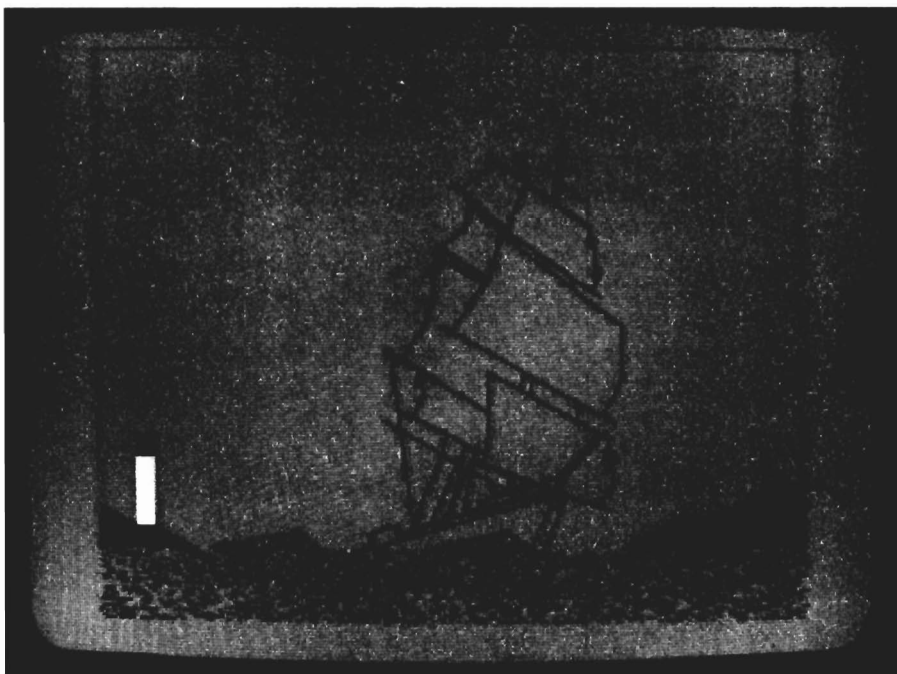
Before adding more lines, check over your typing. Then, put in the line data by typing:

```
2000 REM:::::REAR SAIL LINES
2010 FOR I = 0 TO 25
2020 READ L$(0,I),L$(1,I)
2030 NEXT I
2040 DATA 0, 1, 2, 3, 4, 5, 4, 6
2050 DATA 6, 7, 8, 9, 9,10,10,11
2060 DATA 11,12,13,14,15,16,17,18
2070 DATA 18,19,19,20,21,22,23,24
2080 DATA 25,26,26,27,28,29,30,31
2090 DATA 31,32,33,34,33,35,36,37
2095 DATA 36,38,39,40
2100 REM:::::DRAW REAR SAIL
2110 C = 14
2120 NL=25: X0=114: Y0=26
2130 GOSUB 110
```

Look over your typing. When you are satisfied that it has been entered correctly, change line 1730 to a REM statement so the program won't re-plot the front sails. Type:

```
1730 REM:::GOSUB 110 RETURN
```

RUN the program. It will take a few moments to draw the sails, so sit back and relax. The program should display a picture that looks like this:



If you are having any problems, look back at the checklist previously given.

Drawing and Duplicating the Large Seagull

In this section, you will use Tool 110's duplicating ability to place three large seagulls in the sky. The next section uses the same technique to place two small seagulls in the sky. Using different sizes for the birds adds variety to the design and also creates the sense that the smaller birds are farther back in space than the larger ones.

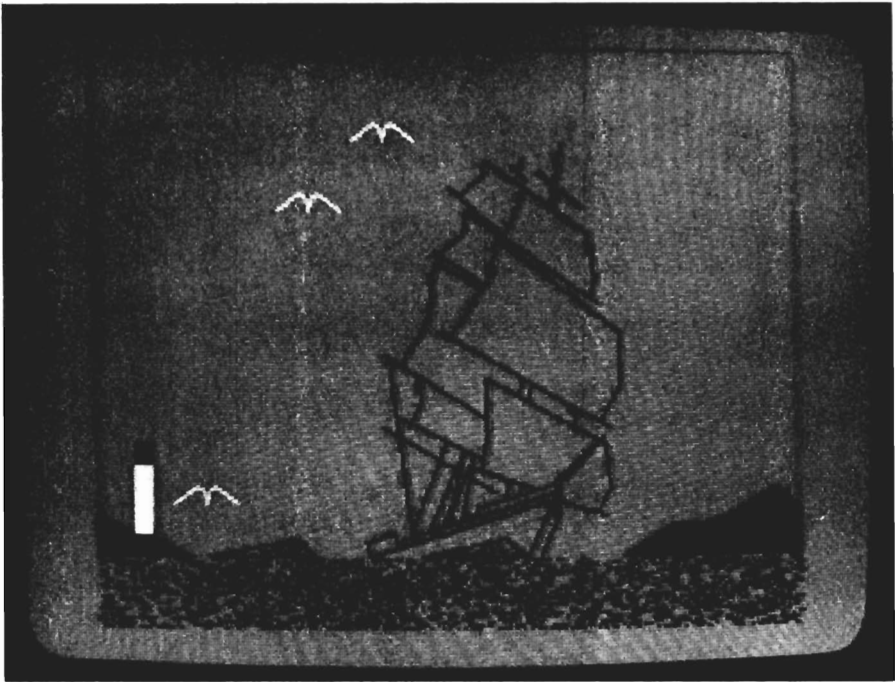
Type these program lines to display the large seagulls:

```
2200 REM:::::LRG SEAGULL ENDPTS
2210 FOR I = 0 TO 6
2220 READ E%(0,I),E%(1,I)
2230 NEXT I
2240 DATA 0, 6,10, 0,13, 2,14
2250 DATA 6,15, 2,18, 0,28, 6
2300 REM:::::LRG SEAGULL LINES
2310 FOR I = 0 TO 5
2320 READ L%(0,I),L%(1,I)
2330 NEXT I
2340 DATA 0, 1, 1, 2, 2, 3, 3, 4
2350 DATA 4, 5, 5, 6
2400 REM:::::DRAW LRG SEAGULLS
2410 C = 30: REM COLOR = WHITE ON BLUE
2420 NL=5: X0=114: Y0=26: GOSUB 110
2430 NL=5: X0=33: Y0=151: GOSUB 110
2440 NL=5: X0=80: Y0=50: GOSUB 110
```

After checking and correcting any errors, change program line 2130 to a REM statement as follows:

```
2130 REM::GOSUB 110 RETURN
```

Then RUN the program. Your picture should display the three large seagulls as illustrated below:



Return to text mode by pressing the **SPACE BAR**. LIST lines 2200-2430. Notice that the endpoint and line data are only entered once for all three birds. To draw the same bird three times, new offset values are entered, each time followed by a GOSUB 110.

Before continuing, change lines 2420, 2430 and 2440 to REM statements. They will then read:

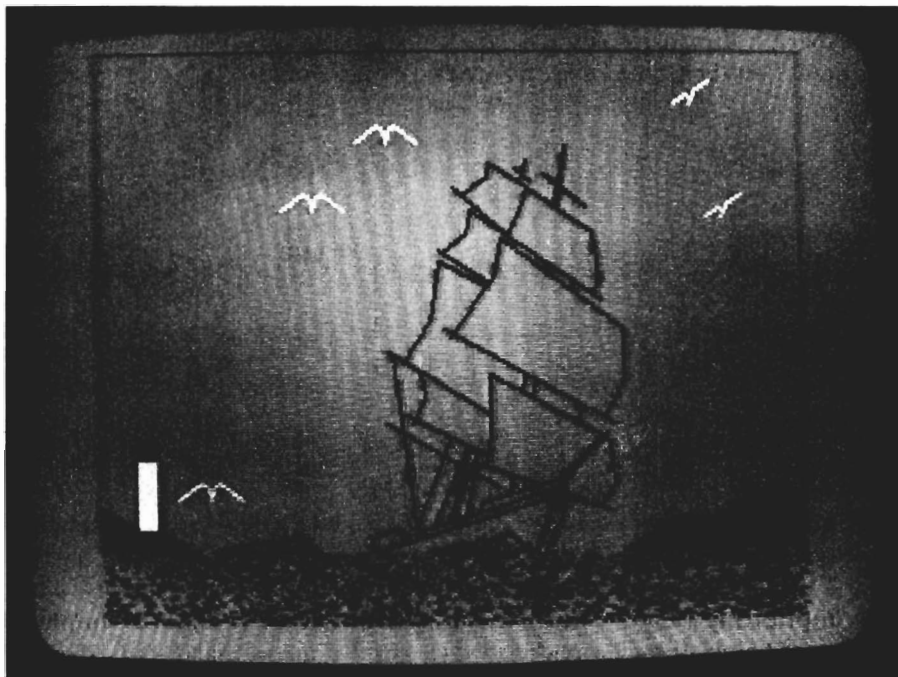
```
2420 REM NL=5: X0=114: Y0=26: GOSUB 110
2430 REM NL=5: X0=33: Y0=151: GOSUB 110
2435 REM NL=5: X0=80: Y0=50: GOSUB 110
```

Drawing and Duplicating the Small Seagull

The small seagulls will be drawn in the same manner as the large seagulls, refilling the E% and L% lists. Type these lines:

```
2500 REM:::::SM SEAGULL ENDPTS
2510 FOR I = 0 TO 5
2520 READ E%(0,I),E%(1,I)
2530 NEXT I
2540 DATA 0,8,5,5,8,6,7,8,9,4,16,0
2600 REM:::::SM SEAGULL LINES
2610 FOR I = 0 TO 3
2620 READ L%(0,I),L%(1,I)
2630 NEXT I
2640 DATA 0, 1, 1, 2, 3, 4, 4, 5
2700 REM:::::DRAW SM SEAGULLS
2710 NL=3: X0=261: Y0=10: GOSUB 110
2720 NL=3: X0=275: Y0=50: GOSUB 110
```

Check your typing and RUN the program. By adding the two small seagulls, you have completed this chapter's program. Check that your screen looks like this:



Before saving this program, go back and *remove* all the REMs inserted in the GOSUB 110 statements. *Leave* the GOSUB 110 statements, just delete the word REM. The lines that you will need to change are:

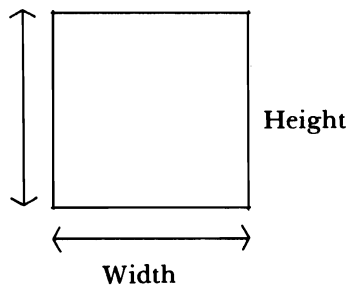
1330
1730
2130
2420
2430
2440

When done, SAVE this program under the filename "CHAPTER 5". *When it is safely stored on disk/tape*, save the picture under "CHAPTER 5" (see Chapter 4 for details on saving pictures).

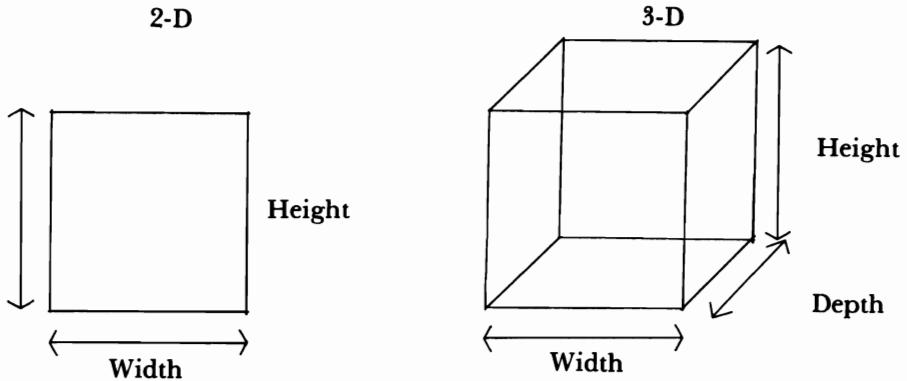
Design Ideas

With your new tool, many complex designs can be easily programmed into the picture. This section deals with some new design techniques that work especially well with the DRAW A SHAPE tool. Following this section you will find this chapter's Summary and two exercises.

A shape, like the one shown below, has height (from lowest Y to highest Y) and width (from lowest X to highest X). Height and width are called "dimensions." If these are the only dimensions that a shape has, it is called a two-dimensional shape. Shapes drawn on the Commodore 64 are two-dimensional.

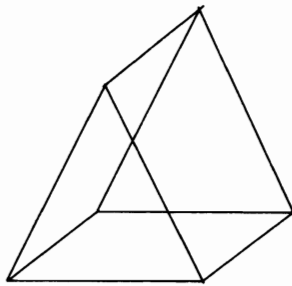


Shapes can have a third dimension of "depth." Shapes having depth are said to be three-dimensional. We live in a world of three-dimensional shapes. There is nothing tangible that does not have height, width, and depth. Pictures that lack the third dimension of depth are deprived of the realism most artists are trying to portray. There is, however, a way to create the illusion of three dimensions. This is done by duplicating a shape, and then connecting a few lines. Take the simple, flat square shown below. It can be made to appear three-dimensional (3-D) by duplicating a copy of it up and over to the right, and then connecting the corners of each copy:

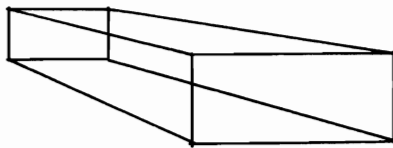


Look closely at the 3-D cube and you'll see that it is made with two squares. These two squares are identical and have been drawn with the same E% and L% lists.

You could also make a triangle look 3-D by using offset values:

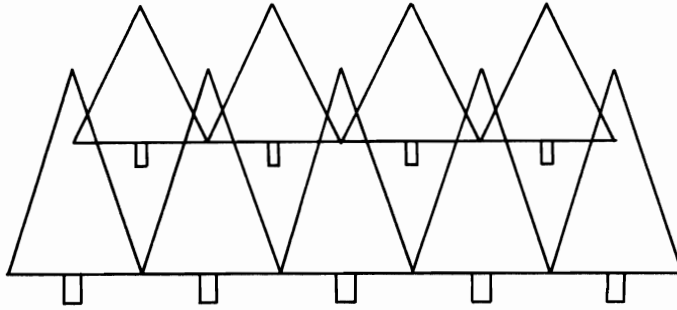


A similar procedure could be used to draw this shape:



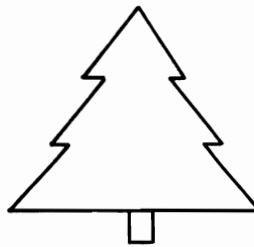
In the shape above, the front rectangle is larger than the back rectangle. This change in proportion is truer to life, as all objects look smaller the farther away they are. (NOTE: The DRAW A SHAPE tool can *not* reduce a shape. You must draw both the large shape *and* then the small shape to produce this proportional picture.)

Shapes that are higher on the screen also appear to be further back in space. You could use this design idea for making a forest of trees, where two different sized trees are duplicated across the screen:

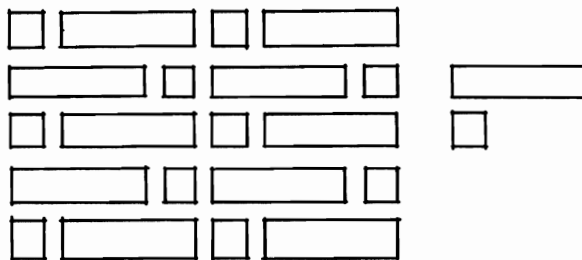


In this example, the trees in the upper row are smaller than the trees in the lower row. Alternating size and location creates an even greater sense of depth.

Interesting designs are made by using different types of shapes, different placements, and different colors. This variety enhances a picture's appearance. Tree shapes could be made more interesting by adding more detail. A tree drawn with more detail might resemble the pine tree in this example.



With the DRAW A SHAPE tool you can quickly create a forest of pine trees. This tool is also useful for making design patterns like a brick wall. A brick wall is made of rectangles which are repeated to create a pattern. The following pattern was made using two different sized bricks:



Some interesting designs and colorful patterns can be made by repeating shapes. With different colors, sizes, and placements of a rectangular shape, you could design a modern city full of buildings. With a circle shape you could design an

outerspace scene of planets. With an assortment of shapes you could make a bouquet of flowers.

As you can see, the **DRAW A SHAPE** tool is great for making all sorts of patterns, shapes, and designs. By practicing and exploring, you should be able to discover and create all kinds of interesting and varied designs.

Summary

Take a bow! (Pun intended.) These last two chapters have been the most difficult yet. Having reached this point, you are probably anxious to begin transferring your own ideas onto the screen. Hold off on any major work, though, until you've gone through this book's last chapter—you'll be glad you did.

The **DRAW A SHAPE** tool was this chapter's addition to your tool kit. Even though you know how to use it, you may be tempted to view this tool as more trouble than it's worth. After all, the **PLOT A LINE** tool can draw shapes, and it's an easier tool to use. Right? Not always. Depending on the shape being drawn and how many times it appears in the picture, the **DRAW A SHAPE** tool could save you a great deal of time and energy. If you set this tool aside as an "extra", you will forfeit the use of two very practical functions: test plotting and shape duplication.

Important things to remember about using your new tool are:

- (1) The I loop must be set to process the correct number of times. For the E% list, I should process from 0 TO the last endpoint # in the list. For an L% list, I should process from 0 TO the last line # in the list.
- (2) Data items in DATA statements are "checked off" when read into a list. Be sure none are read prematurely by an incorrect I loop.
- (3) To call the subroutine, X0, Y0 and NL must be set.

Complete details on using this tool are provided in the **DRAW A SHAPE** technical box. Refer to that box whenever necessary.

Chapter 6 concludes this book, with complete instructions on creating and controlling "sprites." A sprite is any small object or shape designed to move around the screen. This capability provides animation to your picture, and is an enjoyable finale for your graphic lessons.

Below are two example exercises to test your skill with the **DRAW A SHAPE** tool. Try each one before moving on to Chapter 6.

Exercise 1

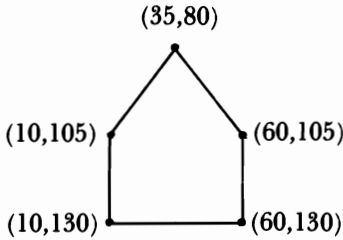
RUN the ZAP routine (type **RUN 10 RETURN**). Now, enter these program lines to clear your screen and set it up for plotting:

```

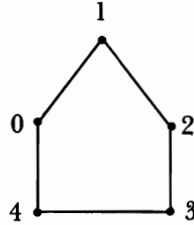
1010 GOSUB 20      : REM GRAPHICS
1020 C=14: GOSUB 40: REM COLORS
1030 GOSUB 50      : REM PAINT BKGD

```

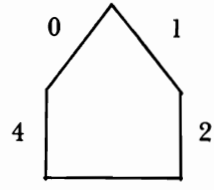
Your first exercise is to enter the program lines necessary to draw the shape below. Draw it using the DRAW A SHAPE tool and E% and L% lists. The X,Y coordinates are given, as well as the endpoint numbers and line numbers. Start at program line number 1100, and set all offset values to 0.



COORDINATES



ENDPOINT #'S



LINE #'S

Solution 1

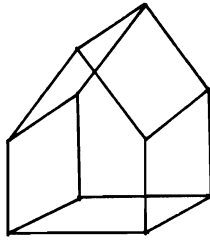
```

1100 DIM E%(1,99), L%(1,99)
1200 REM:::::HOUSE ENDPOINTS
1210 FOR I = 0 TO 4
1220 READ E%(0,I),E%(1,I)
1230 NEXT I
1240 DATA 10,105, 35, 80, 60,105
1250 DATA 60,130, 10,130
1260 REM:::::HOUSE LINES
1270 FOR I = 0 TO 4
1280 READ L%(0,I),L%(1,I)
1290 NEXT I
1300 DATA 0,1,1,2,2,3,3,4,4,0
1310 REM:::::DRAW HOUSE
1320 C=14
1330 NL=4: X0=0: Y0=0
1340 GOSUB 110

```

Exercise 2

Make your house look 3-dimensional by plotting a *duplicate* copy 25 columns right and 10 rows above the original. Then, connect the corner endpoints of each as shown below:



HINT: The lines which will connect the endpoints between each copy are drawn most easily with the PLOT A LINE tool. To figure out the X,Y coordinates of the endpoints on the duplicate copy, *add 25 to each X of the original copy, and subtract 10 from each Y of the original copy.*

Solution 2

```
1410 NL=4: X0=25:Y0=10:GOSUB 110
1420 REM:::::DRAW LINES
1430 X1=10:Y1=130
1440 X2=35:Y2=120:GOSUB 80
1450 X1=10:Y1=105
1460 X2=35:Y2=95:GOSUB 80
1470 X1=35:Y1=80
1480 X2=60:Y2=70:GOSUB 80
1490 X1=60:Y1=105
1400 X2=85:Y2=95:GOSUB 80
1410 X1=60:Y1=130
1420 X2=85:Y2=120:GOSUB 80
```


Chapter 6

MAKING AND MOVING SPRITES

This chapter will teach you about about sprite graphics—one of the most exciting and easy-to-use graphic techniques yet. Sprites are small plotted objects or cartoon-like figures that can be moved around on the screen. They are exciting because they add animation to your picture. They are easy to create and manipulate because almost all of the work is handled by GOSUB statements. This chapter takes you step-by-step through the process of making and moving a sun sprite across the blue sky in your picture.

To get started, the high resolution screen should display the ship, sails, land, water, waves, seagulls and lighthouse. If you see this display on the screen, then skip to the next paragraph. Otherwise, LOAD the picture pattern from Chapter 5 (“CHAPTER 5.PIC”) and the picture colors from Chapter 5 (“CHAPTER 5.COL”). Then, LOAD the program from Chapter 5 (“CHAPTER 5”). To see the picture, type: GOSUB 20 and press RETURN. To get back to text mode, press the RUN/STOP and tap RESTORE. Continue to the next paragraph *only* after Chapter 5’s picture is properly displayed on the high resolution screen.

