COMMODORE 64 COLOR GRAPHICS: AN ADVANCED GUIDE

By Shaffer & Shaffer Applied Research & Development

COMMODORE 64 COLOR GRAPHICS: AN ADVANCED GUIDE

By:
Shaffer & Shaffer Applied Research
& Development

RESTON PUBLISHING COMPANY, INC.
A Prentice-Hall Company
Reston, Virginia

ACKNOWLEDGEMENTS

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

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

General Editor
Robert P. Wells, Ph.D.

Graphics Production
Estela Montesinos
Steve Gunn

ISBN 0-8359-0787-2

TABLE OF CONTENTS

INTRODUCTION

Welcome to the world of Commodore 64 advanced graphics—a dynamic world of colors, imagination, intrigue, and, of course, fun. The fact that you purchased an advanced book suggests that you are already familiar with computer art, or the Commodore 64, or both. As we explain later in this section, some prior computer experience will be necessary.

If you got this book to learn advanced programming techniques for color graphics, you will appreciate the programs covered in these chapters. Programming code that can rotate, scale, and translate (move) images on the screen has been included for use in both simple and complex compositions. Also included are instructions on creating multi-colored images, as well as advanced methods of sprite manipulation. Carefully planned lessons will help you understand how and when you should use each new-found skill. The result will be pictures for school, work, entertainment—*anything.*

If your interest lies in learning more on the use of colors, tones, repetition, patterns, and other artistic techniques, you won't be disappointed. We've provided you with useful tips, suggestions, and facts to help you effectively put your ideas on the screen. This includes over 30 sketched designs illustrating how you can better take advantage of each graphic concept introduced.

The idea of this book is to go one step beyond the technical aspects of computer art. After answering the question ***"How can I rotate a figure?"***, it is just as important to answer the question *"What can I do with rotation?"* Exactly what doors are opened once the programs have been entered? That is what you will explore in the coming chapters.

What You Should Know

In order to write an advanced book, we've made some assumptions about the experience level of our readers. If you have already gone through our *Commodore 64 Color Graphics: A Beginner's Guide,* you are ready for this book.

If not, the first requirement is that you have a good feel for the Commodore 64 keyboard and its collection of special keys. In this text, special keys are printed in **boldface** to distinquish them from the rest of the text. So, for example, when you read "Press the **RETURN** key," you know to press the key marked RETURN on your keyboard (do *not* type R-E-T-U-R-N).

Your programming skills do not have to be extensive, but should include a first-hand knowledge of each item listed below (recommended reading is listed in the right-hand column):

	Commodore 64 User's Guide
—variables (e.g., A, B$, T%)	95-103,112-113
—PRINT statement	23-29,123-124
—GOTO statement	32-34,120
—GOSUB/RETURN statements	120,124

	Commodore 64 User's Guide
—GET statement	47-48,119-120
—POKE statement	60-61
—IF/THEN statements	37-39,120-121
—REM statement	124
—computer memory	2-4,62-65,142
—arrays (computer lists)	95-103
—FOR/NEXT loops	39-40,43-45,119,121-122
—SAVE statement (saving programs)	21,116
—LOAD statement	18-20,115

Finally, you should have some experience with high resolution graphics. You should know about foreground/background colors, screen color blocks, screen memory versus color memory, and the X,Y coordinate system. If you need a review of beginner's graphics, try our *Commodore 64 Color Graphics: A Beginner's Guide,* or the *Commodore 64 Programmer's Reference Guide.*

How to Use This Book

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

—A Commodore 64 computer;

—A video monitor or TV screen (preferably color);

—A Commodore 64 disk drive with a properly formatted diskette, *or* a cassette recorder for the Commodore 64 with a blank cassette tape; and

—Some graph paper to work out your own designs (optional).

Each time you sit down to use this book, you should be at your computer. All equipment should be properly set up and turned on. Information on connecting your computer and monitor is provided in the Commodore 64 User's Guide. Disk drive installation is covered in the manual(s) provided with the disk drive itself. (This manual also covers formatting a diskette.) When the system is turned on, your screen should display "**** COMMODORE 64 BASIC V2 ****" at the top. Only then will you be ready to begin a session with this book.

A "session" can be as long or a short as you like. That is the beauty of working with programs. At the end of each session, simply save the current form of your program. You can then turn the computer off and take a nap, watch TV, or visit your friends. Later, you can easily return to your work by loading the program back into memory, and then picking up in the chapter where you left off.

Each chapter ends at a logical breaking point. This makes it easy for you to read a chapter, SAVE your program, take a break, and then continue later. For this reason, we ask you to SAVE your program at the end of each chapter. When you begin reading the next chapter, you are asked to LOAD the program back into memory. As a general rule, it is a good idea to SAVE and LOAD your program whenever instructed.

The general format of each chapter is as follows:

—New graphics and design concepts are introduced;

—New program lines are typed;
—The program is RUN and discussed in depth;
—Any additional design ideas and sketches are introduced where appropriate;
—All key technical and artistic points are summarized.

In the chapters, each programming technique is packaged as a useful subroutine "tool" that can be inserted and used in any picture-drawing program you create. In fact, by the end of Chapter 8 you will have a complete "tool kit" containing over 20 graphics subroutines. Need to draw a line? No problem. Just pick up the DRAW A LINE tool, specify where you want the line drawn, and the job is done. (This will become clear in Chapter 1.)

Another important aspect of this book is that it concentrates on teaching *how* pictures can be drawn on the Commodore 64. Often, knowing *why* things work 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 placed in a box. These technical descriptions can be read or passed over, as you please. Passing over a technical description will in no way keep you from learning how to create your graphics displays.

As a final comment, practice what you learn before moving on from one chapter to the next, and do not skip chapters. If you have difficulty with some of the material, read through it again and re-try each example you are given. It will be through repetition that your skills are retained and refined.

What You Can Expect to Learn

This book sets out to accomplish two things:

(1) Provide you with advanced programming techniques for color graphics; and

(2) Show you how and where these techniques can be applied to produce more professional looking pieces of artwork.

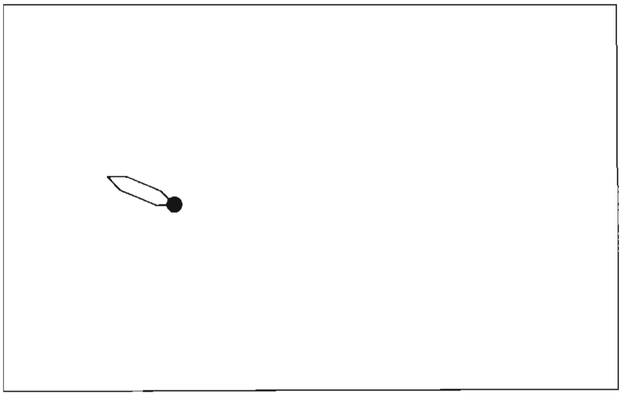
To accomplish the first of these goals, you will learn how to:

—plot points and lines*
—store and retrieve shapes
—draw shapes
—paint shapes
—translate shapes
—rotate shapes
—scale shapes

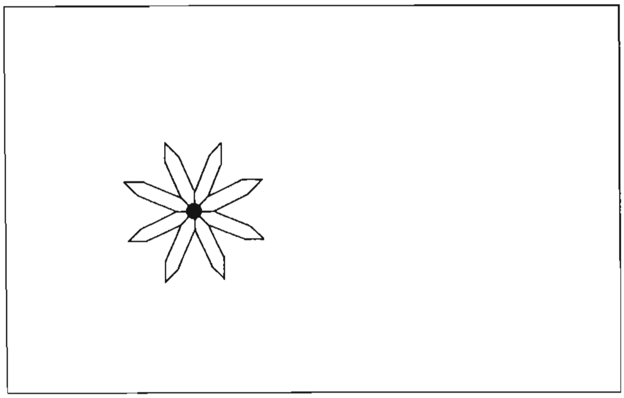
If you already have experience making and moving sprites but want to learn more about them, we also cover connecting sprites to joysticks, and sprite collision detection. Sprites are small, arcade-like figures that can move around on your screen. The ability to create moving designs is just one of the advantages computer art has over sketchpads and canvases.

*These are beginning graphics concepts, and are not discussed in as much detail as the others.

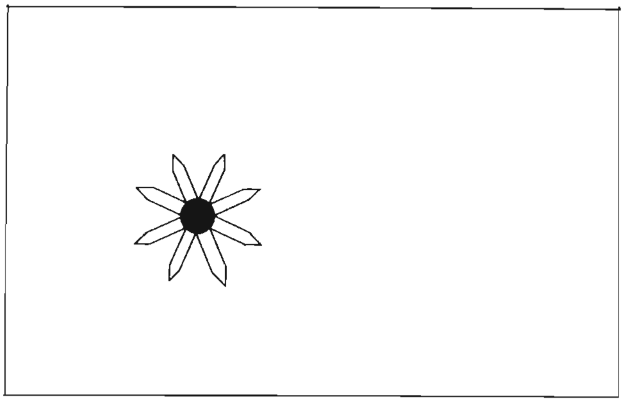
To give you an idea of what some of this means, consider the two basic shapes sketched below.