If you have the Commodore 64 *Programmer’s Reference Guide* or the *User’s Guide*, then you have probably read the chapter covering sprites. In these books, a hot air balloon sprite is moved on the screen by running the corresponding BASIC program. In the next section, you will find out not only what a sprite is and how to make one, but, also, all of the exciting features of sprites.

Introduction to Sprites

What is a Sprite?

A sprite is like one of the little moving figures on a video arcade screen. It is a small shape that can be moved on the screen to create a cartoon-like picture of animation. “Animation” means that the picture shows movement. The versatility of sprites makes them different from any other shape you have previously plotted.

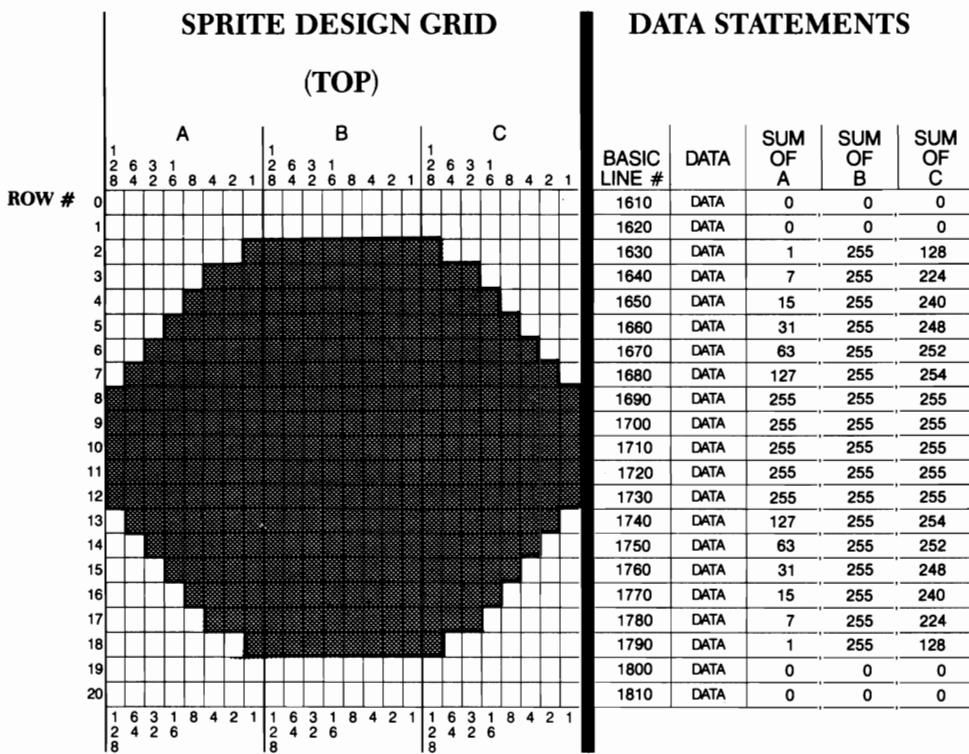
Think of a sprite as a small cut-out figure which can be moved around the screen independently of any other figure already in the picture. It can move up, down, right, left, and diagonally. It can move behind other objects or in front of them. It can fade off or onto the screen. It can even move into a color block with no adverse affect on that portion of your picture. Each sprite is a separate, individual image which acts independently of any other plotted shape, line, point, or even other sprite.

You can design, paint, enlarge, and *move* your sprites. In this chapter, you will be moving a yellow sun sprite across the sky in your picture. All the special features

of sprites will be covered in detail. The next section discusses the various stages in designing a sprite. If you have worked with the hot air balloon sprite in the *User's Guide*, you will see that the sun sprite is made in the same way.

Designing a Sprite

Designing a sprite is similar to drawing and designing your main picture. It is done on a grid (graph sheet), where each little square represents one pixel. Sprites must be designed within a block of 504 pixels. This block of pixels, called a "Sprite Design Grid," is 24 pixels wide and 21 pixels high (24 x 21 = 504 pixels). The sprite's image—what it will look like—is defined inside this grid. Let's take a look at one such grid already shaded for our sun sprite.

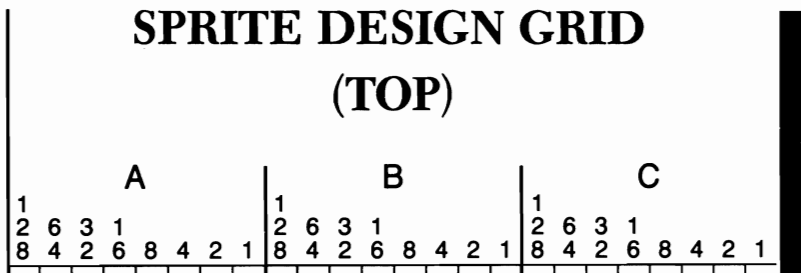


A sprite is originally designed on a Sprite Design Grid. (The Appendix contains a blank Sprite Design Grid to make copies of when designing your own sprites.) The first step in designing a sprite is to lightly pencil sketch an outline of it on the Sprite Design Grid. Make sure that the grid pattern shows through your sketch. Then, lightly shade the squares inside the sprite sketch. It's easiest to outline the shape first, and then shade in the squares. Two important rules to keep in mind when designing a sprite are:

- (1) Each square on the grid represents one pixel on your screen, so your design should *not* cut through any squares. If your sprite's design falls into a square, shade in the *entire* square.
- (2) *A sprite can be only one color.* The sun sprite, for example, will be solid yellow.

Once the sprite has been sketched and shaded, DATA statements that describe the sprite should be gathered. This is the only part to making and moving sprites that will take much concentration at all. Pay careful attention to the next few paragraphs.

Look at the top of the Sprite Design Grid as illustrated below. This grid is divided up vertically (up and down) into three sections. These sections are labeled A, B and C. Each section contains 8 pixel columns. For each section, the columns are numbered: 128, 64, 32, 16, 8, 4, 2 and 1.



These numbers are the key to gathering the necessary data statements. Each number shows the value assigned to *each* shaded pixel in its column. Thus, each *shaded* pixel in the first column has a value of 128. Look back to the sun design. For the first column this would involve the pixels in rows 8 through 12. Each of these five pixels has a value of 128. Each shaded pixel in the second column has a value of 64. The shaded pixels in the third column each have a value of 32. And so on. Notice our emphasis on *each* shaded pixel—we are not talking about an entire column having a value of 128 or 64 or 32 or whatever. Also notice our emphasis on *shaded* pixels. If a square (pixel) is not shaded, it has a value of zero (0) because it is not a part of the sprite. You will not be able to understand anything else about sprites if you don't understand the values associated with each pixel in your design.

For each *row* in the grid, you must compute three totals using these values. First, you must total the values for the 8 squares in section A. Next, you must total the values for the 8 squares in section B. Finally, you must total the values for the 8

squares in section C. These three totals are written to the right of each row, under SUM OF A, SUM OF B and SUM OF C.

These “sums” are your data. Each horizontal row in the grid is equal to three separate pieces of data that the computer can read. The computer will know how to define the sprite from these data sums. Instead of figuring out 504 different numbers for the 504 individual pixels, you only have to determine 63 numbers (21 rows x 3 data sums = 63). Later, these data sums are typed into the program as data statements.

Using the design for the sun sprite, let’s see how the 63 data sums were found.

SPRITE DESIGN GRID																					DATA STATEMENTS									
(TOP)																														
A							B							C							BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C					
1	2	6	3	1	8	4	2	6	3	1	8	4	2	6	3	1	8	4	2	6						3	1	8	4	
ROW #	0																								1610	DATA	0	0	0	
	1																									1620	DATA	0	0	0
	2																									1630	DATA	1	255	128
	3																									1640	DATA	7	255	224
	4																									1650	DATA	15	255	240
	5																									1660	DATA	31	255	248
	6																									1670	DATA	63	255	252
	7																									1680	DATA	127	255	254
	8																									1690	DATA	255	255	255
	9																									1700	DATA	255	255	255
	10																									1710	DATA	255	255	255
	11																									1720	DATA	255	255	255
	12																									1730	DATA	255	255	255
	13																									1740	DATA	127	255	254
	14																									1750	DATA	63	255	252
	15																									1760	DATA	31	255	248
	16																									1770	DATA	15	255	240
	17																									1780	DATA	7	255	224
	18																									1790	DATA	1	255	128
	19																									1800	DATA	0	0	0
	20																									1810	DATA	0	0	0
		1	6	3	1	8	4	2	1	1	6	3	1	8	4	2	1	1	6	3	1	8	4	2	1					
		2	4	2	6					2	4	2	6					2	4	2	6									
		8								8								8												

First, it helps to line up a piece of paper along the row you are summing. Each row is numbered on the left side of the grid, from row 0 to row 20. Again, for each row there will be 3 sums—SUM OF A pixels, SUM OF B pixels, and SUM OF C pixels. Each sum is the sum of only the shaded pixels in sections A, B or C. Looking at these three sections in row 0, you can see that all three are blank. A blank section is unshaded. The sum for any blank section is zero (0). On the right side of the grid are 3 areas called SUM OF A, SUM OF B and SUM OF C:

DATA STATEMENTS

BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
1610	DATA	0	0	0
	DATA			
	DATA			
	DATA			
	DATA			

On the same row as the pixels just added (row 0), you would write down a zero (0) as the total for each section.

Sliding the paper down a row, the values for the next row are summed. Again you find that all three sections in this row for the sun sprite are blank. Thus, a 0 is noted as the sum for A, B and C at the end of row 1.

Moving down to row 2, you will see that there are now some shaded squares to sum up. The first shaded square is in the last column of section A. This column has a number 1 at the top. Since there are no other shaded pixels in section A, the SUM OF A for this row is 1. This is noted at the end of the row, under SUM OF A. Look at section B for row 2. In section B, all of the pixels are shaded. The sum of B for row 2 is the result of adding all the numbers listed at the top of each column: $128 + 64 + 16 + 8 + 4 + 2 + 1$, which is 255. Anytime a section within a row has all 8 pixels shaded, the sum of that section for that row will be 255. At the end of row 2, under SUM OF B, the number 255 is written down. This same procedure is followed for section C of row 2. You will see that the only shaded pixel in section C of row 2 has a value of 128. Under SUM OF C for row 2, the number 128 is written.

This straightforward and easy data collection method is used for each sum of each row, down through the grid. *A blank area always has a sum of zero, and an entirely shaded area always has a sum of 255.*

It's important that each sum is added correctly and written by the appropriate row. When you later enter the program to draw the sprite, these sums will be typed as data statements and will tell the computer exactly how the sprite should look.

In your Commodore 64 *User's Guide* you will notice that the data for the balloon sprite is derived in the same way. Following is an example illustrating the balloon and its data:

SPRITE DESIGN GRID																		DATA STATEMENTS					
(TOP)																							
A						B						C						BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C	
1	2	6	3	1		1	2	6	3	1		1	2	6	3	1							
8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1	8	4	2	6	8	4	2	1
ROW # 0																			DATA	0	127	0	
1																			DATA	1	255	192	
2																			DATA	3	255	224	
3																			DATA	3	231	224	
4																			DATA	7	217	240	
5																			DATA	7	223	240	
6																			DATA	7	217	240	
7																			DATA	3	231	224	
8																			DATA	3	255	224	
9																			DATA	3	255	224	
10																			DATA	2	255	160	
11																			DATA	1	127	64	
12																			DATA	1	62	64	
13																			DATA	0	156	128	
14																			DATA	0	156	128	
15																			DATA	0	73	0	
16																			DATA	0	73	0	
17																			DATA	0	62	0	
18																			DATA	0	62	0	
19																			DATA	0	62	0	
20																			DATA	0	28	0	
1	6	3	1	8	4	2	1	1	6	3	1	8	4	2	1	1	6	3	1	8	4	2	1
2	4	2	6					2	4	2	6					2	4	2	6				
8								8								8							

Note the unshaded pixels inside the balloon. These unshaded pixels will display portions of the main picture as the balloon travels across the screen. These pixels are like “windows” in which you can see the underneath images. Through them, you can see other shapes within the picture, other sprites (if you have more in the picture), and the background color. This see-through effect is a truly unique feature of sprites.

At any one time you can have up to eight sprites in the same picture. Each sprite is independent of anything else in the picture, including other sprites. Each sprite can be a different shape, and so there can be up to eight different sets of data. Sprites can also be the same shape, in which case the same data statements are used.

In order for the computer to keep track of the sprites in your picture, each one is labeled. This label is called a *sprite pointer* and is a number from 0 to 7. The sprite pointer “points” to the place in memory where its sprite’s data is stored. Our program reserves eight special locations in memory for storing sprite data.

Once a sprite is defined, it can be colored with one of the sixteen available sprite colors. It can also be enlarged, erased and re-positioned, erased entirely, and even guided around the screen. There can be up to eight sprites on the screen at one time, each of which can be a different shape, size, and color.

Special Features of Sprites

Sprites have several features which are available only to them. This list of features includes:

- define sprite
- turn on/turn off
- X expand/X unexpand
- Y expand/Y unexpand
- combined X and Y expand
- sprite priority over a sprite
- sprite priority over a shape/shape priority over a sprite
- sprite color
- place sprite at X,Y screen location
- move sprite from X1,Y1 to X2,Y2

Although this list needs some explaining, you get the idea of how much control you have over sprites. These features come in packages called (what else?) *subroutine tools*. In fact, you will be adding 12 more tools to your tool kit before this chapter is over. Don't worry. The tools are very short, and will save you a great deal of repetitious work that might otherwise be required without them. A complete discussion of each sprite feature follows.

Define Sprite

To create a sprite, you must first define what it looks like. This is done by entering data statements describing, row-by-row, which pixels should be painted in order to form the sprite. The data statements are read and stored in memory.

Turn On/Turn Off

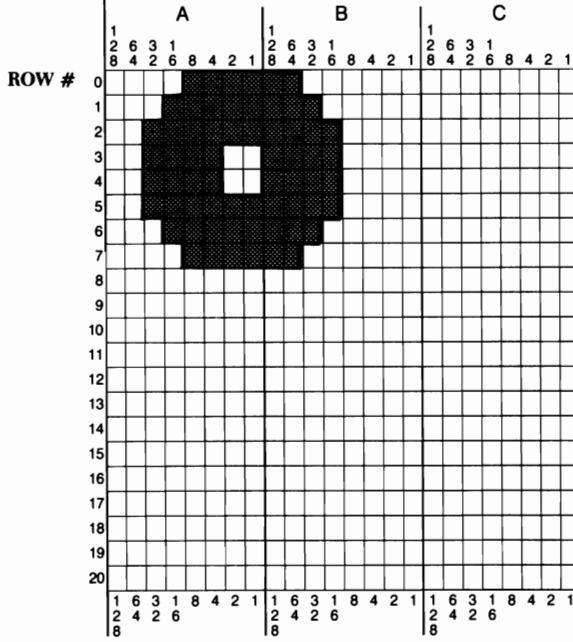
Once a sprite is defined, it must be "turned on." This feature turns on the appearance of a sprite. Note that a sprite must be placed on the screen (see "Place Sprite At X,Y") in order to see it once it is on. Thus, there are three steps to viewing a sprite: defining it, turning it on, and placing it on the screen. To make a sprite disappear from the picture, you would use the "turn off" feature. Turning a sprite on and off is like flipping a light switch. Some interesting visual effects like flashing and blinking can be achieved by alternating between these features.

X Expand/X Unexpand

The "X expand" feature doubles the *width* of the sprite. This is done by duplicating each column in the sprite. The duplicate columns are then alternated with the originals to produce the wider sprite. To understand this better, look at the two diagrams below. The first one shows a sprite in its original form as designed on the sprite grid. The second diagram shows how the columns are duplicated on the screen. (The lightly shaded columns are only shaded as such to point them out as the duplicates. When expanding a sprite, all pixels are plotted in the exact same color.) Notice how the original sprite below does not use the first 2 columns of the sprite grid. When the sprite is made wider, those first 2 columns (even though empty) are duplicated along with the others.

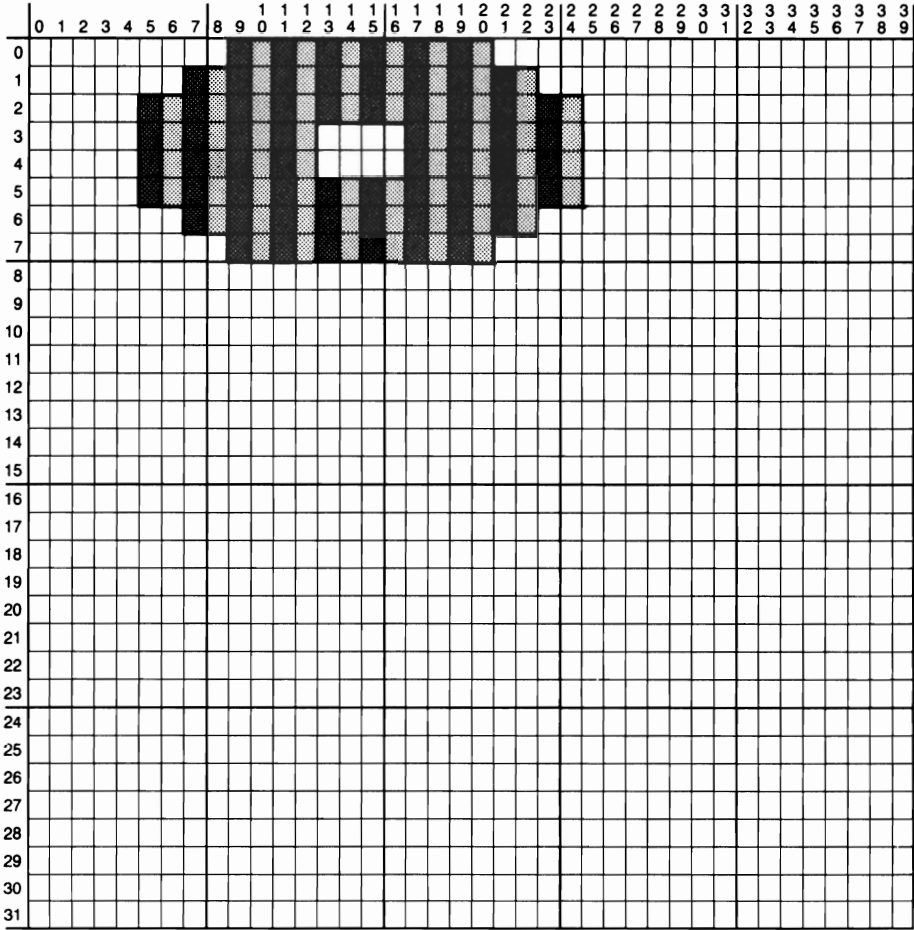
SPRITE DESIGN GRID

(TOP)



DATA STATEMENTS

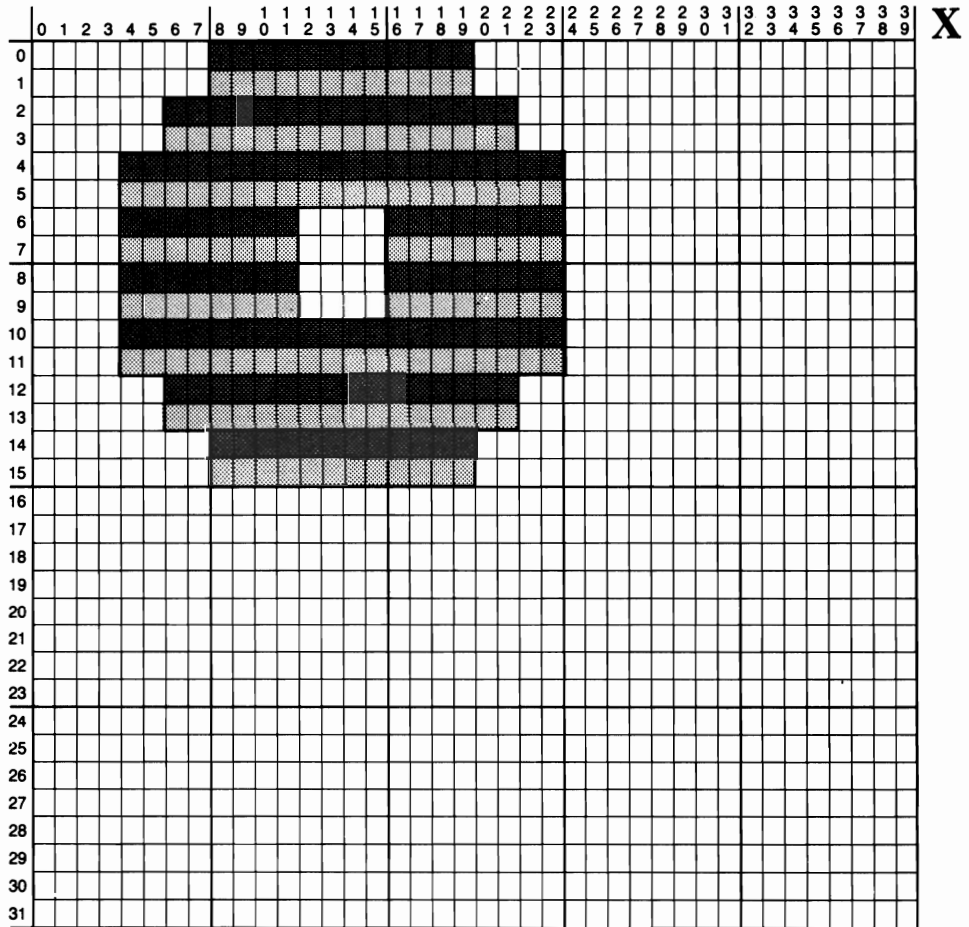
BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C

X**Y**

The “X unexpand” feature removes all the duplicate columns, and thus returns the sprite to its original size. X expand is useful in adding variety to a picture that contains otherwise identical sprites.

Y Expand/Y Unexpand

When “expanded” in Y’s direction, a sprite’s height doubles in size. Each *row* of pixels in the sprite is duplicated. Taking the X expanded sprite in the last diagram, a Y expand would result in:



Y

The “Y unexpand” feature removes the duplicated rows, and thus changes a Y expanded sprite back to its original height.

Combined X and Y Expand

By combining X expand and Y expand, a sprite will double in both width and height. The sprite would resume its original size when both the X unexpand and the Y unexpand features are selected.

Sprite Priority Over a Sprite

A sprite can be made to appear in front of another sprite in a picture. For example, when two moving sprites cross paths, one will appear to pass in front of

the other. This sprite, the one passing in front, is said to have greater screen “priority” than the other. Priority determines the order in which the sprites will stack up visually on the screen. This priority is determined by each sprite’s number. The sprite having the lower number will always have priority over a sprite with a higher number. For example, Sprite 0 has top priority and will always appear in front of other sprites on the screen. Sprite 4 has priority over Sprites 5-7. Sprite 7 has no priority in relation to the other sprites.

Overlapping is particularly successful when the front sprite is defined with an opening or window. Through this window the sprite behind can be seen. Sprite priority should be considered when developing your animation designs, and before you enter the program.

Sprite Priority Over a Shape/Shape Priority Over a Sprite

If you use the “Sprite Priority Over a Shape” feature for the sun sprite, then the sun will appear in front of all other shapes in the picture. As the sun moves across the sky, it would appear to pass in front of the ship and seagulls. This type of sprite priority determines whether a sprite is displayed in front of or behind other plotted shapes. It is an effective feature when used with a sprite that has a “window” in it. As the sprite moves, you will see other shapes and the background color through this hole. If you use the “Shape Priority Over a Sprite” feature, then the sprite would appear behind any other plotted object on the screen.

NOTE: A sprite *always* has priority over the background colors of the screen. When in or passing through a color block, the sprite will always show up in place of any background pixels it comes across.

Sprite Color

The “Sprite Color” paints a sprite with a solid color. It paints the shaded pixels in the Sprite Design Grid which were changed to data numbers. All pixels in the sprite will be displayed in the same color.

A color code should be individually assigned for each sprite, even if all eight sprites are the same color. (If a sprite is not assigned a color in the program, it will automatically be assigned the color black.) There are sixteen sprite colors and color codes (0 through 15):

0 black	4 purple	8 orange	12 medium grey
1 white	5 green	9 brown	13 lite green
2 red	6 blue	10 lite red	14 lite blue
3 cyan	7 yellow	11 dark grey	15 lite grey

Each sprite’s color is independent of any other color block in the picture. It will not influence or change the colors used to plot points, lines or shapes. A sprite can only change the color of a pixel if it passes over it, *and* has priority over it.

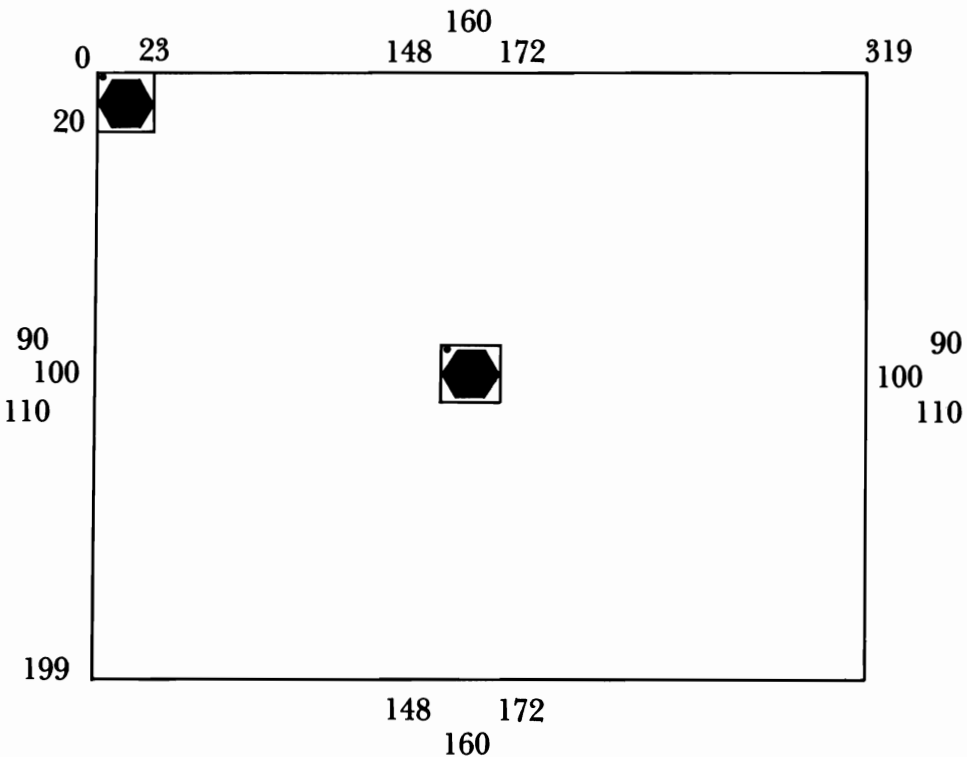
Place Sprite At X,Y

Once you define and “turn on” a sprite, you *must* place it on the screen. A sprite can be placed anywhere on the screen. In fact, sprites can even be placed partially or fully off of the screen.

To place a sprite, you specify the screen location for *the top, left square on the sprite grid*—even if this square is not used by your sprite. This top, left square on the grid is called the *sprite's origin*. Recall that your data statements will describe all 504 squares on this grid. If you correctly place the origin on the screen, the computer can easily place the rest of the squares in relation to it.

To position the origin, an X and a Y value are given in the program. This produces an X,Y coordinate, which is where the origin gets positioned on the screen. X defines the horizontal placement of the origin, and Y defines its vertical placement. Think of a sprite as a piece of paper being pulled around by its origin pixel.

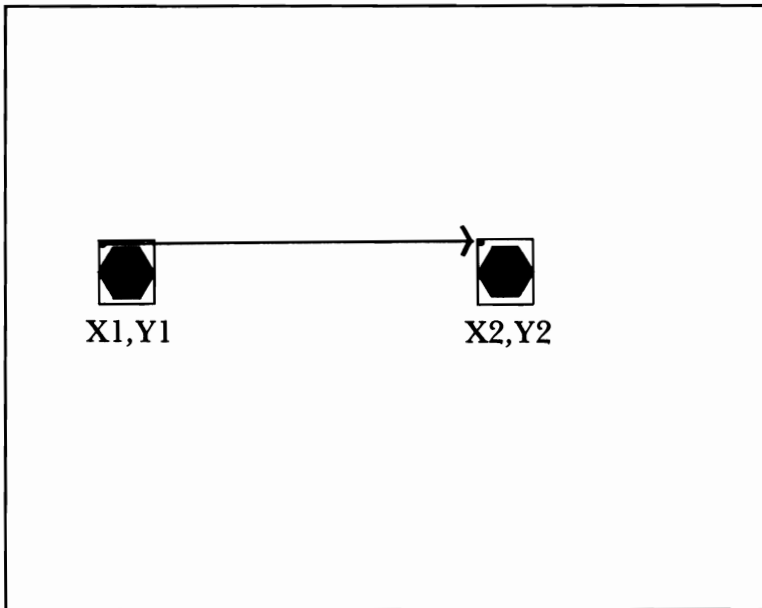
When the origin is re-positioned, the image of the sprite is moved accordingly. Following is an example of two sun sprites placed on the screen. The boxes around each represent the “invisible” grid the sprite was drawn on. The small dot in each box represents the origin square. The sprite in the top, left-hand screen corner has its origin positioned at 0,0. The second sprite has been centered in the screen. To do this, its origin has been placed at position 148,90.



Think of this as if the whole sprite grid will be placed on the screen. The grid will be pulled and moved whenever you re-position the origin square.

Move Sprite From X1,Y1 to X2,Y2

You can move a sprite vertically, horizontally and diagonally across the screen. This means you keep the sprite on the screen at all times, but move it from one location to the next. Its path of movement is like a straight line. To move a sprite, you specify an X1,Y1 location and an X2,Y2 location. The sprite's origin is then positioned at the X1,Y1 location, and it moves in a straight path to the X2,Y2 location. As it does, the rest of the sprite follows. For example, the sun sprite could start where X1=0 and Y1=10, and travel across to where X2=319 and Y2=10. As shown below, it's the sprite's origin that is moved along the linear path.



Sprites can travel in any direction—from left to right, right to left, top to bottom, bottom to the top, and diagonally. The sprite moves in a straight line between the starting (X1,Y1) and ending (X2,Y2) points specified.

When moving a sprite, you can control its speed of movement—how fast it travels. This is not exactly in miles per hour, but sprites can move at a pretty good clip. The rate of speed is expressed numerically by assigning a value to "SD." This value tells the computer how many pixels to skip between each sprite placement. As the sprite moves across the screen, it is actually being erased and re-drawn elsewhere many times over. If SD is set to 10, the computer erases and re-draws the sprite every 10 pixels. When trying out different speeds, be careful. The higher the speed, the more jerky the sprite's movement appears.

With this method of movement, two sprites can *not* be moved at the same time. To move two sprites concurrently, you would have to alternate between them,

creating a loop that erases and then places each one a little bit farther over each time.

In the next section you will have the opportunity to try out these special sprite features. With some practice, you will soon be designing all kinds of interesting animations on your Commodore.

Drawing and Placing the Sun Sprite

With Chapter 5's picture on the high resolution screen, and Chapter 5's program in memory, run the ZAP routine. Type **RUN 10** and press **RETURN**. When the letters stop flashing, type:

```
1010 GOSUB 20
6000 GET A$
6010 IF A$ = " " THEN 6030
6020 GOTO 6000
6030 GOSUB 30
6040 END
```

As in the last chapter, you will be entering this program in sections. After you have typed a section, check your typing for errors. Each section is one step in the process of making and moving the sun sprite. Along the way, suggestions for experimenting with the various sprite features will be made. Try these and any others you might think of as you go along.

Begin by typing the following main routine lines:

```
1100 REM:::::SUN SPRITE
1110 SP = 0:GOSUB 120
1120 GOSUB 130
1130 GOSUB 150
1140 GOSUB 170
1150 GOSUB 200
1160 C = 7:GOSUB 210
1170 X = 232:Y = 10
1180 GOSUB 220
```

The next section contains this sprite's data statements, as shown earlier on the Sprite Design Grid. When you type this section, use three places and a comma for each number listed. For example, you should type the first line as "DATA-ss0,ss0,ss0" (where "s" is a press of the **SPACE BAR**). This method will line up the commas in neat columns. This ordering of the data will help when you check for errors later. Type this section now:

```

2500 REM::::::::::SUN SPRITE DATA
2510 DATA 0, 0, 0
2520 DATA 0, 0, 0
2530 DATA 1,255,128
2540 DATA 7,255,224
2550 DATA 15,255,240
2560 DATA 31,255,248
2570 DATA 63,255,252
2580 DATA 127,255,254
2590 DATA 255,255,255
2600 DATA 255,255,255
2610 DATA 255,255,255
2620 DATA 255,255,255
2630 DATA 255,255,255
2640 DATA 127,255,254
2650 DATA 63,255,252
2660 DATA 31,255,248
2670 DATA 15,255,240
2680 DATA 7,255,224
2690 DATA 1,255,128
3000 DATA 0, 0, 0
3010 DATA 0, 0, 0

```

Check each typed section for errors. In each of the 21 data statements, there should be three numbers. If you missed a number in one line, the line should appear shorter than the others. If you added a number in one line, the line should appear longer than the others. Correct any mistakes, and then type in the first new subroutine:

```

120 REM::::::::::DEFINE SPRITE SP
121 FOR I = 0 TO 62
122 READ A
123 POKE 16384 + 64*SP + I,A
124 NEXT I
125 POKE 18424 + SP,SP
126 RETURN

```

Look over each line. Correct any errors. Rub your eyes a bit, and then continue by typing:

```

130 REM:::TURN ON SPRITE SP
131 POKE 53269, PEEK(53269) OR 2↑SP
132 RETURN
140 REM:::TURN OFF SPRITE SP
141 POKE 53269, PEEK(53269)AND(255-2↑SP)
142 RETURN

```

```

150 REM:::::X EXPAND SPRITE SP
151 POKE 53277,PEEK(53277) OR 2↑SP
152 RETURN
160 REM:::::X UNEXPAND SPRITE SP
161 POKE 53277,PEEK(53277)AND(255-2↑SP)
162 RETURN

```

Take another small break, and again check your typing. When you're ready, type:

```

170 REM:::::Y EXPAND SPRITE SP
171 POKE 53271,PEEK(53271) OR 2↑SP
172 RETURN
180 REM:::::Y UNEXPAND SPRITE SP
181 POKE 53271,PEEK(53271)AND(255-2↑SP)
182 RETURN
190 REM::SPRITE SP PRIORITY OVER SHAPE
191 POKE 53275,PEEK(53275)AND(255-2↑SP)
192 RETURN
200 REM::SHAPE PRIORITY OVER SPRITE SP
201 POKE 53275,PEEK(53275) OR 2↑SP
202 RETURN
210 REM::::SET SPRITE SP TO COLOR C
211 POKE 53287 + SP,C
212 RETURN

```

You are at the home stretch. As soon as you've double checked through line 212, type these subroutine tools:

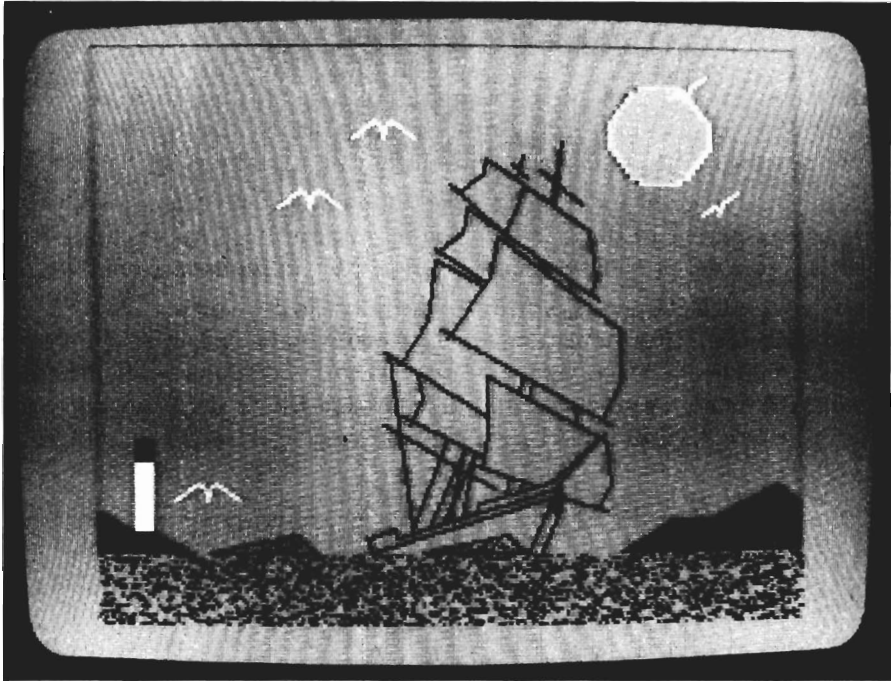
```

220 REM::PLACE SPRITE SP AT X,Y
221 XX = X + 24:YY = Y + 50:Z% = XX/256
222 V = XX - Z%*256:W = 53248 + SP*2
223 WW = 53264
224 PR = ABS((PEEK(WW) AND 2↑SP)<>0)
225 VV = PEEK(WW) AND (255-2↑SP) OR (2↑SP*Z%)
226 IF PR<>Z% THEN GOSUB 140
227 POKE W,V:POKE WW,VV: GOSUB 130
228 POKE 53249 + SP*2,YY
229 RETURN

```

Take one last look at your typing before running this program. Whenever there's this much typing, errors are bound to spring up. After you are finished checking, RUN the program.

If the program is running properly, the yellow sun sprite will appear on the upper right side of the screen:



Look over the image of your sprite. Is it in the same place as ours? It should be a solid yellow color. If you are having any problems, check the following:

If the sprite never appeared:

- check lines 1110 through 1120, which define and turn on the sprite
- check lines 1170 and 1180, which place the sprite on the screen
- check the subroutines in lines 120 through 132

If the sprite was misshapen:

- check your data statements in lines 2510 through 3010 — check the loop in line 121
- check lines 1130 and 1140, which expand the sprite's width and height
- check the subroutines in lines 150-152 and 170-172

If the sprite is the wrong color:

- check line 1160, which sets the color code and calls a tool
- check the subroutine in lines 210-212

You do not have to re-load Chapter 5's picture before running a corrected program. As long as line 1110 has SP=0, and lines 1170-1180 give this sprite's screen position, Sprite 0 will be will be relocated according to the corrected program.

When you see that the program is running correctly, stop it by pressing the **SPACE BAR**. Then, list lines 1100 and 1110.

```

1100 REM::::::::::SUN SPRITE
1110 SP = 0:GOSUB 120

```

Line 1110 actually does three things. First, it states that until SP's value is changed, all program lines dealing with sprites will only affect Sprite 0. The sprite subroutines always affect the sprite whose number is the same as SP's current value.

Second, by giving this sprite the number 0, the sun sprite is given priority over all future sprites placed in this picture. Again, you can have up to eight different sprites in a picture, providing each is given a different number (from 0 to 7).

Finally, GOSUB 120 defines Sprite 0 in memory. To do this, tool 120 READs a set of data statements. It begins reading data statements at the *first* un-read data item in the program. The READ loop is set to read 63 data items, so you must be sure to include all 63 items in your program. LIST lines 120-126. In the following tool box, the subroutine for defining a sprite is explained.

TOOL 120::::::::::DEFINE SPRITE

```

120 REM::::::::::DEFINE SPRITE SP
121 FOR I = 0 TO 62
122 READ A
123 POKE 16384 + 64*SP + I,A
124 NEXT I
125 POKE 18424 + SP,SP
126 RETURN

```

What It Does: This tool defines the shape for the sprite. It reads the sprite data and stores the image in the computer's memory. It does *not* place the sprite on your screen (see Tool 220 for placing sprites).

Example Use: To define a sprite, you must take the following steps:

- (1) Draw the sprite on the "Sprite Design Grid."
- (2) Add up three sums (A,B,C) for the shaded pixels in each row of the grid.
- (3) There should be three numbers (data sums) in each of the 21 rows. In the main routine, type in these data numbers as data statements.
- (4) In the main routine, type a statement to identify the sprite with a number (0-7). Follow this with GOSUB 120. For example: 1110 SP = 0:GOSUB 120

Technical Description: Sprite data is stored at the very beginning of Bank 1. Each sprite requires 64 bytes of memory to store its picture definition. Since there are 8 sprites, 512 free bytes of memory are required (64 x 8 = 512). Since there are 512 bytes at the very beginning of Bank 1, this is a good place to store the sprite data.


```

121 FOR I = 0 TO 62
122 READ A
123 POKE 16384 + 64*SP + I,A
124 NEXT I

```

These lines will first read the sprite data from your data statements. Second, they will store the data in the proper memory location for the specified sprite. Line 123 calculates the proper memory location for each sprite. "16384" is the first memory location in Bank 1. "64*SP" will cause the data to be stored in the proper 64 bytes for that sprite (SP). "I" will be increased for each loop to place the sprite data in consecutive memory locations.

```

125 POKE 18424 + SP,SP

```

Line 125 sets up a pointer which points to the sprite data that was just read in. Each set of sprite data is stored in a block. The first 64 bytes in Bank 1 is Block 0. The second 64 bytes is Block 1.

Each sprite has its own block:

```

Sprite 0 -> Block 0
Sprite 1 -> Block 1
Sprite 2 -> Block 2
(etc.)

```

Line 125 assigns each sprite its own block.

LIST line 1120. This line uses Tool 130 to "turn on" the sprite. This turns the sprite on *in memory*. To see it on the screen, it must also be placed with an X,Y coordinate. Once the sprite is located on the screen, the "turn on" and "turn off" tools will work as switches for viewing/erasing the sprite on the screen. When a sprite is "turned off," it disappears. You can have up to 8 sprites available for a picture. There will be times when some are "turned on" or "off" depending on your animation design. LIST lines 130-132.

TOOL 130:::TURN ON SPRITE

```

130 REM:::TURN ON SPRITE SP
131 POKE 53269,PEEK(53269) OR 2↑SP
132 RETURN

```

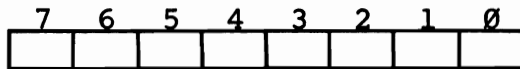
What It Does: This tool "turns on" a sprite in memory.

Example Use: After the sprite is defined using Tool 120, Tool 130 turns it on. If the sprite has been placed on the screen, turning it on will make it show up. To use this tool, make sure SP is equal to the sprite number to turn on, and then enter a GOSUB 130 statement.

Technical Description: First, let's define a new term: register. Most

memory locations are simply storage areas for numbers. A register is a special memory location which performs a special function. The number which is stored in a register can cause very dramatic results. For example, the subroutines at lines 20 and 30 turn on and off high resolution graphics. The memory locations used by those subroutines are examples of registers.