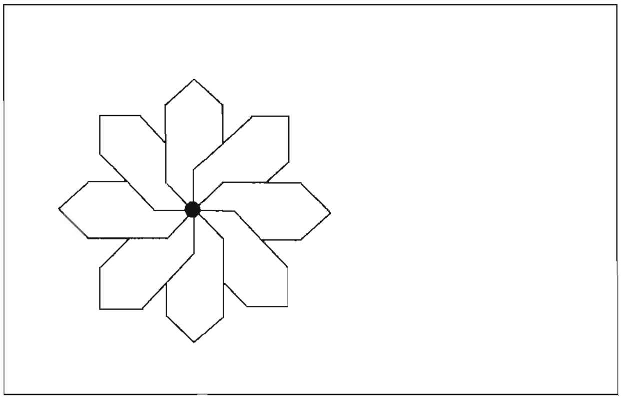
By rotating the petal shape, you arrive at a flower:



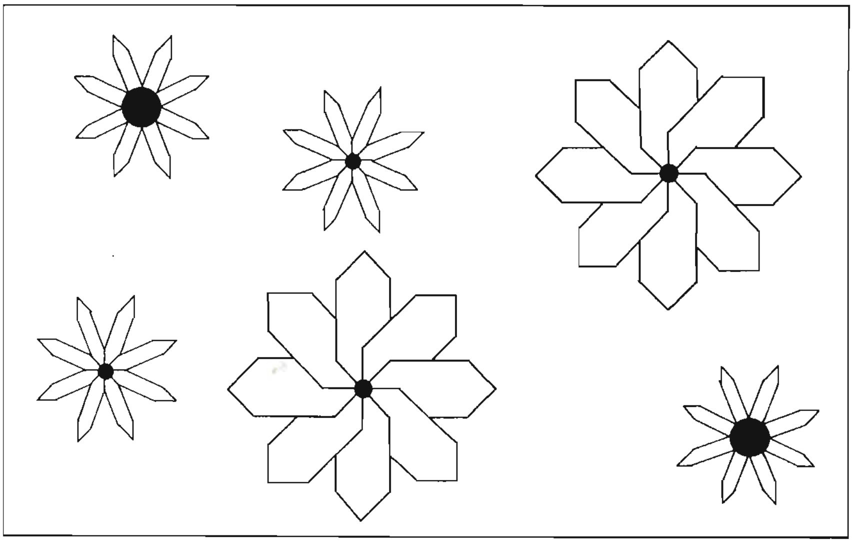
By scaling the center piece, you change the appearance of the flower:



By scaling the petals instead, you achieve another form of the flower:



Finally, by translating all of the flower types, you arrive at a floral display:



To meet the second objective, teaching art concepts, we gave special consideration to those art ideas that related specifically to our programs. Some of the topics discussed include:

—patterns
—repetition
—tone or "value" variation
—the illusion of depth
—the use of horizon lines
—variety through scaling
—using shapes to create other shapes
—the effect of shape placement/size.

For most people, drawing does not come naturally. Fortunately, there are most specific guidelines, "tricks of the trade" if you will, that are easy to learn, understand and apply. For example, you will see how a "horizon line" can significantly add to the feeling of depth in a picture. You will learn about "negative space," and why it is an important consideration in each of your designs. These and other simple facts about design control are discussed and illustrated as you proceed through the chapters.

Chapter One

SETTING UP THE PROGRAM

In order to work on advanced graphics, you have to start with some basic graphics tools. As fundamental as "plot a point" might be, there really can be no advanced graphics without it. In this chapter, we will set you up with tools that can do the following:

—"turn on" graphics mode
—"turn off" graphics mode
—clear the graphics screen and set the background color
—plot a point
—plot a line
—erase the main routine

There are several approaches to placing these tools in a program. We have taken the approach of creating a *subroutine* for each. This saves you the trouble of re-typing them every time you need one in your program. Instead, you set a few variables and insert a GOSUB. By the end of this book you will have a whole range of subroutine tools, ranging from the very simple to the very complex. The "main routine" of your program will vary from picture to picture, but the subroutines will remain the same.

We also chose to take advantage of *machine language*. If you've ever written BASIC programs that draw pictures, you are no doubt aware of the time it can take to run the finished program. This is because BASIC is not a language the computer immediately understands. Instead, it must first "translate" each BASIC statement into machinge language. Only then can it carry out the instructions it finds.

We felt that the time it takes to convert BASIC into machine language was too long for an advanced book. So, in the next section, you will enter some machine language as data statements to streamline and speed up a few of the slower tools. The result will be dramatic.

You will find that the main thrust of this chapter is to set you up for advanced graphics. This involves getting some beginning graphics programming and some machine language typing out of the way.

The Joy of Machine Language

This section's title expresses a mixture of both admiration and sarcasm. There's no doubt about it, nothing beats machine language for speed. Unfortunately, it is not nearly as simple to learn or understand as it is fast. This section does not attempt to explain any part of machine language to you. Instead, you will learn what to type, why it will help you, and how to check it.

1 SETTING UP THE PROGRAM

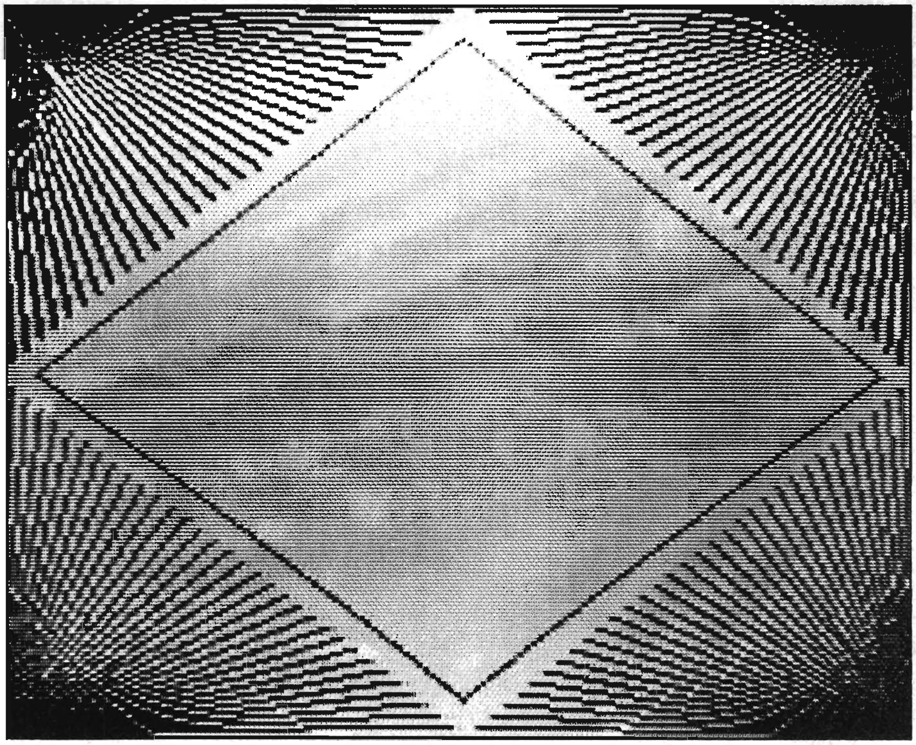
You will perform three steps to enter the machine language data. The steps are:

(1) Enter a small program to help you type the machine language.

(2) Entering the machine language data.

(3) Enter a program that double-checks the machine language for accuracy.

Initially, this may seem like a lot of work. However, spending 45 minutes of typing time now can save you hours of plotting time in the future. To instill enough incentive to get you through the next few pages, we have provided the picture below. You may need to flip back to it from time to time to keep yourself going. This relatively simple picture took 28 *minutes* to plot using BASIC alone, while taking only 41 *seconds* to plot using machine language.



The HELPER Program

Several hundred numbers need to be accurately POKEd into memory. This, needless to say, is quite a task. In addition, there will be many occasions when you need to PEEK into memory to check your entries, more typing.

To aid you in this process, we provide a HELPER program on page 13 that will do all of the repetitious typing for you. This will save you time and also reduce the possibility of typing errors. In addition, the HELPER program produces a "check number" after every eight pieces of data are entered. By comparing this number to one in our text, you can check to make sure you are entering the data correctly.

You will be told to SAVE this HELPER program after typing it. Be prepared with a formatted diskette or blank tape on hand. When you are ready, read the list of instructions below and then type the HELPER program on your Commodore.

—If you own a machine language monitor program that is easy to use *and* you understand how it works, you may use it instead of the HELPER program. If you don't know what a machine language monitor program is, it probably won't help you.

—Type slowly: accuracy is far more important than speed.

—Type in lower-case. This makes it easier to spot errors. To change to lower-case, hold down a **SHIFT** key and press **C=** (located on the lower, left-hand side of your keyboard).

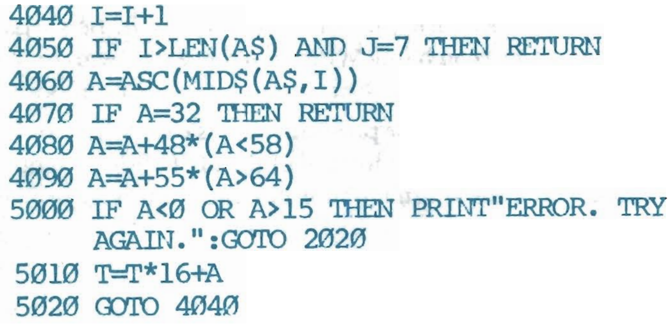
—If you have trouble seeing your typing, press **CTRL** and **2** at the same time. This changes your typing to white.

—If you have a habit of typing oh's for zeroes, or small L's for ones, you must break that habit now. The computer expects numbers typed where numbers are intended.

—Carefully check over your typing when you are done.