The subroutine lines from 120 to 202 will change some register numbers to manipulate sprites. Some registers affect only one sprite. Other registers affect all eight sprites. The color subroutine at line 200 controls the color for each sprite. This is an example of a subroutine where each sprite has its own register. The subroutines from 120 to 202 are examples of registers which affect all eight sprites. To understand this, let's look at a register. Each register has eight "bits" which are numbered 0-7 from right to left.



Each one of the eight possible sprites is given a bit to control it. Sprite 0 gets bit 0, Sprite 1 gets bit 1, and so on. Each bit can contain either a 0 or a 1. This setup works great for certain sprite features which have only two possible states, such as on or off, and expanded or unexpanded. Either state is determined by the value of the bit.

Register 53269 determines which sprites the computer should currently be working with. A bit flipped to 1 means the features and placement of the corresponding sprite are in effect. All 53269 bits flipped to 0 tell the computer not to display or compile those associate sprites. To change the sprite bits, we need a command that tells the computer something like:

CHANGE BIT 5 IN REGISTER 53269 TO 0
or
CHANGE BIT 3 IN REGISTER 53269 TO 1

Unfortunately, there is no such command. We can, however, simulate these lines with real BASIC statements.

POKE REGISTER #, PEEK (REGISTER #) OR 2↑BIT
(BIT = 0 TO 7)

Using Form #1 above, we can turn the specified bit to a 1, while leaving all other bits alone.

POKE REGISTER #, PEEK (REGISTER #) AND (255 - 2↑BIT)

Using Form #2 above, we can turn the specified bit to a 0, while leaving all other bits alone.

The subroutines from 130 to 202 each contain a BASIC statement similar to one of the BASIC forms above, depending upon the desired state of the register. Notice that the program lines use "SP" (abbreviation for sprite) instead of "BIT," since they are the same (Sprite 0 = Bit 0, Sprite 1 = Bit 1, etc.)

The opposite of "turning on" a sprite is turning it "off," which makes it vanish from the viewing screen. LIST lines 140-142. Tool 140 will erase a sprite as easy as you turn one on. All that is needed is the sprite to erase (SP=?) and a GOSUB 140 statement. This tool is described in the following tool box.

TOOL 140:::TURN OFF SPRITE

```
140 REM:::TURN OFF SPRITE SP
141 POKE 53269,PEEK(53269) AND (255 - 2↑SP)
142 RETURN
```

What It Does: This tool removes ("turns off") a sprite from the screen.

Example Use: To use this tool, you need to specify the sprite to turn off by setting SP to the correct sprite number (0-7). This should be followed by a GOSUB 140 statement.

Technical Description: This subroutine uses the second form of the statements introduced in the "turn on sprite" tool box. If you are not familiar with the material in that tool box, please review it before proceeding.

This subroutine sets a bit to 0. Doing this to the appropriate register and correct bit will turn the specified sprite off. This is the opposite of the previous subroutine, where the bit was set to 1 to turn on the sprite. The statement to turn the bit off is in this form:

```
POKE REGISTER #,PEEK(REGISTER #) AND (255 - 2↑SP)
```

Recall that SP is the bit/sprite number to be switched off (0-7).

Notice that the register used in this subroutine and in the previous subroutine is the same: 53269. Register 53269 has the special function of turning sprites "on" and "off."

Since Tool 130 turns on a sprite, and tool 140 turns off a sprite, you can create a flashing sprite by alternating the use of these tools with the same sprite. The sprite will appear, disappear, appear, and disappear in rapid succession. This technique could be applied to images of fire, or the flaming exhaust behind a rocket ship. It could also be used for a blinking red stop light or for twinkling stars in a night scene.

List program lines 1130 and 150-152. On line 1130 the GOSUB 150 uses Tool 150 to enlarge the width of the sun sprite. This tool has doubled the size of the sun in the direction of X.

TOOL 150:::X EXPAND SPRITE

```
150 REM:::X EXPAND SPRITE SP
151 POKE 53277,PEEK(53277) OR 2↑SP
152 RETURN
```

What It Does: This tool enlarges a sprite's *width* to twice its size.

Example Use: To use this tool, you must be sure SP is set equal to the sprite number of the sprite to enlarge (SP=?). Follow this with a GOSUB 150 statement.

Technical Description: This subroutine uses register 53277. This register controls the expansion and contraction of a sprite's width. Each sprite can either be normal size, or expanded in the X direction. If the bit in this register is equal to 0, then the corresponding sprite will be normal width. If the bit is equal to 1, then the sprite's width will be expanded.

This tool uses Form #1 to turn bits to ones. This will immediately expand that specified sprite in the X direction. A sprite does not need to be on when you expand it.

Each sprite's size is individually controlled. To expand a sprite, you need to specify the correct sprite, and then call Tool 150. To change a sprite back to its original size, you need to "unexpand" it. This is easily done by setting SP to the correct sprite number, and then inserting a GOSUB 160 statement in the program. Try out the "X Unexpand" feature by changing line 1130 as follows:

```
1130 GOSUB 160 RETURN
```

Run the program with this change. The sprite should appear at half its previous width because it's "unexpanded." It will be tall and narrow.

TOOL 160:::X UNEXPAND SPRITE

```
160 REM:::X UNEXPAND SPRITE SP
161 POKE 53277,PEEK(53277) AND (255 - 2↑SP)
162 RETURN
```

What It Does: This tool changes an expanded or enlarged sprite back

to its original width. This tool will not affect a sprite which was not previously expanded with Tool 150.

Example Use: To use this tool, you need to set SP equal to the sprite number to expand, and then insert a GOSUB 160 statement.

Technical Description: This subroutine is the opposite of the X expand subroutine (Tool 150). It uses Form #2 of the statements introduced in the "TURN ON SPRITE SP" technical description. In register 53277, turning a bit to 0 will restore the sprite specified by SP's current value to its normal width.

Press the **SPACE BAR** to return to regular text. List lines 1140 and 170-172 on your screen. The GOSUB 170 statement in line 1140 expanded the sun sprite vertically (up and down). The sun is twice its normal (designed) height because this tool was used. The "Y Expand" tool duplicates each row of pixels in the sprite, so the sprite becomes twice as tall.

TOOL 170:::Y EXPAND SPRITE

```
170 REM:::Y EXPAND SPRITE SP
171 POKE 53271, PEEK(53271) OR 2↑SP
172 RETURN
```

What It Does: When this tool is used, a sprite's height doubles in size. This is done by duplicating each row of pixels in the sprite, starting with the top row. Nothing occurs when this tool is used on a sprite that is already expanded in the direction of Y.

Example Use: To use this tool, you must set SP equal to the sprite number that is to be made taller. Then, insert a GOSUB 170 in the main routine.

Technical Description: This tool is identical to the "X EXPAND SPRITE" tool, except register 53271 controls height. This subroutine flips the appropriate bit ("SP"), which causes the computer to double the sprite's height by duplicating each of its rows.

Sprites can have a "normal" height as defined by the original data, or an expanded height using Tool 170. You can choose to use both X and Y expand features, just one expand feature, or neither at all.

To return a sprite to its original height, Tool 180 is used. To unexpand the height of the sun, change line 1140 to:

```
1140 GOSUB 180
```

RUN the program.

The sun should now be displayed at its original height as defined in the data statements. Read the technical discussion below for more information on this sprite feature.

TOOL 180:::Y UNEXPAND SPRITE SP

```
180 REM:::Y UNEXPAND SPRITE SP
181 POKE 53271,PEEK(53271) AND (255 - 2↑SP)
182 RETURN
```

What It Does: The use of the “Y Unexpand” tool changes the height of a “Y expanded” sprite back to its original size as defined in the data statements. Nothing happens when this tool is used for a sprite that is already set to its original height.

Example Use: To use this tool, you will need to set SP equal to the appropriate sprite number (0-7), and then type a GOSUB 180 statement into the main routine.

Technical Description: This tool is identical to the “X UNEXPAND SPRITE,” except that register 53271 controls a sprite’s height. This tool flips the appropriate bit (“SP”), which has the computer display the sprite at its original height.

Return to text mode and list lines 1150 and 200-202. Line 1150 calls Tool 200. This tool gives the *foreground* pixels of any shape priority over the sprite specified by SP’s current value. Again, a sprite *always* has priority over background pixels. This means that when the sun is placed at the same location as a shape, the *shape* will appear in front of the sun. In your picture, the seagull was displayed in front of the sun.

Let’s place the sun down by the water to see how the priority works there. Place the sun at X = 232, Y = 166 by typing this line:

```
1170 X=232: Y=166
```

Run the program with this change. The sun should now appear *behind* the land, and *intermixed* with the water. This is because only foreground pixels have priority over the sprite. The water is made up of 70% background pixels, which the sprite has priority over. Return to text mode and move the sprite one more time:

```
1170 X=207: Y=166
```

Run the program and it will look like the sun is setting over the water. The water appears to reflect the image of the sun, as you might see it when watching a sunset. This is a nice effect, which you can include in other pictures.

You can give shapes priority over one, many or no sprites. To give shapes priority over a sprite, you must specify the sprite (SP=?), and then call this tool with a GOSUB 200 statement.

TOOL 200.....SHAPE PRIORITY OVER SPRITE

```
200 REM SHAPE PRIORITY OVER SPRITE SP
201 POKE 53275,PEEK(53275) OR 2↑SP
202 RETURN
```

What It Does: This tool gives priority to the shapes over a specified sprite (SP=?). When a sprite and any shape are placed at the same location, the sprite will not show up wherever the shape's *foreground* pixels fall.

Example Use: To use this tool, you will need to set SP equal to the number of the sprite in question. Then type a GOSUB 200 statement in your main routine.

Technical Description: Register 53275 controls sprite/shape priority. If a bit is set to 1, then the corresponding sprite will appear to move behind shapes drawn on the screen (such as the ship or the land). If the bit is set to 0, then the corresponding sprite will move in front of those shapes. (Again, "shape" refers to *foreground* pixels.)

To have a sprite appear behind shapes, you must use Form #1 of the statements introduced in the "TURN ON SPRITE" technical box.

Press the **SPACE BAR**. Let's give the sun priority over shapes, and then move it back over the land. To do this, type the following:

```
1150 GOSUB 190
1170 X=232: Y=166
```

With this change, **RUN** the program. This time, the sun should appear in *front* of the water and land. The sun has priority over the shapes when Tool 190 is called. If this sprite had a hole in it, then the shapes underneath could be seen through its "window." Let's put a window in the sun by changing its description in the data statements. Press the **SPACE BAR** to return to text mode.

To put a narrow horizontal hole in the sun, you will change some of the data items in lines 2590 through 2630. These lines define the middle area of the sun sprite. List these lines on your screen. Currently, all the data items in these lines should be 255, because all of the corresponding pixels on the sprite grid were shaded. Change these lines to:

```
2590 DATA 255, 0,255
2600 DATA 255, 0,255
2610 DATA 255, 0,255
2620 DATA 255, 0,255
2630 DATA 255, 0,255
```

In each line, the zero represents blank spaces in the sprite. These unused pixels can be thought of as “transparent,” because they will show the shapes underneath the sprite. RUN the program.

You will see both the land and water through this window. Tool 190 can create some exciting visual effects with sprites—especially when the sprite is animated. (You will animate the sun shortly.)

TOOL 190:.....SPRITE PRIORITY OVER SHAPE

```
190 REM:SPRITE SP PRIORITY OVER SHAPE
191 POKE 53275, PEEK(53275) AND (255 - 2↑SP)
192 RETURN
```

What It Does: When a sprite has priority over shapes, it is displayed completely in front of any shape it falls on. This tool is fun to use with sprites having holes in them, because you can see the shapes through the hole.

Example Use: To use this tool, you will need to set SP equal to the appropriate sprite number. Then type a GOSUB 190 statement into your main routine.

Technical Description: This subroutine uses Form #2 to turn bits to 0. Register 53275 determines which sprites will have screen priority over shapes. Those sprites whose bit is set to 0 will have this priority.

To continue, press the **SPACE BAR**. Let's look at the color tool for the sprite. LIST lines 1160 and 210-212. In line 1160, the color variable “C” is set to the sprite color code 7, which is yellow. With the second statement, GOSUB 210, the sun sprite is painted. Using this tool, all the appropriate pixels in a sprite will be changed to C's color. You can *not* paint the pixels in a sprite with different colors. There are sixteen (0-15) sprite colors which were listed earlier in this chapter. To change the color of a sprite, you need to give C a different color code. Try out the color red (2) on the sun by typing:

```
1160 C = 2:GOSUB 210
```

RUN the program again. Experiment trying out other sprite colors by changing the color code in line 1160 and running the program again. Sprite color codes use the same variable (C) as your shapes, so be sure to reset C's value each time you paint a different sprite or shape.

TOOL 210:::SET SPRITE TO COLOR

```
210 REM:::SET SPRITE SP TO COLOR C
211 POKE 53287 + SP,C
212 RETURN
```

What It Does: This tool paints a sprite with the color specified by C's current value.

Example Use: To use this tool, first enter a program line that sets SP equal to the correct sprite number. Then, enter a program line that sets C to the appropriate sprite color code (0-15). Finally, call this subroutine with a GOSUB 210 statement.

Technical Description: This subroutine is different than the previous sprite subroutines because each sprite has its own color register. The number stored in a sprite's color register is determined by C's current value, and represents one of 16 different available sprite colors.

All 8 registers are placed sequentially, so any of them can be found by adding the sprite number (0-7) to the first register (53287).

When you are finished trying out the various colors, change the color back to yellow (C=7 in line 1160). Next, let's place the sun in the sky with Tool 220. List lines 1170-1180. Change the X and Y values on line 1170 to:

```
1170 X = 232: Y = 10
```

Run the program, and the sun should appear back up in the sky. Tool 220 places the sprite's origin at the specified X and Y location. (Remember, a sprite's origin is the top left square on the Sprite Design Grid.) This tool must be used each time you create a new sprite. Without this placement, the sprite will not show up on the screen. However, once you have placed a sprite, you do not need to place it again unless you would like it moved.

A sprite has a special coordinate system that is described in the technical box below. For most purposes, you should always place the sprite's origin within the 0-319 range for X, and 0-199 range for Y.

You may have noticed that portions of your sun sprite remain on the screen when you return to text mode. This is because sprites are unlike any other shapes, and have to each be *erased* separately by a GOSUB 140 statement. (The use of **RUN/-STOP** and **RESTORE** to return to text mode will erase *all* sprites from the high resolution screen.)

Press the **SPACE BAR**. LIST lines 220-229.

TOOL 220:PLACE SPRITE AT X,Y

```
220 REM::PLACE SPRITE SP AT X,Y
221 XX = X + 24:YY = Y + 50:Z% = XX/256
222 V = XX - Z%*256:W = 53248 + SP*2
223 WW = 53264
224 PR = ABS((PEEK(WW) AND 2↑SP)<>0)
225 VV = PEEK(WW) AND (255-2↑SP) OR (2↑SP*Z%)
226 IF PR<>Z% THEN GOSUB 140
227 POKE W,V:POKE WW,VV:GOSUB 130
228 POKE 53249 + SP*2,YY
229 RETURN
```

What It Does: This tool places the sprite's origin at a specified X,Y screen location. The origin is the top left square on the sprite's Sprite Design Grid. After placing the origin at the X,Y location, the computer can place the rest of the sprite in its relative location. This tool *must* be used each time a sprite is defined and turned on, or you will not be able to see it on the screen.

Example Use: You need to first set SP equal to number of the sprite that you wish to place on the screen. This would be followed by a main routine line in the form of:

```
X=#: Y=#: GOSUB 220
```

In this line, # would be replaced by a number representing a screen coordinate location. X should be a number from 0 to 319 and Y should be a number from 0 to 199. The resulting X,Y coordinate is where the sprite's origin will be positioned on the screen.

Technical Description: It was explained in the text how to position sprites on the screen. It was stated that the X position should range from 0 to 319 and Y should range from 0 to 199. The X,Y coordinate position was where the upper left corner of the sprite was placed. In reality, a sprite can be placed in areas beyond the viewing screen. Sprites can move from 0 to 511 horizontally, and 0 to 255 vertically. However, only the area from 24 to 343 in X, and 50 to 229 in Y are visible on the screen. The added space surrounding the viewing screen lets the sprite move smoothly onto one side of the screen and off of the other. This chapter on sprites talks only about the "normal" screen parameters of 320 x 200, as used in the previous chapters. This was done to maintain consistency between all the chapters. In this PLACE SPRITE AT X,Y subroutine, numbers get added to your X,Y coordinates so that the sprite's origin (0,0) is the same as the viewing screen's origin (0,0). Line 211 does this for you:

```
221 XX = X + 24: YY = Y + 50: Z% = XX/256
```

If you wish to use the expanded area for your sprites, you can remove $XX = X + 24$; $YY = Y + 50$; from this line.

Once again, you will use registers to place sprites. One register controls the X coordinate of the sprite's position. Another controls the Y coordinate of the sprite's position. Each sprite has its own set of X position and Y position registers. Changing the values in these registers will change the placement of the corresponding sprite.

Each memory location can contain a number between 0 and 255. Since a register is just a specialized memory location, it has this same restriction. This works out perfectly, since the Y position can range from 0 to 255. The X position, however, presents a problem because it can range from 0 to 511. If that register were a little bigger, it could handle numbers beyond 255. To solve this problem, another register is added. Each one of the 8 sprites will use 1 bit from it. Sprite 0 gets bit 0, Sprite 1 gets bit 1, etc. The computer will then pretend that these bits have been added to the X position registers to make them bigger. With this added bit, the X position register can handle numbers from 0-511. You cannot, however, POKE the whole number 511 into the X position register. You must break it into two pieces. One piece will go into the sprite's X position register, and the other piece will go into the register that is shared with the other sprites. If the position number is less than or equal to 255, then the bit in the shared register should be 0. If the number is greater than 255, then the bit in the shared register should be 1. Turning this bit to a 1 means you can subtract 256 from the actual position and store this result in the X position register.

$$Z\% = XX/256$$

On this line, the "Z% = XX/256" decides whether the bit in the shared register should be a 0 or a 1.

$$222 \quad V = XX - Z\% * 256; \quad W = 53248 + SP*2$$

Line 222 subtracts 256 from the X position if the result of Z% was equal to 1. It also finds the proper register number for the specified sprite's X position.

$$223 \quad WW = 53264$$

In line 223, WW is the memory location of the register which is shared by all the sprites.

$$224 \quad PR = ABS((PEEK(WW) AND 2↑SP) <> 0)$$

$$225 \quad VV = PEEK(WW) AND (255 - 2↑SP) OR (2↑SP*Z)$$

Line 224 looks at the X position of the sprite. If it is on the left of the imaginary boundary at 255, then PR is set to 0. If it is on the right of the boundary, then PR will be set to 1. This is used to see if the boundary is crossed.

A sprite cannot move horizontally as easily as vertically. When you try to move the sprite past X position 255, a strange thing happens. At the instant it crosses this imaginary border, it will momentarily appear somewhere else on the screen. This happens everytime the sprite is moved from one side of this imaginary border (boundary) to the other. The best solution for this problem is to switch off the sprite just before you change the X position registers. This way you will not see the flash.

Line 225 looks at the current contents of the memory location to make sure the other sprites controlled by this register are not disturbed. The bit which controls the current sprite is set to 0 by the "AND (255-21SP)." This technique was explained in the "TURN ON SPRITE SP" and "TURN OFF SPRITE SP" tool boxes. The "OR (21SP*Z)" resembles Form #1, which turns on the bit. The bit is turned on, however, only if Z equals 1 (i.e., the X position is greater than 255). Multiplying 21SP by Z will result in the bit being set or not.

```
226 IF PR<>Z% THEN GOSUB 140
```

In this line, Z% tells us which side of the boundary the sprite is moving into. If Z%=0, then it will be left of the boundary (imaginary border 0-255). If Z%=1, then it will be right of the boundary (256-511). The variable PR keeps track of where the sprite is currently. If the current section is not equal to the section to be moved into, then you want to switch off the sprite for a moment. GOSUB 140 will do this.

```
227 POKE W,V: POKE WW, VV: GOSUB 130
```

Line 227 changes the X position and turns the sprite back on with a GOSUB 130.

```
228 POKE 53249 + SP*2, YY
```

Finally, line 228 changes the Y position value to the new position.

Try out another placement for your sprite. Change line 1170 so that X=319 and Y=10. RUN the program.

This time, the sprite will be placed almost completely off the screen. Only the thin left edge of pixels will still remain. The rest of the sprite will not be visible.

Remember that the origin determines the placement of the sprite. The origin was placed so far right that the rest of the sprite could not be placed on the screen. In

fact, the rest of the sprite was just “forgotten” by the computer because it had no place to put it. Press the **SPACE BAR**.

Try out $X=315$ and $Y=10$ in line 1170. RUN the program. Now you will see more of the sun’s left side. Press the **SPACE BAR** again and try out these values: $X=0$ and $Y=199$ on line 1170. RUN the program. The sprite will be placed off the screen. This time, the Y coordinate caused this. The origin is placed at $Y=199$ (the last pixel row on the screen).

Press the **SPACE BAR**. Place the sprite’s origin at 0,195 ($X=0$ and $Y=195$). RUN the program and you will see the sun peeking over the border in the lower left corner. Press the **SPACE BAR**. Finally, position the sprite at $X=0$ and $Y=0$ and RUN the program. The sun should appear in the upper left-hand corner. The origin is now at 0,0.