Begin typing:

```
2ØØØ REM :::::: HELPER PROGRAM
2Ø1Ø PRINT CHR$(147) CHR$(18) SPC(15) "HELPER"
2Ø2Ø A$="": INPUT "MEM/DATA"; A$: IF A$="" THEN END
2Ø3Ø I=Ø: J=7: GOSUB 4Ø3Ø: REM GET ADDR
2Ø4Ø ADDR=T: IF T<49152 OR T>5Ø5Ø4 THEN PRINT
     "ERROR. TRY AGAIN.": GOTO 2Ø2Ø
2Ø5Ø IF LEN(A$) = 28 THEN 3Ø7Ø: REM POKER
2Ø6Ø IF LEN(A$)>4 THEN PRINT "ERROR. TRY AGAIN.":GOTO 2Ø2Ø
2Ø7Ø CK=Ø
2Ø8Ø FOR I = Ø TO 7
2Ø9Ø PRINT " ";
3ØØØ P%=PEEK(ADDR+I): CK=CK+P%
3Ø1Ø PH% = P%/16: PL% = P%-PH%*16
3Ø2Ø IF PH%>9 THEN PH%=PH%+7
3Ø3Ø IF PL%>9 THEN PL%=PL%+7
3Ø4Ø PRINT CHR$(PH%+48) CHR$(PL%+48);
3Ø5Ø NEXT I:PRINT:PRINT"SUM FOR THIS ROW:" CK:PRINT
3Ø6Ø GOTO 2Ø2Ø
3Ø7Ø CK=Ø
3Ø8Ø FOR J = Ø TO 7
3Ø9Ø GOSUB 4Ø3Ø: CK=CK + T
4ØØØ POKE AD+J,T
4Ø1Ø NEXT J: PRINT"SUM FOR THIS ROW:" CK:PRINT
4Ø2Ø GOTO 2Ø2Ø
4Ø3Ø T=Ø
```



```
4Ø4Ø I=I+1
4Ø5Ø IF I>LEN(A$) AND J=7 THEN RETURN
4Ø6Ø A=ASC(MID$(A$,I))
4Ø7Ø IF A=32 THEN RETURN
4Ø8Ø A=A+48*(A<58)
4Ø9Ø A=A+55*(A>64)
5ØØØ IF A<Ø OR A>15 THEN PRINT"ERROR. TRY
     AGAIN.":GOTO 2Ø2Ø
5Ø1Ø T=T*16+A
5Ø2Ø GOTO 4Ø4Ø
```

Carefully double-check your typing when you are done, and then SAVE this program under the name "HELPER". After saving any program, always use the VERIFY command to make sure that the program *did* get saved. A summary of the SAVE, VERIFY, LOAD, and LIST commands is given below.

To SAVE on disk, type:	SAVE *"filename"*,8
To SAVE on tape, type:	SAVE *"filename"*,1
To VERIFY on disk, type:	VERIFY *"filename"*,8
To VERIFY on tape, type:	VERIFY *"filename"*,1
To LOAD from disk, type:	LOAD *"filename",8*
To LOAD from tape, type:	*LOAD "filename",1*
*To re-SAVE a program on disk, type:	*SAVE "@O:filename"*,8
to re-SAVE on tape:	*N/A. Save the revised program at the end of the tape.*

* The @0 command is one that allows you to erase and replace a program on diskette, using the same filename. This command has a history of problems, and we therefore do not recommend using it. An alternative is to re-name each modified or corrected program with a filename similar to the original program (i.e., "HELPER", "HELPER.1", "HELPER.2", etc.).

To use the above commands now, be sure to replace *"filename"* with "HELPER" (including quotes). When working with programs of your own, *filename* can be replaced with any 16-character name you wish to assign to the program.

If you are working with a cassette recorder, you will have to make use of your COUNTER with every SAVE, VERIFY, and LOAD command. In addition, the screen will present you with several messages as the commands are executed (e.g., PRESS PLAY AND RECORD). Follow the screen's instructions at all times. If nothing seems to be happening, try pressing C=. This keypress is necessary at certain times in the LOAD and VERIFY commands.

With the HELPER program safely stored on disk/tape, you can now try it out to see just exactly how helpful it is.

Entering the Machine Language

You need to enter the machine language data blocks that begin on page 16. Each *row* in a block contains the following:

—A memory location (e.g., C000);
—Several pieces of data (e.g., 80, 40, 20, 10, etc.);
—A shaded number.

What you need to do is POKE the data from each row into a set of eight memory locations, beginning with the memory location given in the row. (The shaded number is a "typing checker," and will be explained later.)

Normally, you would have to type many, many POKE statements to accomplish this job. Because of the HELPER program, all you need to do is type the numbers themselves. To get started, LOAD and RUN the HELPER program.

> NOTE: We will now describe what should happen on your screen with the HELPER program. If anything else happens, STOP the program. Search for and correct all typing errors. Then re-run the HELPER program.

Your screen will clear, and the title "HELPER" will be centered at the top. Beneath this, the message "MEM/DATA" will be printed. This message stands for "Memory Location/Data," and it directs you to begin entering the first memory location and corresponding eight pieces of data. Look at the first row of your data blocks:

```
CØØØ  8Ø  4Ø  2Ø  1Ø  Ø8  Ø4  Ø2  Ø1              255
```

To enter this row into the HELPER program, type it exactly as it is shown, *except* for the shaded number 255. Be sure to put a space between each piece of data. Use the cursor control keys to back up and correct errors whenever necessary. When you've correctly entered this row, press **RETURN**.

Next you will see the message "SUM FOR THIS ROW: 255." 255 is an important number. Compare it to the shaded number in our text for this row. You will see that the shaded number and the SUM FOR THIS ROW number match (or should). This is how you can find out if you've typed a row correctly. Whenever the computer displays a different SUM than the shaded one in our book, you have made a typing error in that row. To correct it, simply re-type the entire row at the next "MEM/DATA" prompt. The new information will over-write (erase and replace) the incorrect information.

Now type each row of each block into the program. You must enter the data blocks one complete row at a time, followed by a press of the **RETURN** key. If you want to re-check a line, type the memory location only (remember, this is the first number on each row), and press **RETURN**. The computer will then display each number you typed for that row, in the same order you typed them. Most importantly, check each row sum against our pre-calculated shaded sum.

To stop the HELPER program, press **RETURN** at a "MEM/DATA" prompt before entering any information. If you decide to stop midway through the machine language program lines, be sure to read the "Taking a Break" section that follows the machine language data. Before you resume typing after a break, be sure to read the "Picking Up Where You Left Off" section on the next page. The "Taking a Break" section discusses how to save your work. As a precaution, we recommend saving your work at the end of each data block.

> *Note:* The HELPER program has a few built-in error checking devices. It does *not* check for every posssible error—only a few of the more obvious ones. If the message "ERROR. TRY AGAIN." appears at any time, you should re-tvpe the row you last typed into the program.

```
Block #1                                          Checker Values

CØØØ  8Ø  4Ø  2Ø  1Ø  Ø8  Ø4  Ø2  Ø1               255
CØØ8  CØ  3Ø  ØC  Ø3  2Ø  F1  B7  8A               849
CØ1Ø  8D  21  DØ  ØA  ØA  ØA  ØA  18               446
CØ18  65  65  AØ  44  A2  Ø4  2Ø  27               667
CØ2Ø  CØ  A9  ØØ  AØ  6Ø  A2  2Ø  84               943
CØ28  FC  AØ  ØØ  84  FB  91  FB  C8              1391
CØ3Ø  DØ  FB  E6  FC  CA  DØ  F6  6Ø              1693
CØ38  A9  ØØ  85  FC  AD  3E  Ø3  48               864
CØ4Ø  29  F8  85  FB  ØA  26  FC  ØA               983
CØ48  26  FC  65  FB  9Ø  Ø2  E6  FC              127Ø
CØ5Ø  ØA  26  FC  ØA  26  FC  ØA  26               648
CØ58  FC  85  FB  AD  3C  Ø3  29  F8              1161
CØ6Ø  65  FB  85  FB  85  FD  AD  3D              1356
CØ68  Ø3  65  FC  48  4A  66  FD  4A               931
CØ7Ø  66  FD  4A  66  FD  18  85  FE              1195
CØ78  68  69  6Ø  85  FC  68  29  Ø7               842
```

```
Block #2

CØ8Ø  65  FB  9Ø  Ø2  E6  FC  85  FB              1364
CØ88  AD  3C  Ø3  29  Ø7  AE  44  Ø3               529
CØ9Ø  FØ  Ø3  4A  Ø9  Ø8  AA  BD  ØØ               693
CØ98  CØ  8D  48  Ø3  6Ø  AØ  ØØ  8C               8Ø4
CØAØ  47  Ø3  AD  3F  Ø3  Ø9  FØ  CD               767
CØA8  21  DØ  FØ  6A  A9  44  18  65               949
CØBØ  FE  85  FE  AD  44  Ø3  DØ  ØB              11Ø4
CØB8  2Ø  17  C1  2Ø  2Ø  C1  AD  48               75Ø
CØCØ  Ø3  DØ  5Ø  2Ø  17  C1  8A  29               718
CØC8  FØ  8D  45  Ø3  4A  4A  4A  4A               749
CØDØ  CD  3F  Ø3  FØ  ØA  Ø9  FØ  CD               975
CØD8  21  DØ  DØ  ØA  2Ø  2Ø  C1  AD               889
CØEØ  48  Ø3  29  55  DØ  2D  AD  46               697
CØE8  Ø3  CD  3F  Ø3  FØ  ØF  Ø9  FØ               778
CØFØ  CD  21  DØ  DØ  ØF  AD  3F  Ø3               9Ø8
CØF8  ØD  45  Ø3  91  FD  AD  48  Ø3               731
```

Block #3

```
C1ØØ  29  AA  DØ  ØF  A9  94  18  65    876
C1Ø8  FE  85  FE  AD  3F  Ø3  91  FD   1278
C11Ø  AD  48  Ø3  8D  47  Ø3  6Ø  B1    736
C118  FD  AA  29  ØF  8D  46  Ø3  6Ø    789
C12Ø  AD  3F  Ø3  ØA  ØA  ØA  ØA  ØD    292
C128  46  Ø3  91  FD  6Ø  A2  4E  2C    851
C13Ø  A2  53  2C  A2  58  2C  A2  62    843
C138  2C  A2  67  AØ  Ø3  4C  D4  BB    947
C14Ø  A9  4E  2C  A9  53  2C  A9  67    859
C148  AØ  Ø3  4C  A2  BB  2Ø  39  C1    87Ø
C15Ø  2Ø  F7  B7  A6  14  A4  15  8E    975
C158  49  Ø3  8C  4A  Ø3  6Ø  2Ø  FD    674
C16Ø  AE  2Ø  EB  B7  8E  3E  Ø3  A6    997
C168  14  A4  15  8E  3C  Ø3  8C  3D    611
C17Ø  Ø3  C9  A4  FØ  1F  2Ø  F1  B7   1Ø95
C178  8E  3F  Ø3  2Ø  F1  B7  8E  44    874
```

Block #4

```
C18Ø  Ø3  2Ø  38  CØ  2Ø  9D  CØ  AD    837
C188  48  Ø3  49  FF  31  FB  ØD  47    787
C19Ø  Ø3  91  FB  6Ø  2Ø  73  ØØ  2Ø    674
C198  8A  AD  2Ø  ØF  BC  AC  3C  Ø3    781
C1AØ  AD  3D  Ø3  2Ø  91  B3  2Ø  2D    67Ø
C1A8  C1  2Ø  53  B8  2Ø  33  C1  46    838
C1BØ  66  2Ø  4D  C1  2Ø  F1  B7  8A    998
C1B8  A8  2Ø  A2  B3  2Ø  ØF  BC  AC    948
C1CØ  3E  Ø3  2Ø  A2  B3  2Ø  3Ø  C1    711
C1C8  2Ø  53  B8  2Ø  36  C1  46  66    75Ø
C1DØ  A9  67  AØ  Ø3  2Ø  5B  BC  3Ø    794
C1D8  ØB  2Ø  2B  BC  DØ  Ø3  4C  75    678
C1EØ  C1  2Ø  4D  C1  2Ø  46  C1  A9    959
C1E8  58  AØ  Ø3  2Ø  ØF  BB  2Ø  33    568
C1FØ  C1  2Ø  46  C1  A9  62  AØ  Ø3    918
C1F8  2Ø  ØF  BB  2Ø  36  C1  2Ø  F1    786
```

Block #5

```
C2ØØ  B7  8E  3F  Ø3  2Ø  F1  B7  8E    989
C2Ø8  44  Ø3  2Ø  81  C1  2Ø  4Ø  C1    714
C21Ø  A9  58  AØ  Ø3  2Ø  67  B8  2Ø    771
C218  2B  BC  3Ø  37  2Ø  2D  C1  2Ø    636
```

Block #5 (cont.)

```
C22Ø  F7  B7  A5  14  A6  15  8D  3C    1ØØ3
C228  Ø3  8E  3D  Ø3  2Ø  43  C1  A9     67Ø
C23Ø  62  AØ  Ø3  2Ø  67  B8  2Ø  2B     655
C238  BC  3Ø  18  2Ø  3Ø  C1  2Ø  F7     812
C24Ø  B7  A5  14  8D  3E  Ø3  CE  49     853
C248  Ø3  DØ  BF  CE  4A  Ø3  1Ø  BA     887
C25Ø  4C  81  C1  6Ø  2Ø  FD  AE  2Ø     985
C258  EB  B7  8E  3E  Ø3  A6  14  A4     975
C26Ø  15  8E  3C  Ø3  8C  3D  Ø3  2Ø     462
C268  F1  B7  8E  3F  Ø3  8A  Ø9  FØ    1Ø19
C27Ø  CD  21  DØ  DØ  Ø4  2Ø  F1  B7    1114
C278  6Ø  2Ø  F1  B7  8A  FØ  Ø8  AD    1111
```

Block #6

```
C28Ø  3C  Ø3  29  FE  8D  3C  Ø3  8E     7Ø4
C288  44  Ø3  E8  8E  4Ø  Ø3  AØ  ØØ     672
C29Ø  8C  43  Ø3  A9  ØØ  8D  42  Ø3     589
C298  8D  41  Ø3  AD  3C  Ø3  DØ  Ø5     658
C2AØ  CD  3D  Ø3  FØ  2Ø  38  ED  4Ø     898
C2A8  Ø3  8D  3C  Ø3  BØ  Ø3  CE  3D     653
C2BØ  Ø3  2Ø  43  C3  FØ  E5  18  AD     963
C2B8  3C  Ø3  6D  4Ø  Ø3  8D  3C  Ø3     443
C2CØ  9Ø  Ø3  EE  3D  Ø3  EE  3E  Ø3     752
C2C8  2Ø  43  C3  DØ  ØB  AD  41  Ø3     754
C2DØ  DØ  ØB  2Ø  4E  C3  A9  Ø1  2C     738
C2D8  A9  ØØ  8D  41  Ø3  CE  3E  Ø3     649
C2EØ  CE  3E  Ø3  2Ø  43  C3  DØ  ØB     784
C2E8  AD  42  Ø3  DØ  ØB  2Ø  4E  C3     766
C2FØ  A9  Ø1  2C  A9  ØØ  8D  42  Ø3     593
C2F8  EE  3E  Ø3  2Ø  81  C1  AD  3C     89Ø
```

Block #7

```
C3ØØ  Ø3  18  6D  4Ø  Ø3  8D  3C  Ø3     4Ø7
C3Ø8  AD  3D  Ø3  69  ØØ  8D  3D  Ø3     547
C31Ø  FØ  29  AD  3C  Ø3  C9  4Ø  9Ø     926
C318  22  AE  43  Ø3  FØ  2F  CA  BD     956
C32Ø  ØØ  CB  8D  3D  Ø3  BD  ØØ  CA     799
C328  8D  3C  Ø3  BD  ØØ  C9  8D  3E     797
C33Ø  Ø3  CE  43  Ø3  C9  C8  BØ  E1    1Ø81
C338  4C  93  C2  2Ø  43  C3  DØ  D9    1136
C34Ø  4C  C5  C2  2Ø  38  CØ  AØ  ØØ     9Ø7
```

```
C348  B1  FB  2D  48  Ø3  6Ø  AE  43     885
C35Ø  Ø3  AD  3E  Ø3  9D  ØØ  C9  AD     772
C358  3C  Ø3  9D  ØØ  CA  AD  3D  Ø3     659
C36Ø  9D  ØØ  CB  EE  43  Ø3  6Ø  2Ø     796
C368  F1  B7  8E  4B  Ø3  2Ø  F1  B7    11ØØ
C37Ø  8A  48  2Ø  F1  B7  8E  4C  Ø3     887
C378  2Ø  F1  B7  8E  5D  Ø3  2Ø  F1     967
```

Block #8

```
C38Ø  B7  8E  5E  Ø3  2Ø  F1  B7  8E    1Ø2Ø
C388  5F  Ø3  2Ø  F1  B7  68  FØ  1D     927
C39Ø  A9  7F  8D  ØD  DC  8E  Ø7  DC    1Ø39
C398  A9  C9  8D  14  Ø3  A9  C3  8D    1Ø39
C3AØ  15  Ø3  A2  11  AØ  82  8E  ØF     65Ø
C3A8  DC  8C  ØD  DC  6Ø  A9  7F  8D    1126
C3BØ  ØD  DC  A9  31  8D  14  Ø3  A9     784
C3B8  EA  8D  15  Ø3  A2  Ø8  AØ  81     858
C3CØ  DØ  E4  2Ø  A8  C4  AE  5D  Ø3    11Ø2
C3C8  FØ  53  AD  15  DØ  29  Ø4  DØ     978
C3DØ  2D  AD  6Ø  Ø3  29  1Ø  DØ  45     651
C3D8  AD  15  DØ  Ø9  Ø4  8D  15  DØ     785
C3EØ  AD  1Ø  DØ  29  FB  8D  1Ø  DØ    1Ø54
C3E8  29  Ø1  ØA  ØA  ØD  1Ø  DØ  8D     44Ø
C3FØ  1Ø  DØ  AD  ØØ  DØ  8D  Ø4  DØ     958
C3F8  AD  Ø1  DØ  8D  Ø5  DØ  8A  A2    1Ø36
```

Block #9

```
C4ØØ  Ø4  AØ  Ø5  2Ø  1A  C5  AD  1Ø     613
C4Ø8  DØ  29  Ø4  ØD  Ø4  DØ  FØ  Ø5     723
C41Ø  AD  Ø5  DØ  DØ  Ø8  AD  15  DØ    1ØØ4
C418  29  FB  8D  15  DØ  AD  6Ø  Ø3     934
C42Ø  A2  ØØ  AØ  Ø7  2Ø  1A  C5  AD     757
C428  4C  Ø3  FØ  62  AE  5E  Ø3  FØ     928
C43Ø  53  AD  15  DØ  29  Ø8  DØ  2D     787
C438  AD  61  Ø3  29  1Ø  DØ  45  AD     78Ø
C44Ø  15  DØ  Ø9  Ø8  8D  15  DØ  AD     789
C448  1Ø  DØ  29  F7  8D  1Ø  DØ  29     918
C45Ø  Ø2  ØA  ØA  ØD  1Ø  DØ  8D  1Ø     416
C458  DØ  AD  Ø2  DØ  8D  Ø6  DØ  AD    1119
C46Ø  Ø3  DØ  8D  Ø7  DØ  8A  A2  Ø6     873
C468  AØ  Ø4  2Ø  1A  C5  AD  1Ø  DØ     816
C47Ø  29  Ø8  ØD  Ø6  DØ  FØ  Ø5  AD     694
C478  Ø7  DØ  DØ  Ø8  AD  15  DØ  29     874
```

Block #10

```
---------
C48Ø  F7  8D  15  DØ  AD  61  Ø3  A2    1Ø52
C488  Ø2  AØ  Ø6  2Ø  1A  C5  AE  5F     692
C49Ø  Ø3  FØ  Ø8  8A  A2  Ø8  AØ  Ø3     722
C498  2Ø  1A  C5  AD  ØD  DC  29  Ø1     7Ø3
C4AØ  FØ  Ø3  4C  31  EA  4C  BC  FE    112Ø
C4A8  AD  4B  Ø3  DØ  ØD  AD  Ø1  DC     866
C4BØ  8D  6Ø  Ø3  AD  ØØ  DC  8D  61     871
C4B8  Ø3  6Ø  A9  ØØ  AA  A8  8D  ØØ     747
C4CØ  DC  AD  Ø1  DC  A9  FD  2Ø  F6    1314
C4C8  C4  A9  FB  2Ø  D9  C4  AØ  Ø1    1222
C4DØ  A2  Ø3  A9  BF  2Ø  F6  C4  A9    1168
C4D8  DF  8D  ØØ  DC  AD  Ø1  DC  Ø9     987
C4EØ  C9  A2  Ø7  DD  ØA  C5  FØ  Ø4    1Ø42
C4E8  CA  1Ø  F8  6Ø  BD  12  C5  39    1Ø23
C4FØ  6Ø  Ø3  99  6Ø  Ø3  6Ø  8D  ØØ     588
C4F8  DC  AD  Ø1  DC  3D  ØØ  CØ  DØ    1Ø75
```

BLOCK #11

```
---------
C5ØØ  Ø3  A9  ØF  2C  A9  1F  99  6Ø     68Ø
C5Ø8  Ø3  6Ø  FD  EF  FB  DF  F9  EB    1549
C51Ø  DD  CF  1E  1D  1B  17  1E  1D     596
C518  1E  1D  4A  BØ  Ø3  DE  Ø1  DØ     743
C52Ø  4A  BØ  Ø3  FE  Ø1  DØ  4A  BØ     966
C528  12  BD  ØØ  DØ  DØ  Ø9  AD  1Ø     821
C53Ø  DØ  59  ØØ  CØ  8D  1Ø  DØ  DE    1Ø76
C538  ØØ  DØ  6Ø  4A  BØ  ØE  FE  ØØ     822
C54Ø  DØ  DØ  Ø9  AD  1Ø  DØ  59  ØØ     911
C548  CØ  8D  1Ø  DØ  6Ø  ØØ  ØØ  ØØ     653
```

Taking a Break/Finishing Up

If your eyes grow weary, your throat dry and parched, and you want to take a break, *save* your work. We recommend saving your work at the end of every block you type. That way, should you make a mistake in the saving process, you can load the last saved version into memory and then re-type only one block. You must save your work after typing the final data block.