You may want to spend some time experimenting with the sprite features introduced so far. Try expanding and unexpanding the sprite’s height and width. Change the sprite’s color. Give screen priority back to the sprite. Place the sprite at different locations, around and off the screen. When you want to continue, make sure the sprite features are set back to:

```
1110 SP = 0: GOSUB 120
1120 GOSUB 130
1130 GOSUB 160
1140 GOSUB 180
1150 GOSUB 190
1160 C = 7: GOSUB 210
1170 X=0: Y=0
1180 GOSUB 220
```

Animating the Sun Sprite

Finally, the section you’ve all been waiting for—sprite animation. You will be moving the sun across the sky, from right to left. First, change lines 1170 and 1180 to:

```
1170 X1=0:Y1=10:X2=319:Y2=10:SD=5
1180 GOSUB 230: GOTO 1180
```

Before running this program, type in the MOVE SPRITE subroutine below. Be sure to type $X1$ and $Y1$ on lines 231 and 234. On lines 233 and 237, the letter I (“eye”) should be typed after the X and Y .

```
230 REM: :MOVE SPRITE FROM X1,Y1 TO X2,Y2
231 DX = X2 - X1:DY = Y2 - Y1
232 L = ABS(DX):IF ABS(DY) > L THEN L = ABS(DY)
233 IF L > 0 THEN XI = DX/L:YI = DY/L
234 X = X1 + .5:Y = Y1 + .5:SD = SD + ABS(SD = 0)
```

```

235 FOR I = 0 TO L STEP SD
236 GOSUB 220
237 X = X + XI*SD:Y = Y + YI*SD
238 NEXT I
239 RETURN

```

Look over your new program lines. When you feel they have been typed correctly, **RUN** the program. You should see the sun travel across the sky in a continuous motion. Watch as it appears on the left, moves across the screen to the right, and then disappears off the right edge. If you wait only a moment, the sun will appear again at the left edge, and go through this movement all over again.

To stop the animation and learn more about moving sprites, press **RUN/STOP** and tap **RESTORE**. Pressing the **SPACE BAR** will *not* return you to text mode at this point.

```

1170 X1=0: Y1=10: X2=319: Y2=10: SD=5
1180 GOSUB 230: GOTO 1180

```

The sun's movement starts with its origin placed at the X1,Y1 position (0,10). The movement continues in a straight path to the X2,Y2 position (319,10). It is the sprite's origin (upper left sprite grid corner) that follows the straight line from X1,Y1 to X2,Y2. The rest of the sprite is moved relative to the origin. A sprite can be moved horizontally, vertically or diagonally.

At the end of line 1170, SD=5 determines the speed at which the sprite will move. The number 5 indicates how many pixels are skipped along the linear path from X1,Y1 to X2,Y2. The higher the number, the fewer times the computer has to draw the sprite, and the faster the sprite can be moved to its destination. Be careful, though. Too high a number makes the movement appear jerky.

Line 1180 calls the subroutine at line 230. Tool 230 moves the *last* sprite pointed to in the program (SP=?) along the path specified by the *last* X1,Y1 and X2,Y2 values listed in the program.

The second statement on line 1180 (GOTO 1180) moves the sprite across the screen over and over again. When the computer comes to line 1180, it goes to Tool 230 and moves the sprite across the screen. Returning to the main routine, it is sent back to the beginning of line 1180. Doing this, it is sent back to Tool 230, and again moves the sprite along the last X1,Y1-X2,Y2 path listed in the program. This loop is "endless"; that is, it will continue until you stop it outside of the program by pressing **RUN/STOP** and tapping **RESTORE**. Pressing the **SPACE BAR** will *not* stop the program, as the computer never gets the opportunity to read any program lines past 1180.

TOOL 230:.....MOVE SPRITE FROM X1,Y1 TO X2,Y2

```
230 REM: :MOVE SPRITE FROM X1,Y1 TO X2,Y2
231 DX = X2 - X1:DY = Y2 - Y1
232 L = ABS(DX):IF ABS(DY) > L THEN L = ABS(DY)
233 IF L > 0 THEN XI = DX/L:YI = DY/L
234 X = X1 + .5:Y = Y1 + .5:SD = SD + ABS(SD = 0)
235 FOR I = 0 TO L STEP SD
236 GOSUB 220
237 X = X + XI*SD:Y = Y + YI*SD
238 NEXT I
239 RETURN
```

What It Does: This tool will move a sprite along a straight path, starting at X1,Y1 and ending at X2,Y2. The path can be diagonal, vertical or horizontal. It is the sprite's *origin* that follows the path. The speed at which the sprite travels is controlled by setting the variable SD. We recommend that the speed be kept within 1-5.

Example Use:

- (1) Set SP equal to the sprite number of the sprite to move (SP=?).
- (2) Provide the starting (X1=? : Y1=?) and ending (X2=?, Y2=?) locations for the path of movement to be made by the sprite's origin. The sprite will move in a straight line, from X1,Y1 to X2,Y2. These coordinate values should be within the X and Y coordinate ranges.
- (3) Enter a number for the speed (SD=?). The higher the number, the faster the speed.
- (4) Follow all the above with a GOSUB 230.

Technical Description: This subroutine should look familiar to you. It is very similar to the "PLOT A LINE" subroutine. Instead of plotting a line from X1,Y1 to X2,Y2, however, you want to move a sprite from X1,Y1 to X2,Y2. To do this, you use the same variables but send the computer to the PLACE A SPRITE subroutine instead of the PLOT A POINT subroutine. For the sun sprite, this works fine. The sprite moves slowly and smoothly across the sky. This rate of motion would not be satisfactory, however, if the sprite were a spaceship or a race car. To speed things up, we added a new variable: SP (*speed*). A speed of 1 is the "normal" speed. If the speed is between 0 and 1, then the sprite will go slower. If the speed is greater than 1, then the sprite will go faster. What is actually happening is the sprite is skipping over several pixels as the speed increases. If the speed gets too fast, the sprite will move in a jerking motion. At moderate speeds, however, this is not noticeable.

In line 234, `SD = SD + ABS(SD = 0)` makes sure that the value of `SD` is *not* zero. A value of zero would cause problems when using this subroutine. If `SD` does equal zero, this statement will set it equal to 1.

```
237 X = X + XI * SD : Y = Y + YI * SD
```

This statement is identical to the corresponding statement in the line drawing routine, except that the `X` increment (`XI`) and the `Y` increment (`YI`) are multiplied by the speed. This will make the sprite skip over some of the steps as it travels, thus speeding things up.

```
235 FOR I = 0 TO L STEP SD
```

In this statement, “STEP `SD`” is a new addition. It causes the loop index `I` to be increased by the value of `SD` each time it completes a loop, instead of the usual increment of 1. This accounts for the change in the `X` and `Y` increments. This change allows the sprite to skip over some steps, reducing the number of repetitions necessary to complete the journey.

Change line 1170 as follows, and run the program again:

```
1170 X1=0: Y1=0: X2=319: Y2=199: SD=10
```

The sun will move diagonally across and down the screen. The movement, however, will be jagged. This is because the sun’s origin is moving along the straightest path from 0,0 to 319,199. This path is not truly straight, because there is no straight path from 0,0 to 319,199. When you want to move a sprite, consider where a plotted line between your `X1,Y1` and `X2,Y2` points would appear. This plotted line is the exact path your sprite’s origin will follow.

Stop the program by pressing **RUN/STOP** and tapping **RESTORE**. Change line 1170 to:

```
1170 X1=160: Y1=0: X2=160: Y2=199: SD=1
```

RUN the program. The sun should move from the top center screen area *off* the bottom center screen area. Through the sun’s hole, you will see the ship’s lines. Press **RUN/STOP** and **RESTORE**. This time, change the priority of the sprite. Change line 1150 to:

```
1150 GOSUB 200
```

Also, expand the sprite’s height by changing line 1140:

```
1140 GOSUB 170
```


RUN the program. The sun's shape will be tall and thin. The sun will move from the top center of the screen, straight down. The speed this time will be much slower since SD=1 instead of 5. The sun will move behind other shapes because the shapes have been given priority. Use the **RUN/STOP** and **RESTORE** keys to stop the program. LIST lines 1100-1190. Insert a few variations of your own in the different program lines and then RUN the program again.

When you're finished, change lines 1110-1180 back to:

```
1100 REM::::SUN SPRITE
1110 SP = 0: GOSUB 120
1120 GOSUB 130
1130 GOSUB 150
1140 GOSUB 170
1150 GOSUB 200
1160 C = 7: GOSUB 210
1170 X1=0: Y1=10: X2=319: Y2=10: SD=5
1180 GOSUB 230: GOTO 1180
```

Also, change the data so that the sun is a solid shape. You can fill in the hole with these lines:

```
2590 255, 255, 255
2600 255, 255, 255
2610 255, 255, 255
2620 255, 255, 255
2630 255, 255, 255
```

After you have typed in these changes, RUN the program so that the correct picture is displayed on the high resolution screen. Press **RUN/STOP** and **RESTORE**. SAVE the *program* under "CHAPTER 6". Do not continue until the program is safely stored on your disk/tape.

This chapter's picture should *not* be saved. Your SAVE PICTURE routine is not designed to save sprites, which involves different steps than normal shapes.

To display this picture again later, LOAD Chapter 5's *picture*, and then load and run this chapter's *program* (this only takes a moment). Again, because this final program has an "endless" loop, you must press **RUN/STOP** and tap **RESTORE** to stop it.

Design Ideas

This section will demonstrate how sprites and sprite features can be used to your artistic advantage. With practice and imagination, there is no limit to what you can do with sprite animation. The ideas presented here just begin to touch the surface of what is possible with moving sprites. Hopefully, with more time, you can

explore these ideas in your own designs.

To start, you should have already saved this chapter's program. If you have not done this yet, do so now.

In our first example, you will make the sun appear three-dimensional with the use of a shadow. A shadow can be created by overlapping two identically shaped sprites. To do this, the same data statements are used to duplicate the sprite's shape. Then, the two sprites are layered, one on top of the other. The underneath sprite is placed slightly to the right of the top sprite, and is painted a darker color. This gives the appearance of a sprite and its shadow.

To shadow your sun, change the following program lines:

```
1160 C=10: GOSUB 210
1170 X=232: Y=10
1180 GOSUB 220
```

This defines the regular sun sprite. Next, type in the lines below to place the second sprite, the shadow sprite, behind the sun. Note the word "RESTORE" in line 1500. This command is new to you and will be explained later. After typing these lines, RUN the program.

```
1500 RESTORE
1510 SP = 1: GOSUB 120
1520 GOSUB 130
1530 GOSUB 150
1540 GOSUB 170
1550 GOSUB 200
1560 C = 2: GOSUB 210
1570 X = 235: Y = 10
1580 GOSUB 220
```

Your screen will first display a light red sun. Next, the second, darker sun is displayed behind the red one. The priority of these two sprites was determined by their sprite numbers. Since the red sprite has a number of 0 (SP=0), it has priority over any other sprite.

The top sun is light red, as given in line 1160 where C=10. The underneath sun, the shadow, is a darker red, as given on line 1560 where C=2. In this example, the color red is the base color. A base color is the pure, unaltered color. The top sun sprite is a tint, a light red, of the base color. A tint, as you recall, is a lighter, whiter version of the base. Press the **SPACE BAR** to stop the program.

Change lines 1160 and 1560 to use other color codes, and then run the program again. For example, making C=7 in line 1160, and C=4 in line 1560, produces a yellow top sun and a blurry purple shadow. Colors like yellow and purple, orange and blue, and red and green are pairs of opposite colors. In each of these pairs, one color is from the cool color family and the other color is from the warm color family. Press the **SPACE BAR** to stop the program.

Another color combination to try is making C=7 in line 1160, and C=8 in line 1560. When you run this program, the main (front) sun will be yellow, with an orange shadow sun behind it. Colors like yellow and orange are a similar kind of color. Both are in the warm color family. Colors which are similar are called *analogous*. Analogous colors are closely related and have the same base color. Yellow and orange both have a base color of yellow, because there is some yellow in the color orange. Another set of analogous colors would be green and blue, which again are very similar in appearance. Green and blue are both in the cool color family, and are both of the same base color—blue. Press the **SPACE BAR** to get back into text mode.

LIST lines 1110-1510. Both sprites have been labeled with different sprite numbers. On line 1110, the top sun has been given number 0 (SP=0). On line 1510, the underneath sprite has been given number 1 (SP=1).

Line 1500 contains the BASIC command “RESTORE.” Before reaching this command, the computer has READ the sun’s data statements and has “checked off” each data item—so they are not re-read during the program’s execution. The RESTORE command, however, instructs the computer to re-read those data statements to define the second sprite (Sprite 1). The computer essentially forgets that the data was already read during the program’s execution. This way, the same data can be used again to make the second sprite. You will need to remember that RESTORE tells the computer to forget that it has read *any* data statements. Thus, it will always READ the *very first* data statement in your program after coming across a RESTORE command. RESTORE does *not* have the computer forget reading the last *set* of data statements.

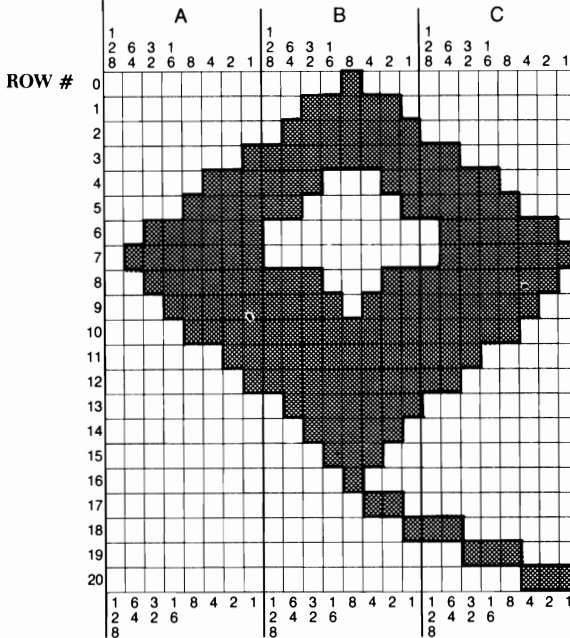
On line 1510, the second sprite has its shape defined with the GOSUB 120 statement. Tool 120 reads and stores the same data for Sprite 1 as was used for Sprite 0.

For variety, let’s enter a new design to move. Exciting animations are a result of thoughtful variations in the sprite designs.

To have a new sprite shape, you will need a new sprite data block. For this example, you will be typing in a data block to define the kite shown in the grid below:

SPRITE DESIGN GRID

(TOP)



DATA STATEMENTS

BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
1210	DATA	0	8	0
1220	DATA	0	62	0
1230	DATA	0	127	0
1240	DATA	1	255	192
1250	DATA	7	227	216
1260	DATA	15	193	248
1270	DATA	63	0	126
1280	DATA	127	0	127
1290	DATA	63	227	254
1300	DATA	31	247	252
1310	DATA	15	255	248
1320	DATA	3	255	224
1330	DATA	1	255	192
1340	DATA	0	127	0
1350	DATA	0	62	0
1360	DATA	0	28	0
1370	DATA	0	8	0
1380	DATA	0	6	0
1390	DATA	0	1	192
1400	DATA	0	0	56
1410	DATA	0	0	7

At the center of this kite, a transparent “window” has been placed. Through it, you will be able to see everything underneath the kite. To make this kite, type these data statements into your program:

```

1200 REM:::::KITE SPRITE DATA
1210 DATA 0, 8, 0
1220 DATA 0, 62, 0
1230 DATA 0,127, 0
1240 DATA 1,255,192
1250 DATA 7,227,216
1260 DATA 15,193,248
1270 DATA 63, 0,126
1280 DATA 127, 0,127
1290 DATA 63,227,254
1300 DATA 31,247,252
1310 DATA 15,255,248
1320 DATA 3,255,224
1330 DATA 1,255,192
1340 DATA 0,127, 0
    
```

```

1350 DATA 0, 62, 0
1360 DATA 0, 28, 0
1370 DATA 0, 8, 0
1380 DATA 0, 6, 0
1390 DATA 0, 1, 192
1400 DATA 0, 0, 56
1410 DATA 0, 0, 7

```

Then, list lines 1100-1180. Convert these lines so they can be used for the kite. Look at the program lines below and compare them to your program. In your program, you will need to change lines 1100 and 1160. Also, C should equal 2 in line 1160 now. Finally, delete lines 1170 and 1180.

```

1100 REM:::::KITE SPRITE 0
1110 SP = 0: GOSUB 120
1120 GOSUB 130
1130 GOSUB 150
1140 GOSUB 170
1150 GOSUB 200
1160 C = 2: GOSUB 210

```

After correcting the above section, add the following lines:

```

4000 SP = 0
4010 X1 = 319: Y1 = 40: X2 = 0: Y2 = 0: SD = 4
4020 GOSUB 230
4030 GOTO 4010

```

Lines 4000-4030 give the necessary values for moving the sprite using Tool 230. These lines were inserted *after* the sun's data so that the kite will move after the sun is placed in the sky. If these statements came before line 1100, then the kite would fly in a sunless sky.

Delete line 1500 (type 1500 and press **RETURN**).

Now list lines 1500-1580. These lines are for the sun sprite. Make the following changes to them:

```

1500 REM:::::SUN SPRITE
1510 SP = 1: GOSUB 120
1520 GOSUB 130
1530 GOSUB 150
1540 GOSUB 170
1550 GOSUB 200
1560 C = 7: GOSUB 210
1570 X = 232: Y = 10
1580 GOSUB 220

```

Delete lines 1590, 1600, 1610, 1620, 1630 and 1640.

Make sure that the color code in line 1560 is changed to C=7. Note that lines 1570 and 1580 are different from before. This time the sun will stay in one place. It will not be moved in the sky. Tool 220 is the placement routine. Since you are using two sets of data for the sprites, there was no need for the "RESTORE" on line 1500. RUN this program.

A red kite with a triangular window in it is flying across the sky. The fluttering of the kite is a result of the kite's diagonal path of movement. It starts from a lower place on the right (Y=40), and moves up to a higher place on the left (Y=0). The speed is controlled with SD=4.

Each sprite has its own data. The data defines the shape of the sprite. The hole within the kite is made up of "transparent" or "blank" pixels. A hole like this is referred to as the *negative space* in the design. The negative space in your kite is diamond shaped, which is far more interesting than a simple square hole. When designing your sprites, take as much time in the planning of negative space as with positive. Make your negative spaces as interesting as possible. For example, look at the variation of negative spaces in this illustration of a butterfly sprite:

SPRITE DESIGN GRID		DATA STATEMENTS							
(TOP)									
		A	B	C					
		1 2 6 3 1	1 2 6 3 1	1 2 6 3 1	BASIC	DATA	SUM	SUM	SUM
		8 4 2 6 8 4 2 1	8 4 2 6 8 4 2 1	8 4 2 6 8 4 2 1	LINE #		OF	OF	OF
ROW #							A	B	C
0					2510	DATA	192	0	3
1					2520	DATA	248	0	31
2					2530	DATA	184	0	29
3					2540	DATA	158	0	121
4					2550	DATA	143	153	241
5					2560	DATA	255	219	255
6					2570	DATA	255	255	255
7					2580	DATA	225	231	135
8					2590	DATA	255	231	255
9					2600	DATA	248	231	31
10					2610	DATA	255	231	255
11					2620	DATA	126	102	126
12					2630	DATA	7	255	224
13					2640	DATA	31	231	248
14					2650	DATA	63	126	252
15					2660	DATA	127	231	254
16					2670	DATA	127	165	254
17					2680	DATA	127	231	254
18					2690	DATA	63	195	252
19					2700	DATA	15	129	240
20					2710	DATA	7	0	224
		1 6 3 1	1 6 3 1	1 6 3 1					
		2 4 2 6	2 4 2 6	2 4 2 6					
		8	8	8					

In this butterfly, the size and shape of the negative spaces are varied. This variation creates an interesting pattern within the wings.

Press the **RUN/STOP** and **RESTORE** keys. In the next example, the kite will be layered on top of the sun. New placement values and a **GOSUB 220** statement will do this for you. List line 4010-4030. Change lines 4010 and 4020 so they read:

```
4010 X = 232: Y = 15
4020 GOSUB 220
```

Delete line 4030 and **RUN** the program.

This time, the kite is placed right on top of the sun. As you can see, sprite colors do not mix with any other color blocks. You could layer several colored sprites together without any problems. Layering would be effective for making a stop light. For a stop light, you would place a round red sprite on top of a square black sprite. Then, using Tools 130 and 140, the red light could be turned on and off. Alternating the use of these tools would create the flashing effect. Let's try out this flashing technique by changing a few lines in your program. Return to text mode. Type in these lines:

```
4030 GOSUB 140: GOSUB 130
4040 GOTO 4030
```

RUN the program.

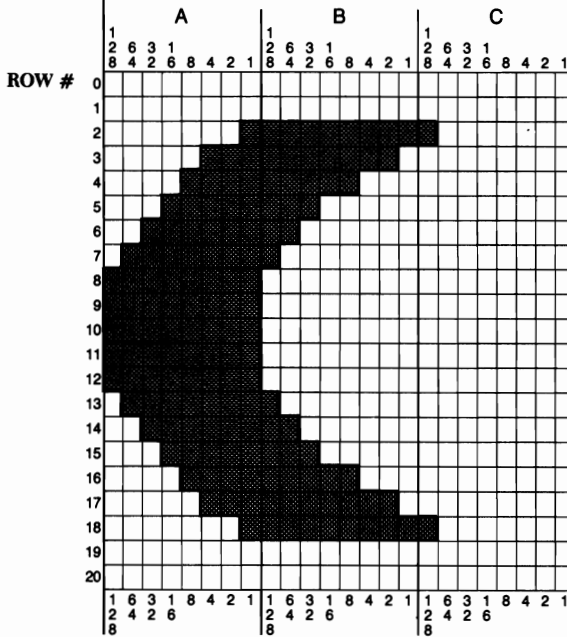
When you **RUN** it with these added lines, the red kite will be "flashing." This flashing effect is caused by turning the sprite "off" and "on." This is done with alternating **GOSUB 140**'s and **GOSUB 130**'s. The kite will continue to disappear, appear, disappear, appear, because the program is in an endless loop.