Saving machine language data is done in a special way, because you are not saving a "normal" file. Again, if you save the machine language incorrectly, you will probably lose anything that was not saved at an earlier time. Save your work carefully.

Cassette tape recorder users should prepare the recorder as when saving any other file, then move the cursor to a free line and very carefully type:

```
SYS57812"filename",1:POKE174,8Ø:POKE175,197:POKE193,Ø
:POKE194,192:SYS62954
```

filename can be replaced with any 16-character name you desire. You should use filenames like "ML.1", "ML.2", "ML.3", etc. each time you save the machine language at the end of a data block. The final name should be "M/L" (which stands for Machine Language), as that is how we will refer to it in this text.

Type the above as one command, typing *no* extra spaces between words and letters. The computer will wrap your typing around to the second line after the first one is filled. Just continue typing when this happens. When you are sure you have typed this line correctly, press **RETURN**.

Disk drive users should follow the same procedures, but type this command instead:

```
SYS57812"filename",8:POKE174,8Ø:POKE175,197:POKE193,Ø
:POKE194,192:SYS62954
```

Picking Up Where You Left Off

To get back to work, LOAD the "M/L" file back into the computer's memory. You can do this in the same manner as any other file, except you *must* attach ",1" to the end of the LOAD command. For disk drive users, this would be: LOAD "M/L",8,1. For cassette tape users, this would be: LOAD "M/L",1,1. Type NEW and press **RETURN** after the "M/L" file has been properly loaded.

> IMPORTANT: You *must* type NEW and press **RETURN** after loading the "M/L" file into memory with the keyboard. This will reset the computer's pointers, which direct it to the location of your BASIC program. You will later type a program that automatically loads the "M/L" file into memory. There is no need to reset the pointers from within a BASIC program. You only need to type NEW after loading the "M/L" file with the keyboard.

You should next check to make sure that the "M/L" file was properly saved, and thus is correctly loaded in memory. The only way to do this is to LOAD and RUN the HELPER program. Then, randomly check 5 to 10 rows of data you entered *before* you last saved "M/L". To check a row, type only the first number in the row (the memory location), and then press **RETURN**. The data for that row will be displayed, and can be compared to our printed data blocks.

When everything checks out, use the HELPER program to finish entering the data blocks.

The Final Check

Because it is essential that the machine language be entered and saved properly, we are providing one final check. The program shown below will look at the memory locations containing the "M/L" file and test each data block. Any block containing an error will be reported to you. You should then re-load the HELPER program and check each row in that block. Re-type any rows containing errors.

Type NEW and press **RETURN** to clear out any old program. Then, begin typing below:

```
1 I=49152 :R=1
2 T=0
3 FOR J=I TO I+127
4 T=T-PEEK(J)*(J<50509)
5 NEXT
6 READS: IFS<>T THEN PRINT"ERROR IN BLOCK"R:END
7 I=I+128:PRINT"BLOCK"R"IS OK":R=R+1:IFR>11 THEN END
8 GOTO 2
10 DATA 15494,13405,13567,12697,13656,11506,13622
11 DATA 14447,13025,15192,8817
```

When you are finished, save this program under "CHECKER", and then run it. This program looks at the machine language data blocks you have placed in memory with the HELPER program, and determines if each data block (as a whole) has been entered correctly. The message "BLOCK #X IS OK" should appear for Block #1 through Block #11. If the message "ERROR IN BLOCK #X" ever appears, you will need to check and re-type one or more rows in the block number given. (This assumes, of course, that the CHECKER program has no typing errors.)

Do not continue in this book until this final check has been met. When it has, save the "M/L" file in the manner discussed above.

Entering the First Tools

Now the fun begins. The first three subroutines are shown below. One subroutine turns graphics on, another turns graphics off, and still another sets the screen to a background color. Clear out any previous program from memory (type NEW and press **RETURN**), then type the subroutines:

```
1 GOTO 1000
10 REM::::::GRAPHICS
11 IF MU THEN POKE 53270,216
12 POKE 53265,59
13 POKE 53272,29
14 POKE 56576,198
15 RETURN
20 REM::::::TEXT
21 POKE 53270,200
22 POKE 53265,27
23 POKE 53272,21
24 POKE 56576,199
25 RETURN
30 REM::::::CLEAR HIRES/MULTI
31 SYS 49164,C
32 XL=0: XH=319: YL=0: YH=199
33 RETURN
```

To see if you entered them correctly, add these main routine lines that call each subroutine when a specific key is pressed:

```
1ØØØ REM::::::::::::::::::::::::::
1ØØ1 REM      MAIN ROUTINE
1ØØ2 REM::::::::::::::::::::::::::
1Ø1Ø PRINT "PRESS ANY KEY TO BEGIN:"
1Ø2Ø GET A$
1Ø3Ø IF A$="G" THEN GOSUB 1Ø
1Ø4Ø IF A$="T" THEN GOSUB 2Ø
1Ø5Ø IF A$="C" THEN C=INT(RND(1)*16): GOSUB 3Ø
1Ø9Ø GOTO 1Ø2Ø
```

Your new subroutines count on the "M/L" file being in memory. A convenient way to ensure this is to have the *program* look for the machine language in memory. If the machine language is not found, then the program should load the "M/L" file into memory. (Be sure that a disk/tape containing the "M/L" file is in your disk drive/cassette player when using this technique.)

Disk drive users should add this line to the program:

```
Ø IF PEEK(49152) <> 128 THEN LOAD "M/L",8,1
```

Cassette recorder users should add this line instead:

```
Ø IF PEEK(49152) <> 128 THEN LOAD "M/L",1,1
```

Notice that the file with the specific name of "M/L" will be loaded by this program line. Be sure to change this program line if you find you need to correct and re-save the "M/L" file under a different filename. The program line would remain the same, except that "M/L" would be replaced with the new filename.

RUN the program to see how you did.

> IMPORTANT: If the program does not operate as described below, the problem could either be in the program itself *or* in the "M/L" file. It might be that the most current version of your "M/L" file was saved incorrectly. The best approach is to stop this program and check it over carefully for typing errors. Correct any you find and try again. If you can find no typing errors, save this program under the filename "GRAPHICS". Then, LOAD and RUN the "CHECKER" program to test the "M/L" file in memory. If any data blocks are in error, you will need to run the "HELPER" program to correct them. Then, SAVE the corrected "M/L" file, re-load the "GRAPHICS" program (changing line 0 if you re-named the corrected "M/L" file), and try again (whew!).

The first thing your new program will do is check for the "M/L" file in memory. If it is not found, the program will load it. Cassette recorder users will need to press PLAY and C= as the need arises.

Once the "M/L" file is loaded, your program causes the message "PRESS ANY KEY TO BEGIN:" to appear at the bottom of your screen. Press G. You should immediately be placed in Graphics mode. If you screen is showing colors, either in what appears to be a pattern, or just rough, jagged lines, you are in graphics mode. If you don't get this pattern on your screen, press **RUN/STOP** and tap **RESTORE**. You will return to text mode and can check the subroutine at line 10 for errors.

Next, press **C** to see the screen colors change to one uniform color (the *background* color). We can't tell you what color your screen will display, because the program selects a *random* color each time **C** is pressed. The subroutine at line 30 sets the screen's background color to the one your program has randomly selected, and then changes all foreground pixels to background pixels.

> IMPORTANT: The background color of the graphics screen and the background color of the text screen are always the same. If you change the background of the text screen, the background color of the graphics screen changes to the same color. Likewise, if you change the background color of the graphics screen, the background color of the text screen changes. This is important because the Commodore will display your text screen *letters* in light blue, unless specifically directed otherwise. If you give the graphics screen a light blue background, the text screen will have a light blue background, and you will *not* be able to see your program listing when you return to text mode.

Always change the cursor to white (press **CTRL** and **2** at the same time) before running a program where light blue will be used as the background color. From that point on, all text letters will be displayed in white, which is easily visible on a lightblue background color.

For now, you can change the screen's background color by pressing **C** one or more times, until a color other than light blue is generated.

Finally, press **T** to return to Text mode. The text should again be displayed on the screen. (If not, press **RUN/STOP** and tap **RESTORE**. Check the subroutine at line 20.)