Press **RUN/STOP** and tap **RESTORE** to stop the program. Try out some other color codes on lines 1160 and 1560 for these sprites. Then **RUN** the program again.

When ready, change some of the sun's data to make a different shape. With a few alterations, the data would define a half moon instead of a sun.

SPRITE DESIGN GRID

(TOP)



DATA STATEMENTS

BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
2510	DATA	0	0	0
2520	DATA	0	0	0
2530	DATA	1	255	128
2540	DATA	7	254	0
2550	DATA	15	248	0
2560	DATA	31	224	0
2570	DATA	63	192	0
2580	DATA	127	128	0
2590	DATA	255	0	0
2600	DATA	255	0	0
2610	DATA	255	0	0
2620	DATA	255	0	0
2630	DATA	255	0	0
2640	DATA	127	128	0
2650	DATA	63	192	0
2660	DATA	31	224	0
2670	DATA	15	248	0
2680	DATA	7	254	128
2690	DATA	1	255	0
2700	DATA	0	0	0
2710	DATA	0	0	0

Below is an example of the data block that would define the half moon. Change your sun's data block to contain these data items:

```

2500 REM:::MOON DATA
2510 DATA 0, 0, 0
2520 DATA 0, 0, 0
2530 DATA 1,255,128
2540 DATA 7,254, 0
2550 DATA 15,248, 0
2560 DATA 31,224, 0
2570 DATA 63,192, 0
2580 DATA 127,128, 0
2590 DATA 255, 0, 0
3000 DATA 255, 0, 0
3010 DATA 255, 0, 0
3020 DATA 255, 0, 0
3030 DATA 255, 0, 0
3040 DATA 127,128, 0
3050 DATA 63,192, 0
    
```



```
3060 DATA 31, 224, 0
3070 DATA 15, 248, 0
3080 DATA 7, 254, 0
3090 DATA 1, 255, 128
4000 DATA 0, 0, 0
4010 DATA 0, 0, 0
```

RUN this program to see the half moon.

Experiment by trying different shapes for your sprites. Since a sprite can be only one color, you will find some shapes more successful than others. Also, try layering your sprites. Layering different colored butterflies together would create one multi-colored butterfly. To do this, you would first display a solid colored butterfly on the screen. Then, on top of that sprite, place a similar butterfly that is colored differently. The butterfly on top should have some blank, negative spaces defined in it so that the underneath colors could show through. There's no end to all the possibilities that you can try with sprite features.

At this point, if you want to save the "design ideas" as a program, then do so. In the next section, all the sprite features and the procedures for moving a sprite are reviewed. Following this summary are two final exercises. In the exercises, you will make a cloud sprite on Chapter 6's picture.

Summary

We hope you've enjoyed discovering the colorful and exciting world of Commodore 64 graphics. You have gone from plotting a single point, to creating a reasonably complex piece of graphic art work. Although not all the bases were covered in this book, you should be much more confident the next time you set out to program a picture of your own. Remember that your tool kit can and should be applied to all of your pictures. In addition, try applying the color and shading tips we've presented throughout the chapters. You might be surprised at the results.

The appendices at the end of this book provide a copy of the color chart, the sprite grid, as well as any other grid used in presenting this book's material. You may want to make copies of these. Also included in the appendices is a programmer's "trouble-shooting" guide to program bugs, and several additional tools that will be of use in any picture-drawing program. Be sure to browse through these appendices when you get the chance.

This section will briefly review the various steps involved in making a sprite. These steps are followed with a short description about the various sprite features, and then this book's final exercises.

The procedure used for making a sprite is:

- (1) Design the sprite on the "Sprite Design Grid" by lightly sketching in an

outline of its shape. Then, shade the squares inside this outline. In designing a sprite, consider having "holes" or blank spaces inside the sprite shape.

- (2) Add up the data sums (A,B,C) for the three areas in each row of the design. Write down these sums on the lines alongside the "Sprite Design Grid." A blank section of 8 squares has a sum of 0. An entirely shaded section of 8 squares has a sum of 255.
- (3) Enter 21 data statements in the main routine that correspond to the data items listed on the Sprite Design Grid.
- (4) In the main routine of your BASIC program, type in a value (a number from 0 through 7) for this sprite. For example, SP=0 means you wish to refer to Sprite 0. The priority of the sprites is determined by the sprite numbers. Sprite 0 has priority over all other sprites. Sprite 7 has no priority.
- (5) Enter a GOSUB 120 in the main routine to define the sprite's shape in memory. Tool 120 reads and stores the data in an array (list). This step, and step (4) above, could be typed on one line in the form of:

```
1110 SP=0: GOSUB 120
```

- (6) In the main routine, type in the selected sprite features which will be applied to this sprite. These sprite features are written as GOSUBs. Each GOSUB calls a particular tool to manipulate a sprite. Below are several example program lines for specifying a sprite's features. A complete review of all sprite features immediately follows.

```
1120 GOSUB 130                ("turns on" the sprite)
```

```
1130 GOSUB 150                (X expand sprite)
```

```
1140 GOSUB 170                (Y expand sprite)
```

```
1150 GOSUB 200                (shape priority over a sprite)
```

```
1160 C=7: GOSUB 210           (set color of sprite)
```

```
1170 X=232: Y=10              (X,Y placement location)
```

```
1180 GOSUB 220                (place sprite at X,Y)
```

Lines 1170 and 1180 could easily be changed to move the sprite. To do this, replace lines 1170 and 1180 with these lines:

```
1170 X1=0: Y1=10: X2=319: Y2=10: SD=5  (set values for  
path of movement and speed)
```

1180 GOSUB 230: GOTO 1180 (*endless loop that moves the sprite continuously*)

Summary of Sprite Features

An overview of sprite features is given below. For a more thorough discussion of each feature, refer to the beginning of this chapter.

SP=0: GOSUB 120

This line refers to Sprite 0. It also uses Tool 120 to define the sprite's shape in memory. Note that this format should be used to define each sprite, with SP set to a different number (0-7) each time. The sprite number determines sprite priority over sprites. 0 is highest priority; 7 is lowest priority.

GOSUB 130

This line "turns on" a sprite. If the sprite has not been placed on the screen, you won't be able to see it after turning it on.

GOSUB 140

This tool "turns off" a sprite so that it disappears from the screen.

GOSUB 150

The sprite's width is doubled in size with Tool 150.

GOSUB 160

This tool is used with a sprite whose width has been doubled with tool 150. This tool will return the sprite to its normal width.

GOSUB 170

The sprite's height is doubled in size with Tool 170.

GOSUB 180

This tool is used with a sprite whose height has been doubled with tool 170. This tool will return the sprite to its normal height.

GOSUB 190

This tool gives a sprite priority over all foreground pixels in any shape. The sprite will be displayed in front of any shape which is placed at the same location.

GOSUB 200

Tool 200 allows any shape in the picture to have priority over the sprite. When the sprite and a shape appear in the same screen location, the shape will be in front of the sprite.

C=7: GOSUB 210

Tool 210 sets the sprite to the color determined by C's value. There are 16 different sprite colors (0-15). Each sprite can be only one color.

X=232: Y=10: GOSUB 220

In this line, the X and Y values for positioning the sprite on the screen are given. This X,Y pixel point determines the screen location for the sprite's origin. The rest of the sprite is then positioned relative to the origin. Tool 220 actually places the sprite on the screen at the given X,Y values.

X1=0: Y1=10: X2=319: Y2=10: SD=5

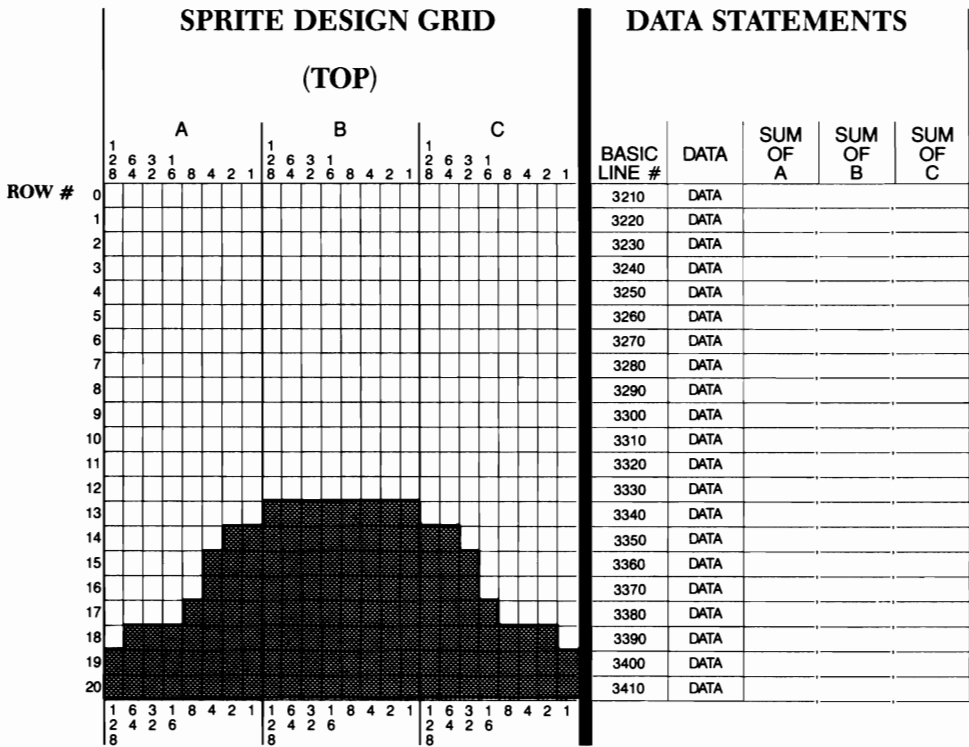
To move a sprite in a straight line, the starting and ending points for this line are given. X1,Y1 are for the coordinate values for the starting point. X2,Y2 are the coordinate values for the ending point. These values determine the path of movement. A sprite can move in any straight direction. It is, however, the sprite's *origin* that will move along the path from X1,Y1 to X2,Y2. The rest of the sprite moves along with the origin. SD indicates the rate of speed for moving the sprite. A higher number for SD means the sprite will travel faster.

GOSUB 230

Tool 230 uses the given X1,Y1 and X2,Y2 coordinate values to move the sprite in a straight line.

Exercise #1

To start, load Chapter 5's picture, and then load this chapter's program and run it. In this exercise, you will be creating a cloud sprite for your picture. Below is an illustration of the cloud on the Sprite Design Grid. The data sums—A, B, C—for each row in the sprite have been omitted.



Using a pencil, figure out and pencil-in the data items for each row in the cloud. Then, starting on line 3200 in Chapter 6's program, type in the data statements necessary for this sprite. Begin like this:

```

3200 REM:::CLOUD SPRITE DATA
3210 DATA 0, 0, 0
3220 DATA 0, 0, 0

```

Solution #1

The data statements for the cloud sprite are as follows:

```

3200 DATA 0, 0, 0
3210 DATA 0, 0, 0
3220 DATA 0, 0, 0
3230 DATA 0, 0, 0
3240 DATA 0, 0, 0
3250 DATA 0, 0, 0
3260 DATA 0, 0, 0
3270 DATA 0, 0, 0
3280 DATA 0, 0, 0
3290 DATA 0, 0, 0
3300 DATA 0, 0, 0
3310 DATA 0, 0, 0
3320 DATA 0, 0, 0
3330 DATA 0, 0, 0
3340 DATA 0, 255, 0
3350 DATA 3, 255, 192
3360 DATA 7, 255, 224
3370 DATA 7, 255, 224
3380 DATA 15, 255, 240
3390 DATA 126, 255, 254
3400 DATA 255, 255, 255
3410 DATA 255, 255, 255

```

Exercise #2

Beginning on line 3100, type in all the program lines necessary to define the sprite and give it the following features:

- (1) priority over the sun sprite
- (2) expanded width
- (3) expanded height
- (4) no priority over shapes
- (5) a color of white (color code = 1)
- (6) origin placement of 215,4

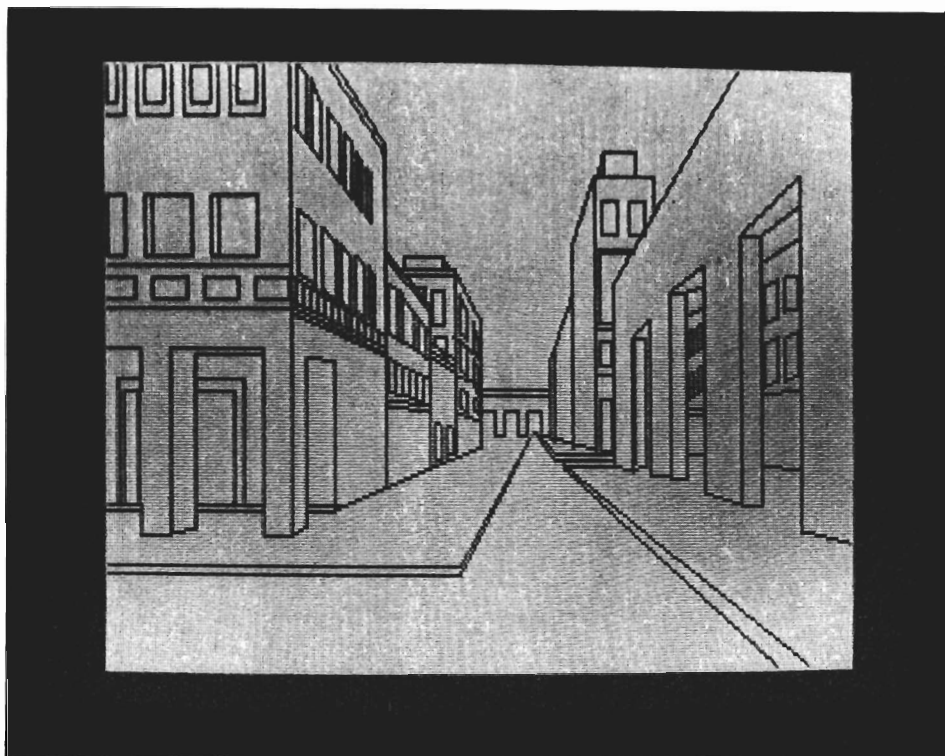
In addition, change the sun features so that it remains stationery (line 1180 should have a GOSUB 220). Put the sun's origin at 232,10.

Solution #2

```
1110 SP=1: GOSUB 120
1170 X=232: Y=10
1180 GOSUB 220
3100 REM::::::::::CLOUD SPRITE
3110 SP=0: GOSUB 120
3120 GOSUB 130
3130 GOSUB 150
3140 GOSUB 170
3150 GOSUB 200
3160 C=1: GOSUB 210
3170 X=215: Y=4
3180 GOSUB 220
```

POSTSCRIPT

Now that you have acquired the tools needed to construct graphics on your Commodore 64, you can begin to discover the excitement surrounding computer graphics. To whet your appetite, this postscript shows three different scenes our artist has created using this book's tool kit. The first is a city scene that effectively uses the PLOT A LINE tool. The second illustration shows an abstract design created with the DRAW A SHAPE tool. The final piece shows how some of the different sprite design tools can be used to create a jungle scene.



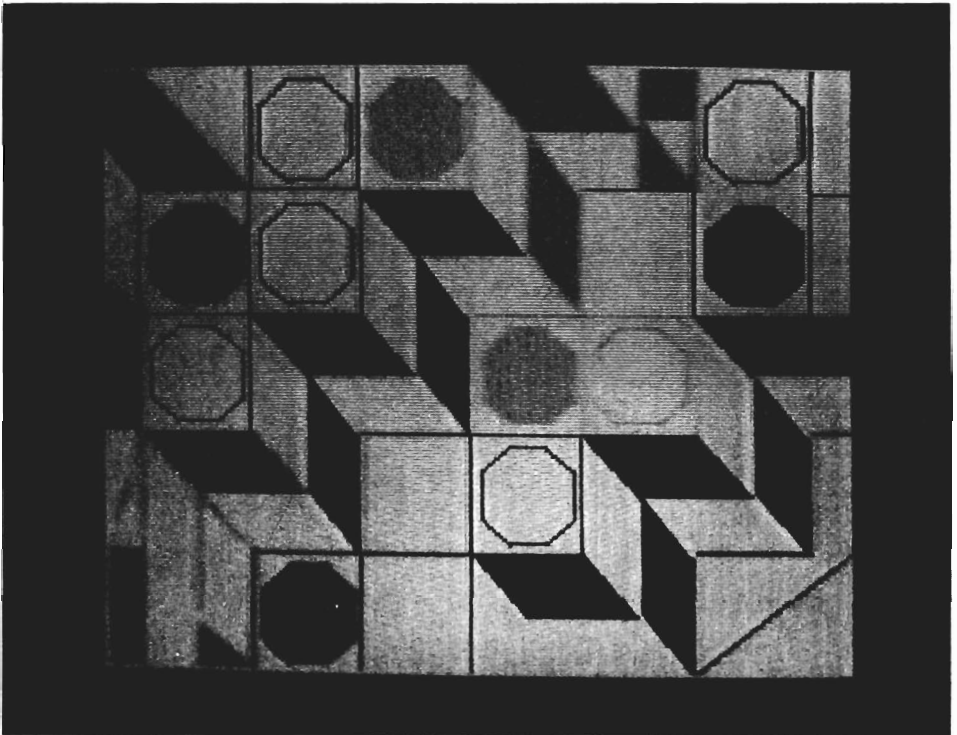
“City Scene”

In this line drawing, the buildings appear to recede into space. As the buildings and windows recede, they become smaller. This sense of space was created with the use of “perspective.” Perspective shows lines that seem to disappear into the distance. The point where all the lines seem to meet and disappear is called the “vanishing point.” Notice how certain lines of the buildings, windows, and the center street angle off towards the vanishing point. The converging lines direct our eyes to this point.

All the lines in this picture were drawn using Tool 80 (PLOT A LINE). First, the artist drew the picture on an X,Y PIXEL POINTS grid. All the points (X,Y coordinates) and the sequence for connecting them were listed on scratch paper. This list was divided into two parts: one for all the points and lines on the left side, and one for all the points and lines on the right side.

Each list was entered as a separate program. The programs were labeled with REM statements to identify the specific buildings. Using two programs and numerous REM statements helped in detecting any program errors.

Finally, both programs were run to display the whole picture. The image was then saved as one picture, using Tool 100. In creating your own designs, you can use perspective to suggest train tracks, roads, or a row of buildings.

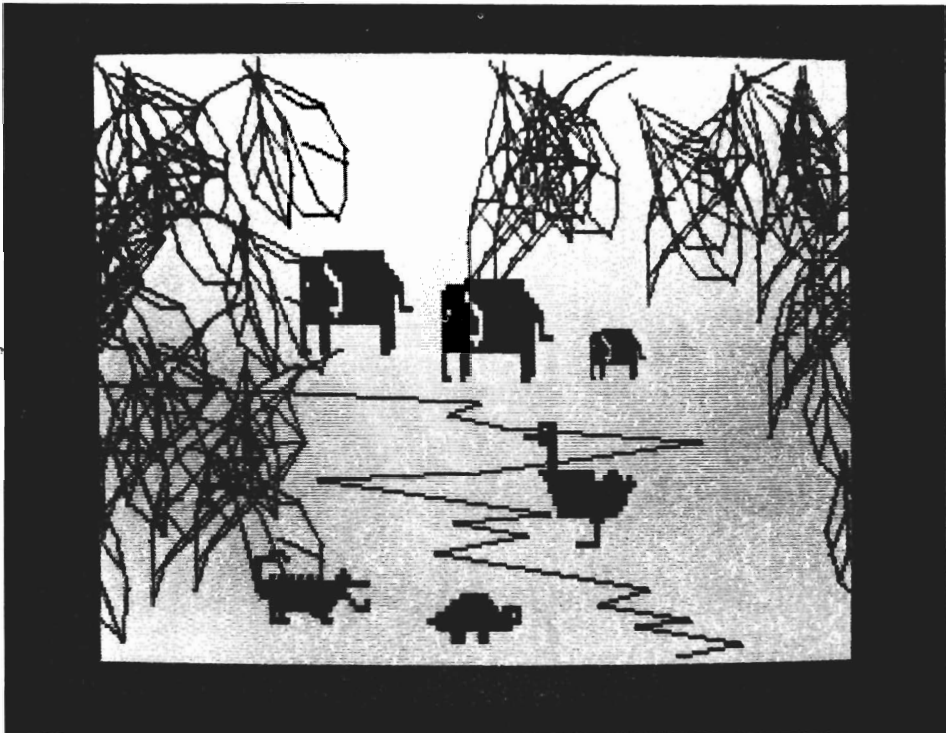


“Flip Box Picture”

Geometric shapes and lines were repeated throughout this “non-figurative” composition. A non-figurative design uses geometric shapes, like squares and octagons, rather than life-like figures (trees, people, birds). An assortment of contrasting tones and various placements of shapes was used for variety in this design. A compositional balance was created between the repetition of shapes and

the variety of tones used. Notice the way in which tones were applied to the shapes. Sometimes, the shapes are completely filled in. At other times only the outlines have been drawn. This variety in shading and the strong tonal combinations make the overall design more interesting.

In creating the design, the artist first drew and shaded the picture on the X,Y PIXEL POINTS grid. The shades were carefully chosen, and placed so that different tones did not overlap in the same color block. A list of points, lines, and offset values was made for each shape which was repeated in the design. These repeated shapes include the top of the box, the side of the box, and the octagon shapes. A program was then written and entered into the computer which would display these shapes. This program used the DRAW A SHAPE tool and the PAINT A SHAPE tool. Another program was entered to draw shapes that would not be repeated, such as the irregular shapes along the sides of the picture. This second program also drew in the many lines between the shapes. Tools 80, 90 and 110 were used in this program. Afterwards, the completed picture was saved using Tool 100.