Your program contains an endless loop, and is thus still running. Press **RUN/STOP** to get out of the loop. Then, add the program lines below that provide a more graceful escape from the program. These lines allow you to return to text mode from the program any time you press ← (the top, left key on your keyboard).

```
1Ø4Ø IF A$ = "←" THEN GOSUB 2Ø: END
```

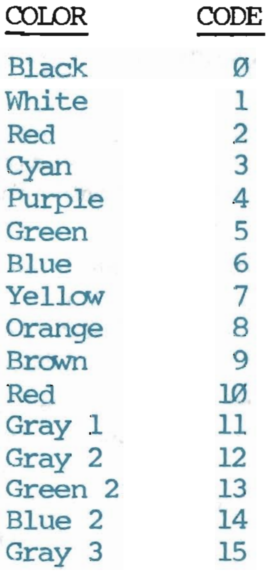
Practice with your program if you like. Pressing **C** will continually generate a new background color for the screen (bearing in mind that the same color can be randomly generated twice in a row). Pressing **G** still takes you to graphics mode, but you must press ← now to get back to text mode and END the program.

When you are done, press ← to return to text mode.

Let's briefly go over the program. Line 0 loads "M/L", in the event it hasn't been already. Line 1 bypasses the subroutines, sending the computer directly to your main routine. Lines 1010 through 2060 form an endless loop that examines keys being pressed on the keyboard. When a keypress of **G**, ← or **C** is observed, the appropriate subroutine is called.

The subroutine at line 10 takes you into the graphics mode. This is usually one of the first subroutines called in a picture-drawing program. The subroutine at line 20 returns you to text mode, and is usually called based on a certain condition being met (such as a keypress of ← occurring). The last subroutine at line 30 clears any image off of the graphics screen, meanwhile setting all pixels to a common background color. This color is determined by the variable C. In addition, line 32 sets the values of four variables (XL, XH, YL, and YH). These variables need to be set in preparation for tools coming in later chapters, and will be explained in detail at that time.

In your program, C is set to a random number between 0 and 15 each time line 1050 is gone through. Coincidentally, there are 16 color codes, numbered 0 through 15, from which to choose. The color codes and their corresponding colors are shown below.



COLOR	CODE
Black	Ø
White	1
Red	2
Cyan	3
Purple	4
Green	5
Blue	6
Yellow	7
Orange	8
Brown	9
Red	1Ø
Gray 1	11
Gray 2	12
Green 2	13
Blue 2	14
Gray 3	15

(This list is also provided in the appendices for quick reference.)

The colors white and light grey cannot be used as the background color on some monitors. This is because of the strong contrast they create with the other available colors. Foreground images (those images and shapes you draw on top of the background color) will be drawn in black when the contrast is too great for the monitor to handle.

You will also find that some lines are plotted in a rainbow of colors, and that some colors do not remain constant from day to day. Usually, adjusting your monitor's color, tint, and contrast will help bring in a sharper picture.

Your three new subroutines are the foundation on which you will build your tool kit. Below are three tool boxes, one for each of the subroutines. Take the time to read each "What it Does" and "Example Use" discussion. "Technical Descriptions" are provided for those readers who want more technical information on a subroutine. These can be passed over if you wish.

TOOL 10 :::::: GRAPHICS

```
10 REM::::::GRAPHICS
11 IF MU THEN POKE 53270,216
12 POKE 53265,59
13 POKE 53272,29
14 POKE 56576,198
15 RETURN
```

What It Does: This tool turns on the graphics mode, allowing you to create and display artwork on your Commodore 64. Although not yet discussed, setting the variable MU to 1 (MU=1) before calling this tool will produce multi-color graphics. Re-setting MU back to 0 will return you to high resolution graphics.

Example Use: To use this tool, you only need a GOSUB 10 statement in your main routine. If the sole purpose of your program is to draw a picture, this GOSUB statement should be one of the first statements in your program.

Technical Description: This subroutine changes the memory locations in the Commodore's VIC II Chip. This chip contains the set of memory locations that control what is viewed on your monitor.

Memory location 53270 controls the multi-color graphics capability of the computer. If the variable MU has been set to 1, then line 11 will POKE the value 216 into this memory location. This enables multicolor. If MU is set to 0, this POKE statement will be skipped over.

Memory location 53265 controls whether the computer displays text or graphics. Code 59, which is being POKEd in this location, has the computer display graphics on the screen.

Memory location 53272 controls where the graphics "pixel patterns" are stored within a bank. (Pixel patterns specify which pixels are foreground color and which pixels are background colored.) POKEing code 29 into this location makes sure the pixel patterns are not placed in the same memory locations used to store the graphics colors.

Memory location 56576 controls which bank the computer uses to store and retrieve graphics pixel patterns and color codes. The Commodore has four banks, each of which have 16K of memory. POKEing 198 into memory location 56576 tells the computer to use bank 1 to store and retrieve graphics information. This is necessary to keep other information, like a program listing, from interfering with your picture codes.

TOOL 20 :::::: TEXT

```
20 REM::::::TEXT
21 POKE 53270,200
22 POKE 53265,27
23 POKE 53272,21
24 POKE 56576,199
25 RETURN
```

What It Does: This tool turns off the graphics mode, taking you back to regular text.

Example Use: To use this tool, you need a GOSUB 20 statement in your program. For controlled use, you should establish a condition that must be met before this subroutine is called. In our program, the condition is that the ← key be pressed.

Technical Description: This subroutine restores the memory locations which were changed by Tool 10 (GRAPHICS). Doing this will return you to text mode. Notice that the same 4 memory locations (53270, 53265, 53272, and 56576) are used by each tool. For a complete explanation of the purpose of these memory locations, refer to the Technical Description for Tool 10.

TOOL 30 :::::: CLEAR HIRES/MULTI

```
30 REM::::::CLEAR HIRES/MULTI
31 SYS 49164,C
32 XL=Ø: XH=319: YL=Ø: YH=199
33 RETURN
```

What It Does: This tool clears the graphics screen, whether you are working in high resolution or multi-color (not discussed yet). All images are cleared off, and the entire screen is set to the color indicated by C's current value.

In addition, four variables get set. XL (X-Low) is set to 0, XH (X-High) is set to 319, YL (Y-Low) is set to 0, and YH (Y-High) is set to 199. These parameters describe your screen's left, right, top, and bottom boundaries. Later, these variables will be used by a CLIP A SHAPE tool to determine the rectangular screen area where plotting is allowed.

Example Use: To use this tool, you need both a GOSUB 30 statement in your main routine and a line that specifies C's intended value. C's value determines the color setting of your screen. See the color chart listed earlier in this chapter for the list of available colors and their corresponding color codes.

Technical Description: This subroutine calls a machine language tool to clear and set the graphics screen to a solid color. Memory locations 24576-32575 control the pixel patterns displayed on the graphics screen. The machine language will POKE a 0 into each of these memory locations, setting zero pixels to the foreground color. This, by default, changes all pixels to the background color.

Memory locations 17408-18407 control the foreground and background colors of the high-resolution screen. Memory location 53281 controls the background color of the multi-color screen. This subroutine POKEs C's current value into all of these memory locations (17408-18407, and 53281), setting the background color of both graphics screens. NOTE: The text screen uses memory location 53281 to store its background color also. Thus, the background color of the text screen is automatically changed to C's current value whenever this tool is called.

This tool was done in machine language beacuse BASIC is painfully slow at performing this task. See the appendices for a commented listing of the machine language program.

Plotting Points and Lines

These next tools put you into the real world of graphics. The ability to plot points and lines is, of course, invaluable to anyone trying to draw using the Commodore. Their importance, however, goes further. You will later discover that each advanced tool depends on the ability to call on these basic tools.

The PLOT A POINT and PLOT A LINE tools follow. Type them into your program now.

```
4Ø REM::::::PLOT A POINT
41 SYS 495Ø2,X,Y,C,MU
42 RETURN
5Ø REM::::::PLOT A LINE
51 SYS 495Ø2,X1,Y1 TO X2,Y2,C,MU
52 RETURN
```

These tools are short enough that they can be easily double-checked. But the real test lies in using them. The PLOT A POINT tool requires an X (column) and Y (row) location of the point to plot. In addition, C's value must be changed to a new color, so that the plotted point can be seen. Enter these main routine lines:

```
1Ø6Ø C=Ø
1Ø7Ø IF A$="P" THEN X=RND(1)*32Ø:Y=RND(1)*2ØØ: GOSUB 4Ø
```

Line 1060 resets the color variable to 0 (black). This is so the test points and lines will be visible when plotted. Line 1070 will randomly plot a point on your screen each time **P** is pressed. The screen has 320 columns, from X=0 to X=319. This line sets the X coordinate to a random value *between* 0 and 320. Your screen also contains 200 rows, from Y=0 to Y=199. This line sets the Y coordinate to a random value *between* 0 and 200.

To test the PLOT A LINE subroutine, add this line to your program:

```
1Ø8Ø IF A$="L" THEN X1=16Ø:Y1=99:X2=RND(1)*32Ø:Y2=RND
     (1)*2ØØ:GOSUB 5Ø
```

When specifying a line to plot, you must indicate the two points that should be connected by the line. This is done by using X1,Y1 and X2,Y2 notations. Line 1080 sets the X1,Y1 point at 160,99, but randomly selects a X2,Y2 position. The result will be an interesting selection of plotted lines, all stemming from the center point of your screen.

RUN the program to begin the test.

The program has not changed significantly—just a few enhancements tacked on to the end. Press **G** to view the graphics screen. If necessary, press **C** to set the screen's background color. You want to make sure that the background color is light enough to see the points and lines as you plot them. If your screen is set to a dark color, keep pressing **C** until a light color appears.

Test the PLOT A POINT subroutine by pressing **P**. Lean forward, squint, adjust your spectacles, and you should see a tiny plotted point. Press **P** several times, watching as random points get plotted around the screen.

Test the PLOT A LINE subroutine by pressing **L**. Beginning at the center area of your screen, a line will be plotted outward. This line may be of any length, and may go in any direction. Continue to press **L**, and watch as a starburst begins to develop. Pay particular attention to the speed with which the lines are plotted. The DRAW A LINE tool is using the "M/L" data you typed earlier to achieve this high speed. (If the program does not react each time you press **P** or **L**, keep in mind that you might be plotting over the top of a previously plotted point or line.)

If you press **C**, the screen will clear, and a new background color will appear. Try a starburst with the new color.

That completes the new subroutine tests. Press **C** until a color other than light blue is generated for the background. Then press ← to return to text mode. If you had any problems, check your new main routine lines, and the new subroutines.

You will need to save these tools before continuing. Save them under the filename "GRAPHICS1".

The next section gives you a ZAP routine. Its function will be to erase (*ZAP!*) your main routine upon command. This enables you to keep all subroutine tools in memory and begin drawing a new picture at any time.

TOOL 40::::::PLOT A POINT

```
4Ø REM::::::PLOT A POINT
41 SYS 495Ø2,X,Y,C,MU
42 RETURN
```

What It Does: This tool plots a point on the graphics screen. The point will be plotted at the last X,Y location given in the program. If no values for X and Y are assigned in the program, then the point will be plotted at 0,0. The color used to plot the point will be determined by C's current value.

Example Use: You must set three variables before calling this tool with a GOSUB 40 statement. C must be set to the desired color code. X and Y must be set to the vertical and horizontal locations of the point to plot. Example program lines for plotting a single point are:

```
1060 C=2
1070 X=10: Y=50: GOSUB 40
```

Technical Description: The PLOT A POINT subroutine was done in machine language in order to speed up the process. This same machine language routine is also used by the PLOT A LINE and PAINT A SHAPE subroutines. This section of machine language is the main section affected by the use of multi-color. If MU = 1 (Multi-color is enabled), then special operations must be performed to deal with the double width pixels and the extra color capability. The PLOT A LINE and PAINT A SHAPE tools will essentially ignore MU and let the PLOT A POINT tool handle this extra work.

How Plot A Point Handles Multi-Color

When Tool 10 turns on Multi-Color (MU = 1), the computer uses a 160 x 200 resolution screen. The high resolution screen is 320 x 200 pixels. This means that the pixels are twice as wide in Multi-Color as they are in high resolution. In other words, the pixels are two bits wide. A two bit pixel has four possible states:

(1) 0 0
(2) 0 1
(3) 1 0
(4) 1 1

Remember, we cannot see the two bits in Multi-Color. We see only one pixel. The point of having four possible states is to allow more colors:

(1) 0 0 is background color
(2) 0 1 is foreground # 1
(3) 1 0 is foreground # 2
(4) 1 1 is foreground # 3

When the graphics screen is cleared, all the pixels are changed to the specified background color. However, not only will your graphics screen be changed to the specified background color, your text screen will be changed as well. Thus, if you changed your graphics background to black, then your text background will also be black when you return to text mode.

All 1,000 color blocks on the graphics screen must share the same background color. However, the three foreground color "slots" (0 1, 1 0, and 1 1) and the colors associated with those "slots," can change from color block to color block. Thus, if you desire to plot a "red" pixel point against a light-blue background, and no other plotted pixel occurs within that color block, then "red" will be automatically assigned to the first foreground color position (i.e., 0 1 will be POKEd into the proper bit pair for the given X,Y position). If, however, you have previously plotted "green" pixels within that color block, then "red" will be assigned to the second foreground color position (i.e., 1 0). If both "green" and "white" had been previously plotted within that color block, then "red" will be assigned to the third foreground color position (i.e., 1 1). If *three* colors had been previously plotted within the color block before you plotted your "red" point, then "red" would automatically "bump" the color which previously appeared in the third foreground color position, or 1 1, and all pixels which had been previously plotted in that color would be changed to "red."

A commented listing of the machine language used by this book can be found in the appendices.

TOOL 50 :::::: PLOT A LINE

```
5Ø REM::::::PLOT A LINE
51 SYS 495Ø2,X1,Y1 TO X2,Y2,C,MU
52 RETURN
```

What It Does: This tool plots a line from a given X1,Y1 point to a given X2,Y2 point. This line is plotted in the color represented by C's current value.

Example Use: Five variables must be set before calling this tool with a GOSUB 50. C must be set to the desired plotting color. X1 and Y1 must be set to the vertical and horizontal locations of the starting point for the line to plot. X2 and Y2 must be set to the vertical and horizontal locations of the ending point for the line to plot. Example program lines that plot a line are:

```
1Ø8Ø C=7
1Ø9Ø X1=1Ø: Y1=5Ø
11ØØ X2=1ØØ: Y2=5Ø: GOSUB 5Ø
```

Technical Description: Tool 50 was also written in machine language because it so slow in BASIC. Yet, even our machine language version is not as fast as it could be because we wanted to minimize the typing required at the start of this book.

There is a point concerning Multi-Color which we must mention here. The Multi-Color resolution is 160 x 200 pixels, while high resolution is 320 x 200 pixels. Normally, we design our pictures on a 320 x 200 grid. However, if we used such a grid to design Multi-color pictures, all pixels from X = 160 to X = 319 would be lost. This means that we would have to completely redesign our shape in order to work in Multi-Color.

Tool 50 resolves this dilemma. By setting MU = to 1, we have indicated that we are working in Multi-Color. Tool 50 will then automatically divide all of the program's X coordinates by two. Thus X = 319 becomes X = 159.5. When our point is plotted, this "X" coordinate will be automatically rounded down to "159", and our problem is solved.

Notice that this "rounding down" method affects our picture, but only to a very slight degree. If we try to plot X = 150 and X = 151 (150/2 = 75, and 151/2 = 75.5), then only X = 75 will be plotted. This causes only a slight variance, however, and your shapes should still be accurately drawn.

The Zap Routine

The ZAP routine is longer than the subroutines you have entered so far, and must be typed with the utmost care. Many things can go wrong if this routine is entered incorrectly and then run. Be especially careful of line 172 (A=256: B=2049: C=1003). Make sure you set C equal to 1003 and no other number. As you type, each line too long to fit across your screen will automatically wrap around to the next line for your convenience.

```
170 REM::::::ZAP!
171 GOSUB 2Ø:PRINT"DO YOU KNOW WHAT YOU ARE DOING?":END
172 A=256: B=2Ø49: C=1ØØ3
173 IF PEEK(B+2)+A*PEEK(B+3)>=C THEN 176
174 B=PEEK(B)+A*PEEK(B+1):ON ABS(B<>Ø) GOTO 173:END
175 A=256: B=PEEK(251)+A*PEEK(252)
176 IF PEEK(B+1)=Ø THEN END
177 PRINT CHR$(147) PEEK(B+2)+A*PEEK(B+3): PRINT"GOTO 175"
178 POKE 251, B-INT(B/A)*A: POKE 252,B/A
179 POKE 631,19: POKE 632,13: POKE 633,13: POKE 198,3: END
```

After typing and carefully re-checking each line in this routine, you should test it. If you have not yet saved your program, now is be an extremely opportune time to do so.

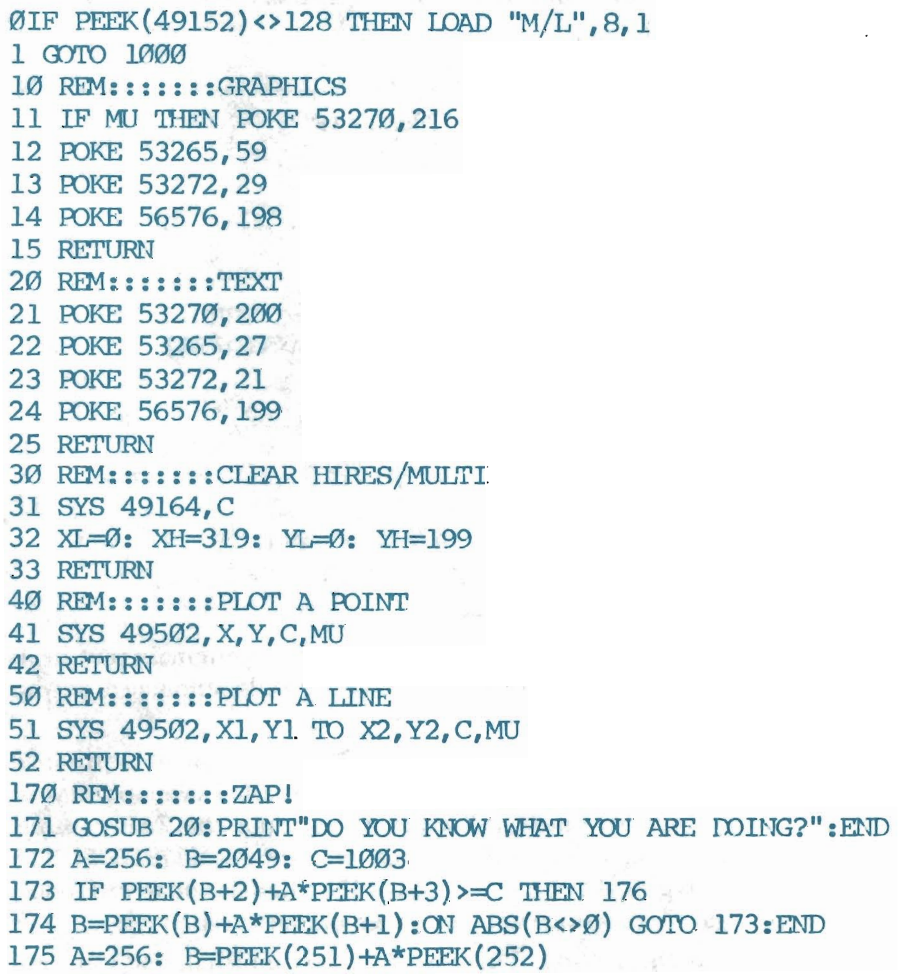
The ZAP routine is meant to be executed by *itself* from outside the program. It should never be called from within the main routine, as its purpose is to erase the main routine. Line 171 helps prevent an accidental erasure of your main routine. If this tool is ever called by a GOSUB 170 statement, line 171 will return you to text mode, print "DO YOU KNOW WHAT YOU ARE DOING?" on the screen, and end the program. We did this because it is far too easy to type GOSUB 170 in the main routine, when perhaps only GOSUB 70 was intended.