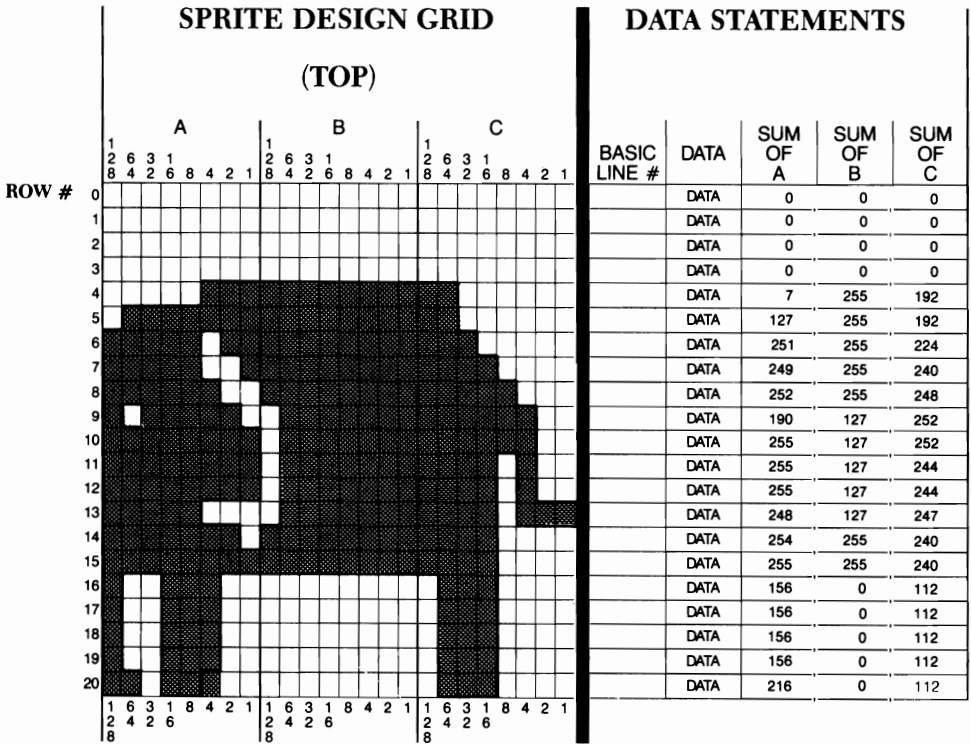
“Jungle Scene”

Several leaf motifs were repeated throughout this picture to create a jungletmosphere. Some areas of the jungle are darker than others. The darker areas are

where the leaves are placed close together so they overlap. As the number of leaves increases within an area, the tone becomes increasingly dark and dense. In areas where the leaves are placed further apart, the tone is much lighter. The light background shows through where there is less intersection of leaves.

Three different leaf motifs are used. Each one has a character all its own. Take a close look at the picture and examine each leaf separately. You will notice that each leaf is a unique shape and size.

Within the jungle surroundings there are sprite animals: three elephants, an ostrich, a crocodile, and a turtle. Note the use of negative space in the elephants:



Here, the negative space outlines the ear. The design and data items for the three other animals are:

SPRITE DESIGN GRID (TOP)

ROW #	A								B								C							
	1	2	6	3	1	8	4	2	1	2	6	3	1	8	4	2	1	2	6	3	1	8	4	2
0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								

DATA STATEMENTS

BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	28	0	0
	DATA	242	0	0
	DATA	128	0	0
	DATA	128	0	32
	DATA	149	84	113
	DATA	255	255	255
	DATA	255	255	224
	DATA	127	255	240
	DATA	31	255	137
	DATA	63	255	15
	DATA	40	12	0
	DATA	60	6	0

SPRITE DESIGN GRID (TOP)

ROW #	A								B								C							
	1	2	6	3	1	8	4	2	1	2	6	3	1	8	4	2	1	2	6	3	1	8	4	2
0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								

DATA STATEMENTS

BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	63	0
	DATA	0	255	128
	DATA	1	255	222
	DATA	7	255	250
	DATA	31	255	254
	DATA	7	255	248
	DATA	0	227	0
	DATA	0	227	0

SPRITE DESIGN GRID (TOP)															DATA STATEMENTS									
ROW #	A					B					C					BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C				
	1	2	3	4	1	1	2	3	4	1	1	2	3	4	1									
0																	DATA	30	0	0				
1																	DATA	22	0	0				
2																	DATA	254	0	0				
3																	DATA	30	0	0				
4																	DATA	6	0	0				
5																	DATA	6	0	0				
6																	DATA	6	0	0				
7																	DATA	6	0	0				
8																	DATA	31	254	48				
9																	DATA	15	254	122				
10																	DATA	7	255	255				
11																	DATA	7	255	254				
12																	DATA	3	255	250				
13																	DATA	3	255	250				
14																	DATA	1	255	248				
15																	DATA	1	255	240				
16																	DATA	0	7	0				
17																	DATA	0	1	0				
18																	DATA	0	1	0				
19																	DATA	0	1	0				
20																	DATA	0	62	0				
	1	6	3	1	8	4	2	1	1	6	3	1	8	4	2	1	1	6	3	1	8	4	2	1
	2	4	2	6					2	4	2	6				2	4	2	6					
	8								8							8								

The sprite animals were positioned in the picture for a feeling of three-dimensional space. For example, the elephants look further away from the ostrich. The elephants are placed higher on the screen to give this effect.

To make the jungle, the artist designed three leaf shapes on the X,Y PIXEL POINTS grid. The leaf data was then entered into the program in the form of data statements. This data was placed in the E% and L% lists for use with the DRAW A SHAPE tool. Numerous offset placement values (X0,Y0) were used to position all the leaves.

The sprite animals were individually designed on the Sprite Design Grid (shown previously). Then, the sprite data and sprite features were entered into the computer program. Once the program was run, the picture was saved with Tool 100. Since this SAVE PICTURE routine can not save sprites, another program was made. This program contained only the program lines that dealt with the sprites. To see the picture again, the jungle *picture* was loaded, and then the sprite *program* was loaded and run.

APPENDIX A:

THE PROGRAMMER'S TROUBLE SHOOTING GUIDE

This appendix offers an overview of some of the more common program “bugs” (errors). It is broken down into two sections: *Preventive Measures* and *Common Cures*. The Preventive Measures section gives helpful hints on avoiding program bugs. The Common Cures section is a checklist of possible cures for some common programming errors.

This appendix is by no means a complete and absolute checklist for all possible program problems.

PREVENTIVE MEASURES

All cassette decks should be kept as far as possible from the computer, monitor, and any other metallic object. It is also a good idea to store your cassette tapes and floppy disks at a safe distance (5 feet or more) from the computer.

Type your programs in lower-case. It is much easier to distinguish small oh's (o) from zeroes (0) than it is to distinguish large oh's (O) from zeroes (0). Also, 8's are more clearly distinguished from lower-case b's.

Re-name corrected programs on a disk rather than use the “@0” command. This command has a bug that can sometimes, although not always, be disastrous for your stored programs. We recommend saving corrected programs under new names, like “CHAPTER 6.1”, “CHAPTER 6.2” etc.

Create a complex picture in small steps. Type in a small section of the program (about 2 handwritten pages of text), and then run it. At that time, locate and correct any errors. The time spent locating errors can be greatly reduced by following this procedure.

Label your programs with numerous REM statements. These REMs will help you later when you are looking for bugs in the program.

Save your program frequently. This way, if the unexpected happens (loss of electricity), you will not lose all of your work.

Break up long, complicated programs into 2 or 3 smaller programs. Again, this will help locate errors by isolating the problem in a smaller area of program lines.

COMMON CURES

In moments of deepest depression; when searching for a way out of that dark abyss known as the SYNTAX ERROR hole of horror, recall these words of wisdom: *The Leading Cause of Program Death is a Typing Error*. Fortunately for all of us, programs can be brought back to life.

It is anyone's guess as to where the typing error(s) might be, so careful checking is the only solution. Whenever a program fails, begin by returning to text mode (do *not* use **RUN/STOP RESTORE**—instead, carefully type **GOSUB 30** and press **RETURN**). Look for an error message with a line number. This is the first and

most valuable clue to the problem.

If your program was typed in upper-case, you can easily switch the entire listing to lower-case by holding down a **SHIFT** key and pressing C=. Oh's (O) typed for zeroes (0), el's (l) typed for ones (1), and b's (B) typed for eights (8) are then quickly identified.

Also, check to be sure you have given a value to all necessary variables *before* each GOSUB statement. A complete listing of all variables needed for each tool can be found in Appendix G.

If your program crashes, and your main routine disappears, you probably had a GOSUB 10 statement (instead of, perhaps, a GOSUB 110) in the main routine. This causes the program to erase itself during execution.

Plot a Point Problem? Check the color code (C=?) of the point to plot. Make sure the color code represents two *different* colors. Check that X and Y are within their respective ranges.

Plot a Line Problem? Check the color code (C=?) for the line to plot. Make sure the color code represents two *different* colors. Check that you have an X1, a Y1, an X2 and a Y2. Reversing 1's with 2's and/or X's with Y's is very common. If you have several sets of these variables, backtrack from the GOSUB 80 statement to the *last* X1, Y1, X2 and Y2 values. Compare these to the line you *want* to plot.

Paint a Shape Problem? Check that your shape doesn't need to be divided into sections in order for this tool to paint it properly. Add 1 to your H value and 1 to your W value, and then run the program again.

Draw a Shape Problem? Check each I loop and make sure it is set to process the correct number of times. Check each set of data statements. Make sure each set has an *even* number of data items. Look for periods (".") typed instead of commas (",") between the data items. If you get an OUT OF DATA ERROR *all* data statements in your program are suspect, and should be checked.

Sprite Problem? To see a sprite on the screen you need to assign it a sprite number between 0 and 7 (SP=?), define it in memory (GOSUB 120), turn it on (GOSUB 130), and place it on the screen (X=? : Y=? : GOSUB 220).

APPENDIX B: COMPLETE LISTING OF TOOLS

```
1 GOTO 1000
10 REM::::::::::ZAP!
11 A = 256: B = 2049: C = 1003
12 IF PEEK(B+2) + A * PEEK(B+3) >= C THEN 15
13 B = PEEK (B) + A * PEEK (B+1): ON ABS(B<>0) GOTO 12: END
14 A = 256: B = PEEK(251) + A * PEEK (252)
15 IF PEEK(B+1) = 0 THEN END
16 PRINT CHR$(147) PEEK (B+2) + A * PEEK (B+3): PRINT "GOTO 14"
17 POKE 251, B - INT (B/A) * A: POKE 252,B/A
18 POKE 631,19: POKE 632,13: POKE 633,13: POKE 198,3: END
20 REM::::::::::GRAPHICS
21 POKE 53265,59
22 POKE 53272,29
23 POKE 56576,198
24 RETURN
30 REM::::::::::TEXT
31 POKE 53265,27
32 POKE 53272,21
33 POKE 56576,199
34 RETURN
40 REM::::::::::COLORS
41 FOR I = 17408 TO 18407
42 POKE I,C
43 NEXT I
44 RETURN
50 REM::::::::::PAINT BACKGROUND
51 FOR I = 24576 to 32575
52 POKE I,0
53 NEXT I
54 RETURN
60 REM::::::::::FIND A POINT
61 ROW = INT(Y/8)
62 COL = INT(X/8)
63 LINE = Y AND 7
64 BIT = 7 - (X AND 7)
65 BYTE = 24576 + ROW*320 + COL*8 + LINE
66 CBYTE = 17408 + ROW * 40 + COL
67 RETURN
70 REM::::::::::PLOT A POINT
71 GOSUB 60
72 POKE BYTE,PEEK(BYTE) OR 2 ↑ BIT
```

```

73 POKE CBYTE,C
74 RETURN
80 REM::::::::::PLOT A LINE
81 DX = X2 - X1: DY = Y2 - Y1
82 L = ABS(DX): IF ABS(DY) > L THEN L = ABS(DY)
83 IF L > 0 THEN XI = DX/L: YI = DY/L
84 X = X1 + .5: Y = Y1 + .5
85 FOR I = 0 TO L
86 GOSUB 70 : REM PLOT POINT
87 X = X + XI: Y = Y + YI
88 NEXT I
89 RETURN
90 REM::::::::::PAINT A SHAPE
91 PC=PC+ABS(PC=0): FOR X = X0 TO X0 + W: FL$ = "F": PR = 0
92 FOR YC = Y0 TO Y0 + H: Y = YC: GOSUB 60
93 ON ABS((PEEK(BYTE) AND 2 ^ BIT) <> 0) GOTO 97: IF PR=0
    THEN 96
94 PR = 0: IF FL$ = "F" THEN Y1 = YC: FL$ = "T": GOTO 96
95 GOSUB 99: FL$ = "F"
96 NEXT YC: GOTO 98
97 PR = 1: NEXT YC: IF FL$ = "T" THEN GOSUB 99
98 NEXT X: RETURN
99 FOR Y = Y1 TO YC -1: ON ABS(RND(1) < PC) GOSUB 70:
    NEXT Y: RETURN
100 REM::::::::::SAVE PICTURE
101 INPUT "ENTER FILENAME"; FILE$
102 INPUT "ENTER 8 FOR DISK, OR 1 FOR CASSETTE"; DE
103 SYS 57812 FILE$ + ".PIC", DE
104 POKE 174,64: POKE 175,127: POKE 193,0: POKE 194,96
105 SYS 62954
106 SYS 57812 FILE$ + ".COL", DE
107 POKE 174,232: POKE 175,71: POKE 193,0: POKE 194,68
108 SYS 62954: END
110 REM::::::::::DRAW A SHAPE
111 FOR J = 0 TO NL
112 E1 = L%(0,J): E2 = L%(1,J)
113 X1=E%(0,E1) + X0: Y1=E%(1,E1) + Y0
114 X2=E%(0,E2) + X0: Y2=E%(1,E2) + Y0
115 GOSUB 80
116 NEXT J
117 RETURN
120 REM::::::::::DEFINE SPRITE SP
121 FOR I = 0 TO 62
122 READ A
123 POKE 16384 + 64*SP + I,A

```



```

124 NEXT I
125 POKE 18424 + SP,SP
126 RETURN
130 REM:::TURN ON SPRITE SP
131 POKE 53269, PEEK(53269) OR 2↑SP
132 RETURN
140 REM:::TURN OFF SPRITE SP
141 POKE 53269, PEEK(53269)AND(255-2↑SP)
142 RETURN
150 REM:::X EXPAND SPRITE SP
151 POKE 53277, PEEK(53277) OR 2↑SP
152 RETURN
160 REM:::X UNEXPAND SPRITE SP
161 POKE 53277, PEEK(53277)AND(255-2↑SP)
162 RETURN
170 REM:::Y EXPAND SPRITE SP
171 POKE 53271, PEEK(53271) OR 2↑SP
172 RETURN
180 REM:::Y UNEXPAND SPRITE SP
181 POKE 53271, PEEK(53271)AND(255-2↑SP)
182 RETURN
190 REM::SPRITE SP PRIORITY OVER SHAPE
191 POKE 53275, PEEK(53275)AND(255-2↑SP)
192 RETURN
200 REM::SHAPE PRIORITY OVER SPRITE SP
201 POKE 53275, PEEK(53275) OR 2↑SP
202 RETURN
210 REM:::SET SPRITE SP TO COLOR C
211 POKE 53287 + SP,C
212 RETURN
220 REM::PLACE SPRITE SP AT X,Y
221 XX = X + 24:YY = Y + 50:Z% = XX/256
222 V = XX - Z%*256:W = 53248 + SP*2
223 WW = 53264
224 PR = ABS((PEEK(WW) AND 2↑SP)<>0)
225 VW = PEEK(WW) AND (255-2↑SP) OR (2↑SP*Z%)
226 IF PR<>Z% THEN GOSUB 140
227 POKE W,V:POKE WW,VV: GOSUB 130
228 POKE 53249 + SP*2,YY
229 RETURN
230 REM::MOVE SPRITE FROM X1,Y1 TO X2,Y2
231 DX = X2 - X1:DY = Y2 - Y1
232 L = ABS(DX):IF ABS(DY) > L THEN L = ABS(DY)
233 IF L > 0 THEN XI = DX/L:YI = DY/L
234 X = X1 + .5:Y = Y1 + .5:SD = SD + ABS(SD = 0)

```

```

235 FOR I = 0 TO L STEP SD
236 GOSUB 220
237 X = X + XI*SD:Y = Y + YI*SD
238 NEXT I
239 RETURN

```

APPENDIX C: ADDITIONAL TOOLS

In this book, you learned how to draw and paint shapes, like squares and triangles, as an introduction to drawing lines and painting shapes. You saw how these basic shapes could be used as the starting points for many familiar figures. For example, you could use a square for a house and a triangle for its roof. This appendix presents three other shapes which you will use frequently in your own drawings: rectangles, polygons and circles. As with squares and triangles, these shapes can be used in your drawings as the basic building blocks for many other figures. Before typing these tools, load Chapter 6's program and run the ZAP routine.

TOOL 240:.....DRAW A RECTANGLE

The DRAW A RECTANGLE subroutine can draw any rectangle when you give it the height, width, placement, and color of the rectangle to draw. As long as you know the top-left corner coordinates, you will not have to figure out any other coordinates. Type this new tool as:

```

240 REM:.....DRAW A RECTANGLE
241 X1 = X0 + W: Y1 = Y0
242 X2 = X0: Y2 = Y0: GOSUB 80
243 X1 = X0: Y1 = Y0 + H: GOSUB 80
244 X2 = X0 + W: Y2 = Y0 + H: GOSUB 80
245 X1 = X0 + W: Y1 = Y0: GOSUB 80
246 RETURN

```

An example program that will draw a rectangle using this tool is:

```

1200 REM:..RECTANGLE
1210 X0 = 10: Y0 = 100
1220 H = 30: W = 70
1230 C = 30: GOSUB 240

```

The top, left corner of the rectangle will be placed at X0,Y0. The rectangle will be the width of W (0-based width), and the height of H (0-based height). The outline of the rectangle will be plotted in the color represented by C's current value.

TOOL 250:.....DRAW/PAINT RECTANGLE

This subroutine will both *draw* and *paint* a rectangle, based on the same variables discussed in Tool 240 above. Obviously, if you need to have a rectangle plotted and painted, this is the tool to use. However, if you only want an outline of a rectangle, use Tool 240 instead. The new subroutine lines to type are:

```
250 REM:.....DRAW/PAINT RECTANGLE
251 GOSUB 240
252 GOTO 90
```

An example program that will draw and paint a rectangle using this tool is:

```
1300 REM::PAINT RECTANGLE
1310 X0 = 50: Y0=25
1320 W = 15: H = 10
1330 C = 46: PC = 1
1340 GOSUB 250
```

This subroutine draws a rectangle using Tool 240, and then paints it using Tool 90. The top left corner of the rectangle will be placed at X0,Y0. The rectangle will be the width of W (0-based width), and the height of H (0-based height). It will be painted in the color represented by C's current value. The percentage of pixels painted within the rectangle will be in accordance with the decimal fraction entered for PC.

TOOL 260:.....DRAWA POLYGON

A polygon is a many-sided figure. Although technically a polygon is any figure with 3 or more sides, the term is generally used to designate a figure with 5 or more sides.

The following subroutine can be used to draw polygons:

```
260 REM:.....DRAW A POLYGON
216 K = 2 * π / T - .0001
262 FOR J = 0 TO 2 * π STEP K
263 W = R * SIN(J) * 1.2345
264 H = R * COS(J) * SC
265 IF J = 0 THEN X1 = X0 + W: Y1 = Y0 + H
266 X2 = X0 + W: Y2 = Y0 + H: GOSUB 80
267 X1 = X2: Y1 = Y2
268 NEXT J
269 RETURN
```

Note that “ π ”, as shown in lines 261 and 262, can be typed by holding down a **SHIFT** key and pressing **†** immediately to the left of **RESTORE**.

If you are at all mathematically inclined, some of the equations listed above will look suspiciously like equations dealing with circles. There’s a good reason for this. The **DRAW A POLYGON** subroutine actually draws a many-sided figure, with its endpoints lying on an imaginary circle. This subroutine can be used directly from the main routine, or it can be used indirectly from the **DRAW A CIRCLE** subroutine that follows.

Example program lines that will draw a polygon using this tool are:

```
1400 REM::DRAW POLYGON
1410 X0=100: Y0=75
1420 R = 20: T = 5
1430 SC = 1: C = 62
1440 GOSUB 260
```

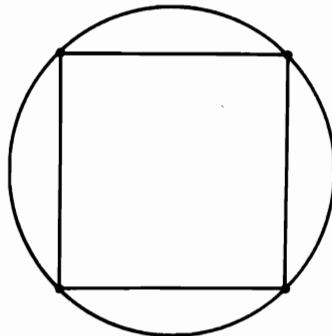
The **DRAW A POLYGON** subroutine draws a many-sided figure (5-sided in this example), using an imaginary circle as the boundary for the endpoints. We will describe the circle before the actual polygon.

Most of us have used a compass at some time or other to draw circles on paper. Using a similar method, the **DRAW A POLYGON** tool draws an imaginary circle on the computer monitor.

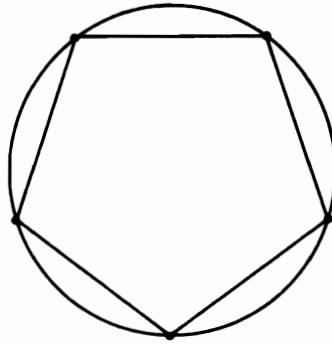
To draw a circle with a compass, you first place the compass point on the paper where the center of the circle should be. You then spread the pencil away from the center point, depending on the size of circle you desire, and you rotate the compass around the central point.

Conceptually, the **DRAW A POLYGON** subroutine does exactly the same thing. Given the central point of the circle, and the radius (distance from the compass point to the pencil), the computer has all the information necessary to draw the imaginary circle.

If you then chose 4 points on this circle to connect with straight lines, you would have a 4-sided polygon:



If you chose 5 points to connect, you would have a 5-sided polygon:



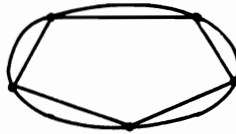
The DRAW A POLYGON subroutine essentially does the same thing. If you define the number of sides you want (T), assign offset values to X0,Y0 (central point of circle), and assign a value for the radius (R), the computer will pick its own points along the circle and draws lines between them.

First, the “scale factor” (SC) is taken into consideration. The scale factor is the imaginary circle’s *height* in relation to its width. If the circle’s height is 2 times its width, the scale factor is 2, and you really have an oval. If the circle’s height is $\frac{1}{2}$ that of its width, the scale factor is .5, and, again, you have an oval.

Assuming that the scale factor is equal to 1 (a perfect circle), the endpoints of the polygon will be evenly spaced so that all sides of the polygon are equal in length. If the scale factor is greater than 1, then the imaginary circle will be elongated, and thus the polygon will be stretched vertically:



If the scale factor is less than 1, then the circle will be flattened, and the polygon will be shortened vertically:



Line 261 of the subroutine calculates how far apart the endpoints should be, given the number of sides in the polygon and the scale factor. Line 262 begins at the bottom point in the imaginary circle. This will always be the first endpoint of the polygon that is plotted on your screen. The program then loops in order to rotate the “compass” around the central point at X_0, Y_0 , skipping over part of the imaginary circle each time it loops. Each “landing” point is plotted, and becomes an endpoint in the polygon. This loop continues until the initial endpoint at the bottom of the circle has once again been reached.

Lines 263 and 264 convert the endpoints to X, Y coordinates for use with the `FIND A POINT` and `PLOT A POINT` subroutines. Next, line 265 initializes the first endpoint (X_1, Y_1) the first time through the loop. At that stage, there are no connecting lines between any endpoints because only one endpoint on the circle has been plotted (all the rest have only been *found*).

Line 266 calculates the next endpoint (X_2, Y_2), and jumps to the `DRAW A LINE` tool to connect the current endpoint (X_2, Y_2) and previous endpoint (X_1, Y_1). This, of course, results in the first line of your polygon. Line 267 places the current endpoint coordinates into variables X_1 and Y_1 in preparation for the next loop. Line 268 sends the computer back to look for the next endpoint.

TOOL 270:::::DRAW/PAINT POLYGON

This subroutine will both *draw* and *paint* a polygon, based on the same variables discussed in Tool 260 above. If you want to draw and paint a polygon, this is the recommended tool. However, if you only want the outline of a polygon, use Tool 260 instead.

This tool should be typed as follows:

```
270 REM:::::DRAW/PAINT POLYGON
271 GOSUB 260
272 X0 = X0 - R * 1.2345
273 Y0 = Y0 - R * SC
274 H = R * 2 * SC
275 W = R * 2 * 1.2345
276 GOTO 90
```

Examples of program lines which will draw and paint a polygon using this tool are:

```
1500 REM::PAINT POLYGON
1510 X0 = 50: Y0 = 50
1530 R = 10: T = 6
1540 SC = 1: PC = 1: C = 78
1550 GOSUB 270
```

This tool uses the DRAW A POLYGON subroutine and the PAINT A SHAPE subroutine to draw and paint a polygon. For a complete explanation of the variables X0, Y0, R, T, and SC (as used here), see the discussion of Tool 260 above. PC determines the percentage of pixels to paint within the polygon. 1 (1.00) indicates that 100% of the pixels are to be painted. The value entered for C determines the color of the polygon.

TOOL 280:::::DRAW A CIRCLE

A circle is one of the hardest figures to plot on your computer. This is because circles are actually made up of many plotted pixels, or, many short, plotted lines. This DRAW A CIRCLE subroutine can quickly draw a 30-sided circle, using the same variables discussed in Tool 260 above. Type this tool as:

```
280 REM:::::DRAW A CIRCLE
281 T = 30
282 GOTO 260
```

Examples of program lines which will draw a circle using this tool are:

```
1600 REM::DRAW A CIRCLE
1610 X0 = 100: Y0 = 100
1620 R = 15: C = 110
1630 SC = 1
1640 GOSUB 280
```

Notice that to draw a circle, you use the same variables as when drawing a polygon—except that T is not defined in the main routine. This is because T gets set in the subroutine (line 281). Also, when drawing a perfect circle, always set SC equal to 1. To learn more about variables X0, Y0, and R, see the discussion on Tool 260 previously given in this appendix.

TOOL 290:.....PRINT PICTURE

This final tool will print a copy of the high resolution screen to a VIC-1525 printer. If you have such a printer, type in this tool as:

```
290 PRINT PICTURE
291 OPEN 1,4: BA = 24888
292 A$=CHR$(15) + CHR$(16) + "20 " + CHR$(8)
293 FOR J = 0 TO 44: IF (JAND7)>0 THEN BA=BA-8
294 BY = BA: PRINT #1,A$;
295 B1%=JAND7: B2%=8-B1%: FORK= 0 TO 199
296 T=PEEK(BY)*2 ↑ B1% AND 127
297 B=INT(PEEK(BY+8)/2 ↑ B2%)
298 PRINT#1, CHR$(128+T+B);
299 BY=BY+1: IF(KAND7)=7 THEN BY=BY+312
300 NEXT K: PRINT#1: NEXT J: CLOSE 1
```

To use this subroutine, take the following steps:

- (1) Check to make sure there is a picture on the high resolution screen.
- (2) Connect the printer as described in the VIC-1525 User's Manual.
- (3) Check to be sure you have paper and that the printer is on.
- (4) Type RUN 290 and press RETURN.

This subroutine takes advantage of the VIC-1525 printer's graphic ability. By chopping the high resolution picture into 7 x 7 pixel chunks, this subroutine can make the VIC-1525 print each chunk as a character. When all of the chunks are printed, you end up with a complete picture.

APPENDIX D: SPEEDING UP YOUR TOOLS

The subroutines in this book will be of great help each time you draw a picture on the Commodore 64. Unfortunately, the subroutines are not always as quick as they are useful. At times, it could take up to 20 full minutes to run a picture-drawing program. The problem lies in the fact that BASIC, the programming language you have used throughout this book, is not the computer's "native" language. The computer's native language is *machine* language. In order for the computer to understand the programs you have entered, it has a little translator that reads your BASIC statements, and then translates them into machine language. This can be pretty time-consuming.

To speed things up, you can modify some of the slower tools to take advantage of machine language. This takes a considerable amount of initial typing, but will be

well worth it in the end. If you are interested, load Chapter 6's program and run the ZAP routine (type RUN 10 and press RETURN). Begin by modifying your tools as follows:

```
1 GOTO 500

41 SYS 49165,C
(delete lines 42 and 43)

51 SYS 49157
(delete lines 52 and 53)

71 SYS 49321,X,Y,C
(delete lines 72 and 73)

81 SYS 49321,X1,Y1 TO X2,Y2,C
(delete lines 82 through 88)

91 SYS 49551,X0,Y0,W,H,C,PC
(delete lines 92 through 98)
99 RETURN
```

You now need to type several sections of data statements. These data statements store machine language versions of the modified tools. The first section to type is:

```
500 FOR I = 49152 TO 49189
501 READ A: POKE I,A: T = T+A
502 NEXT I
503 IF T<>5205 THEN PRINT "ERROR IN 500-516":STOP
504 T=0
510 REM::::::::::CLEAR AND PAINT
511 DATA 134, 32, 0, 0, 0,169, 0
512 DATA 160, 96,162, 32,208, 8, 32
513 DATA 241,183,138,160, 68,162, 4
514 DATA 132,252,160, 0,132,251,145
515 DATA 251,200,208,251,230,252,202
516 DATA 208,246, 96
```

Now, run the program. Either *nothing* will happen, or you will get an error message. If an error message occurs, check your typing and correct any mistakes. If no error message occurs, then this section has been typed correctly, and you are ready to move on to the next section.

Below are 6 program sections for you to type in. When you have finished typing a section, run the program. If nothing happens, move on to the next section. If an error message occurs, check your typing and correct all errors. Do *not* move on to a new section until all errors have been corrected.

When all sections have been typed and corrected, save these modified tools under TOOL BOX. This tool box will work in exactly the same manner as you were taught in the book. The difference will be in its operation. Try drawing and painting a simple shape. You'll be pleasantly surprised.

```
520 FOR I = 49190 TO 49263
521 READ A: POKE I,A: T = T+A
522 NEXT I
523 IF T<>8819 THEN PRINT"ERROR IN 520-541":STOP
524 T=0
530 REM::::::::::FIND A POINT
531 DATA 173, 62, 3, 72, 41,248,168
532 DATA 32,162,179,169, 0,160,192
533 DATA 32, 40,186, 32,247,183, 24
534 DATA 173, 60, 3, 72, 41,248,101
535 DATA 20,133,251,133,253,173, 61
536 DATA 3,101, 21, 72, 74,102,253
537 DATA 74,102,253, 74,102,253, 24
538 DATA 105, 68,133,254,104,105, 96
539 DATA 133,252,104, 41, 7,170,104
540 DATA 41, 7,101,251,144, 2,230
541 DATA 252,133,251, 96
```

Stop and run the program here.

```
550 FOR I = 49264 TO 49367
551 READ A: POKE I,A: T=T+A
552 NEXT I
553 IF T<>10943 THEN PRINT"ERROR IN 550-575":STOP
554 T=0
560 REM::::::::::MISC. ROUTINES
561 DATA 162, 64, 44,162, 69, 44,162
562 DATA 74, 44,162, 79, 44,162, 84
563 DATA 160, 3, 76,212,187,169, 64
564 DATA 44,169, 69, 44,169, 84,160
565 DATA 3, 76,162,187, 32,124,192
566 DATA 32,247,183,166, 20,164, 21
567 DATA 142, 89, 3,140, 90, 3, 96
568 DATA 128, 64, 32, 16, 8, 4, 2
569 DATA 1, 32,253,174, 32,235,183
570 DATA 142, 62, 3,166, 20,164, 21
571 DATA 142, 60, 3,140, 61, 3,201
572 DATA 164,240, 24, 32,241,183,142
573 DATA 63, 3, 32, 38,192,160, 0
574 DATA 177,251, 29,161,192,145,251
575 DATA 173, 63, 3,145,253, 96
```

Stop and run the program here.

```
580 FOR I = 49368 TO 49444
581 READ A: POKE I,A: T=T+A
582 NEXT I
583 IF T<>7925 THEN PRINT"ERROR IN 580-601":STOP
584 T=0
590 REM::::::::::PLOT PART 1
591 DATA 32,115, 0, 32,138,173, 32
592 DATA 15,188,172, 60, 3,173, 61
593 DATA 3, 32,145,179, 32,112,192
594 DATA 32, 83,184, 32,118,192, 70
595 DATA 102, 32,144,192, 32,241,183
596 DATA 138,168, 32,162,179, 32, 15
597 DATA 188,172, 62, 3, 32,162,179
598 DATA 32,115,192, 32, 83,184, 32
599 DATA 121,192, 70,102,169, 84,160
600 DATA 3, 32, 91,188, 48, 11, 32
601 DATA 43,188,208, 3, 76,192,192
```

Stop and run the program here.

```
610 FOR I = 49445 TO 49550
611 READ A: POKE I,A: T=T+A
612 NEXT I
613 IF T<>11077 THEN PRINT"ERROR IN 610-636":STOP
614 T=0
620 REM::::::::::PLOT PART 2
621 DATA 32,144,192, 32,137,192,169
622 DATA 74,160, 3, 32, 15,187, 32
623 DATA 118,192, 32,137,192,169, 79
624 DATA 160, 3, 32, 15,187, 32,121
625 DATA 192, 32,241,183,142, 63, 3
626 DATA 32,198,192, 32,131,192,169
627 DATA 74,160, 3, 32,103,184, 32
628 DATA 43,188, 48, 52, 32,112,192
629 DATA 32,247,183,165, 20,166, 21
630 DATA 141, 60, 3,142, 61, 3, 32
631 DATA 134,192,169, 79,160, 3, 32
632 DATA 103,184, 32, 43,188, 48, 21
633 DATA 32,115,192, 32,247,183,165
634 DATA 20,141, 62, 3,206, 89, 3
635 DATA 208,191,206, 90, 3, 16,186
636 DATA 96
```

Stop and run the program here.

```
640 FOR I = 49551 TO 49658
641 READ A: POKE I,A: T=T+A
642 NEXT I
643 IF T<>9829 THEN PRINT"ERROR IN 640-666":STOP
644 T=0
650 REM::::::::::PAINT A SHAPE PART 1
651 DATA 32, 89,194,141, 60, 3,140
652 DATA 61, 3,142, 66, 3, 32, 89
653 DATA 194,141, 64, 3,140, 65, 3
654 DATA 142, 67, 3, 32,241,183,142
655 DATA 63, 3, 32,253,174, 32,138
656 DATA 173, 32,118,192,169, 0,141
657 DATA 72, 3,141, 73, 3,173, 66
658 DATA 3,141, 69, 3,173, 67, 3
659 DATA 141, 68, 3,173, 69, 3,141
660 DATA 62, 3, 32, 38,192,160, 0
661 DATA 177,251, 61,161,192,208, 52
662 DATA 173, 73, 3,240, 34,169, 0
663 DATA 141, 73, 3,173, 72, 3,208
664 DATA 16,173, 69, 3,141, 70, 3
665 DATA 169, 1,141, 71, 3,141, 72
666 DATA 3,208, 8
```

Stop and run the program here.

```
670 FOR I = 49659 TO 49763
671 READ A: POKE I,A: T=T+A
672 NEXT I
673 IF T<>11207 THEN PRINT"ERROR IN 670-695":STOP
680 REM::::::::::PAINT A SHAPE PART 2
681 DATA 32, 59,194,169, 0,141, 72
682 DATA 3,238, 69, 3,238, 71, 3
683 DATA 206, 68, 3,208,188,240, 21
684 DATA 169, 1,141, 73, 3,238, 69
685 DATA 3,206, 68, 3,208,173,173
686 DATA 72, 3,240, 3, 32, 59,194
687 DATA 238, 60, 3,208, 3,238, 61
688 DATA 3,206, 64, 3,208,132,206
689 DATA 65, 3, 48, 3, 76,182,193
690 DATA 96,173, 70, 3,141, 62, 3
691 DATA 32,190,224,169, 74,160, 3
692 DATA 32, 91,188, 16, 3, 32,198
693 DATA 192,238, 62, 3,206, 71, 3
```


Commodore Staff, *Commodore 64 User's Guide*, First Edition, Wayne, PA: Commodore Business Machines, Inc., and Howard W. Sams & Co., Inc., 1982

Commodore Staff, *VIC-1541 Single Drive Floppy Disk User's Manual*, Second Edition, Commodore Business Machines, Inc., 1982

Gracely, Jim, "Bit-Mapped Graphics on the Commodore 64," *Commodore Power/Play Magazine*, Vol. II, No. 2 (Summer, 1983), pp. 47-49

Hampshire, Nick, *VIC Revealed*, Rochelle Park, NJ: Hayden Book Co., Inc., 1982

Lane, John Michael, "A Sprite Editor for the Commodore 64," *Creative Computing*, Vol. 9, No. 9 (September, 1983), pp. 290-293

Myers, Roy E., *Microcomputer Graphics*, Reading, MA: Addison-Wesley Publishing Co., 1982

Newman, William M. and Robert F. Sproull, *Principles of Interactive Computer Graphics*, New York, NY: McGraw-Hill Book Co., 1979

Petts, Ronald A., "A Shape Generator for the Commodore 64," *Compute!*, Vol. 4., No. 11 (November, 1982), pp. 160-163

West, Raeto Collin, *Programming the PET/CBM*, Greensboro, NC: Compute! Books, 1982

HIGH RESOLUTION COLOR CHART

BACKGROUND COLORS

FOREGROUND COLORS	B	W	R	C	P	G	B	Y	O	B	R	G	G	G	B	G
	L	H	E	A	U	R	L	E	R	R	D	R	R	R	L	R
	K	T	D	N	P	N	U	L	G	N	2	1	2	2	2	3
Black	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
White	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31
Red	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47
Cyan	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63
Purple	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79
Green	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95
Blue	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111
Yellow	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127
Orange	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143
Brown	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159
Lt. Red	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175
Gray 1	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191
Gray 2	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207
Lt. Green	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223
Lt. Blue	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239
Gray 3	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255

APPENDIX G COLOR CHARTS

APPENDIX H

TOOL KIT REFERENCE CARD

(Under "How To Use," you will find that all variables are set equal to "#". See back of this card for the value ranges allowed for each variable.)

TOOL #	DESCRIPTION	HOW TO USE
10	DELETE MAIN ROUTINE	<i>Type RUN 10 and press [RETURN]</i>
20	ENTER GRAPHICS MODE	GOSUB 20
30	RETURN TO TEXT MODE	GOSUB 30
40	SET SCREEN COLORS	C = #: GOSUB 40
50	PAINT BACKGROUND	GOSUB 50
60	FIND A POINT	X=#: Y=#: GOSUB 60
70	PLOT A POINT	X=#: Y=#: C=#: GOSUB 70
80	PLOT A LINE	X1=#: Y1=#: X2=#: Y2=#: C=#: GOSUB 80
90	PAINT A SHAPE	X0=#: Y0=#: W=#: H=#: PC=#: C=#: GOSUB 90
100	SAVE PICTURE	<i>type RUN 100 and press [RETURN]</i> <i>enter filename</i> <i>enter device # (8 or 1)</i>
110	DRAW A SHAPE	fill E%(1,#): fill L%(1,#) NL=#: C=#: X0=#: Y0=#: GOSUB 110
120	DEFINE SPRITE SP	<i>enter 63 data items in data statements</i> SP=#: GOSUB 120
130	TURN ON SPRITE SP	SP=#: GOSUB 130
140	TURN OFF SPRITE SP	SP=#: GOSUB 140
150	X EXPAND SPRITE SP	SP=#: GOSUB 150
160	X UNEXPAND SPRITE SP	SP=#: GOSUB 160
170	Y EXPAND SPRITE SP	SP=#: GOSUB 170
180	Y UNEXPAND SPRITE SP	SP=#: GOSUB 180
190	SPRITE PRIORITY OVER SHAPE	SP=#: GOSUB 190
200	SHAPE PRIORITY OVER SPRITE	SP=#: GOSUB 200
210	SET SPRITE SP TO COLOR C	C=#: SP=#: GOSUB 210
220	PLACE SPRITE SP AT X,Y	X=#: Y=#: SP=#: GOSU 220
230	MOVE SPRITE FROM X1, Y1, TO X2, Y2	X1=#: Y1=#: X2=#: Y2=# SP=#: GOSUB 230

Cut Here

VARIABLE LIST

The following variables are commonly needed by this book's subroutine tools:

<u>Variable</u>	<u>Description</u>	<u>Value Range</u>
X0	X Offset of Shapes	0 - 319
Y0	Y Offset of Shapes	0 - 199
X1	Initial X Coordinate	0 - 319
Y1	Initial Y Coordinate	0 - 199
X2	Final X Coordinate	0 - 319
Y2	Final Y Coordinate	0 - 199
C	Color Code	0 - 255
X	X Coordinate	0 - 319
Y	Y Coordinate	0 - 199
W	Width of Shape	0 - 319
H	Height of Shape	0 - 199
PC	% of Area to Paint	0.0 - 1.0
NL	# of Lines in Shape	>0
E%	List of endpoint data	N/A
L%	List of line data	N/A
SP	Sprite Number	0 - 7
SD	Speed of Sprite	>0

The following variable are needed by some of the additional tools provided in Appendix C:

<u>Variable</u>	<u>Description</u>	<u>Value Range</u>
R	Radius of Shape	0 - 100
T	Number of Sides for Polygon	>2
SC	Vertical Scale for Shape	1=Normal, <1=Shorter <1=Taller

COMMODORE 64 COLOR GRAPHICS: A BEGINNER'S GUIDE

Designed by Ellis Ross-Tami, Los Angeles

Commodore 64 Color Graphics: A Beginner's Guide is a step-by-step guide to creating animated color graphics on your personal computer.

The easy to follow yet comprehensive instructions give you everything you need in order to perform the computer "magic" that transforms your blank screen into a detailed sailing scene complete with animation. In addition, you will have developed an entire "tool kit" of graphics programs. You can use this tool kit at any time to have an automatic head-start in any graphics adventures you begin on your own.

You'll learn the basics of design and programming techniques at your own pace. The instructions show you how to save programs at the end of every chapter, giving you a logical breakpoint for saving your work so you can continue at a later time.

Every program that you need to enter is given to you and thoroughly explained, to help you understand exactly what the BASIC commands mean. You will also see many illustrations and photos that help to simplify programming concepts and allow you to check your progress.

Even first-time computer users will have an enjoyable time discovering Commodore 64 graphics.

This unique format of step-by-step programming instructions, comprehensive illustrations and photographs, and interesting programming exercises will speed you on your voyage through the world of computer graphics.

With *Commodore 64 Color Graphics: A Beginner's Guide* you will have everything you need to begin your adventure.

THE BOOK COMPANY

A Division of Arrays, Inc.

\$14.95
U.S.A.

ISBN 0-912003-06-5