Typing RUN 172 outside the program will execute this routine properly (skipping over line 171 entirely). The ZAP routine will then delete your main routine lines, beginning at line 1003. Line 179 contains an END statement, which stops the computer from executing any lines past that one. Test the ZAP routine by moving the cursor to a free, blank screen line and typing:

```
RUN 172
```

First, the screen should clear. Then, in the top left-hand corner you will see several flashing items. At the top, the line numbers you are deleting will flash by. Beneath this, the words "GOTO 175" and "READY." will flash on and off. When the computer has completed the ZAP request, the blinking cursor will re-appear.

LIST your program to see what happened. You should find that all of your subroutines still remain. What will be missing is the main routine (lines 1003 and higher). Compare your program to that shown below to make sure you didn't zap any necessary tools.



```
ØIF PEEK(49152)<>128 THEN LOAD "M/L",8,1
1 GOTO 1ØØØ
1Ø REM::::::GRAPHICS
11 IF MU THEN POKE 5327Ø,216
12 POKE 53265,59
13 POKE 53272,29
14 POKE 56576,198
15 RETURN
2Ø REM::::::TEXT
21 POKE 5327Ø,2ØØ
22 POKE 53265,27
23 POKE 53272,21
24 POKE 56576,199
25 RETURN
3Ø REM::::::CLEAR HIRES/MULTI
31 SYS 49164,C
32 XL=Ø: XH=319: YL=Ø: YH=199
33 RETURN
4Ø REM::::::PLOT A POINT
41 SYS 495Ø2,X,Y,C,MU
42 RETURN
5Ø REM::::::PLOT A LINE
51 SYS 495Ø2,X1,Y1 TO X2,Y2,C,MU
52 RETURN
17Ø REM::::::ZAP!
171 GOSUB 2Ø:PRINT"DO YOU KNOW WHAT YOU ARE DOING?":END
172 A=256: B=2Ø49: C=1ØØ3
173 IF PEEK(B+2)+A*PEEK(B+3)>=C THEN 176
174 B=PEEK(B)+A*PEEK(B+1):ON ABS(B<>Ø) GOTO 173:END
175 A=256: B=PEEK(251)+A*PEEK(252)
```

```
176 IF PEEK(B+1)=Ø THEN END
177 PRINT CHR$(147) PEEK(B+2)+A*PEEK(B+3): PRINT"GOTO 175"
178 POKE 251, B-INT(B/A)*A: POKE 252,B/A
179 POKE 631,19: POKE 632,13: POKE 633,13: POKE 198,3: END
1ØØØ REM::::::::::::::::::::::::::
1ØØ1 REM      MAIN ROUTINE
1ØØ2 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 that have mistakes, as well as any other subroutine lines that were erroneously erased. If all else fails, load the GRAPHICS1 file into memory, and re-enter the ZAP routine. You must test this routine and correct any errors before continuing to next chapter.

This tool is discussed in greater detail in the tool box below. It is only important for you to understand why this tool is of benefit (it can quickly erase any main routine of any size, leaving all subroutines in memory), and how to use it (type RUN 172 **RETURN**).

TOOL 170::::::ZAP!

```
170 REM::::::ZAP!
171 GOSUB 2Ø:PRINT"DO YOU KNOW WHAT YOU ARE
    DOING?":END
172 A=256: B=2Ø49: C=1ØØ3
173 IF PEEK(B+2)+A*PEEK(B+3)>=C THEN 176
174 B=PEEK(B)+A*PEEK(B+1):ON ABS(B<>Ø)
    GOTO 173:END
175 A=256: B=PEEK(251)+A*PEEK(252)
176 IF PEEK(B+1)=Ø THEN END
177 PRINT CHR$(147) PEEK(B+2)+A*PEEK(B+3):
    PRINT"GOTO 175"
178 POKE 251, B-INT(B/A)*A: POKE 252,B/A
179 POKE 631,19: POKE 632,13: POKE 633,13:
    POKE 198,3: END
```

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

Example Use: You need to have a GOTO statement positioned somewhere before the ZAP routine begins. This allows you to run your program whenever necessary, without executing the ZAP routine. When you *do* want to erase the main routine, type RUN 172 **RETURN**. Mistakenly typing RUN 170 or GOSUB 170 will take you to text mode, where a warning message will be printed on the screen.

Technical Description: This routine is very unusual, and too complex to explain here. Its value outweighed omitting it entirely, 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 press **RETURN**. If there are many lines to be deleted (as there will soon be in your main routine), this could take some time to do.

One alternative is to use 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? This is exactly what happens when you use the ZAP routine. Note, however that this routine is actually a small program within your larger program. That is why it is executed by typing RUN.

Summary

You have successfully completed the "setup" portion of this book. Using your current collection of tools, you can display an assortment of drawings and designs on the screen. Stick figures, geometric art, and dot designs can be easily created. Best of all, you are prepared for advanced material.

Some key points to remember about your new tools are:

—C determines the color of everything. Whenever you need to work with a new color, change C's value.

—Any time you specify a point or a line to be plotted, you must also have a GOSUB statement.

—Use the ZAP routine with caution. As a general rule, save your program before running the ZAP routine. You may be very glad you did.

SAVE your set of tools under the filename "CHAPTER 1". Then, spend some time practicing with them. As an incentive, and to start your ideas flowing, we have provided the exercise program below. This program is a little too complex to explain at the end of a chapter. However, since you have worked hard to get here, we thought it would be good to give you an exercise that produced something slightly complex, colorful, and vibrant. If you have a good understanding of FOR/NEXT loops, and the X,Y relationship, you may be able to follow the program easy enough.

Run the ZAP routine (by typing RUN 172 **RETURN**), and then begin typing:

```
2ØØØ GOSUB 1Ø: C=15: GOSUB3Ø
2Ø1Ø FOR K = Ø TO 99 STEP 3
2Ø2Ø X1=Ø: Y1=Ø
2Ø3Ø Y2=K: X2=16Ø-K*1.6: C=2: GOSUB5Ø
2Ø4Ø Y1=199
3ØØØ Y2=199-Y2: C=5: GOSUB5Ø
```

```
3Ø1Ø X1=319
3Ø2Ø X2=319-X2: C=8: GOSUB5Ø
3Ø3Ø Y1=Ø
3Ø4Ø Y2=199-Y2: C=6: GOSUB5Ø
4ØØØ NEXT K
5ØØØ X1=159: Y1=1Ø
5Ø1Ø X2=3Ø5: Y2=1ØØ: C=1Ø: GOSUB5Ø
5Ø2Ø X2=319-X2: GOSUB5Ø
5Ø3Ø Y1=199-Y1: GOSUB5Ø
5Ø4Ø X2=319-X2: GOSUB5Ø
5Ø5Ø FOR P = Ø TO 3ØØØ: NEXT P
6ØØØ RUN
```

This program will draw the design shown earlier in this chapter—the one taking only 41 seconds to plot with machine language. Refer to the picture on page 12 to see if your program is running properly.

You can quickly modify this program in several ways to change the design. You can increase/decrease the STEP value of K in line 2010 (e.g., STEP 2). You can change the various color codes assigned to C. You can even change the retangular centerpiece to any other design that comes to mind. Have fun!

Chapter Two

WORKING WITH SHAPES

Shapes are the essence of all computer art. Whether you are drawing pictures, plotting graphs, creating games, or simply trying your hand at this unique form of artwork, you need to work with shapes. In this chapter you will learn how to define, retrieve, clip, and draw shapes. In addition, you will learn a little about multi-color.

A technique involving DATA statements is used to define a shape. If you have defined shapes with DATA statements before, you may be tempted to skip over portions of this chapter. Please don't. With few exceptions, the steps to defining a shape will be new to you. At a minimum, read the "What It Does" and "Example Use" sections of each tool box.

If you aren't familiar with defining shapes, you may be wondering exactly what we mean. That is, since you can already PLOT A POINT and PLOT A LINE, you have the necessary tools to draw any shape that you may require. However, defining a shape is a little different than plotting it point-by-point or line-by-line. When you define a shape, you describe it as a *single, whole object.* A shape described as one object can be moved, copied, and rotated with a minimum of effort. If, instead, you enter a shape as a series of plotted points and lines, each point and line must be dealt with individually whenever a global change to the shape is necessary (for example, rotating the entire shape). This can become a very tedious task.

Think of plotting a complex shape, like a brick building, line-by-line. If you later wanted to move this shape to a new screen position, you might be facing hours of work. Now consider if this same shape could be handled like a single plotted line. Adjustments would revert from hours to only minutes.

Once you define a shape, it can be retrieved each time you want to work with it. This is done using a RETRIEVE A SHAPE tool, which reads the shape's description into memory. More specifically, this tool will search through an entire list of shape descriptions, retrieving only the shape you choose. Thus, you can have a "library" of various shapes, each of which can be easily searched for and re-used within the same picture.

Another tool that will be introduced in this chapter is CLIP A SHAPE. This tool looks at the shape to draw, and determines if any part of it falls outside of the X and Y coordinate ranges. If so, those parts are "clipped" off. This helps ensure that all plotting remains within the visible screen area. Accidentally plotting off the screen can cause the computer to "freeze up." When you begin working with translation, scaling and rotation, you won't always know where a shape is going to end up on (or off) the screen. This clipping tool will become very valuable to you then.

After defining, retrieving and clipping your shape, the DRAW A SHAPE tool will place it on the screen.

Load the "CHAPTER 1" program into memory, get out your graph paper, and begin.

Defining Shapes

The very first step to defining a shape is *picking* a shape. This is not always easy; but until you've decided on the object of your attention, you can't go much further.

Look around you. What should you draw? Suppose you start with something familiar—say, something within eyesight. Still, what will you choose? A good idea is to walk around the room, observing everything at different angles. Search for small, isolated areas that interest you. Are you in your bedroom? Try the chair with the shirt hanging on the back. At the office? What about the floor lamp and the vinyl recliner next to it?

Another good idea is to look through books and magazines for designs. When you find one you like, trace it. As long as you don't sell, publish or otherwise publicize it, you can trace and use any design you like. This method is not uncreative, either, as you still have the freedom to choose the colors used in each section of the design. Some professionals feel that tracing helps you gain an understanding of how other artists use lines, angles and curves to create successful images. This may or may not be true. It doesn't really matter; tracing and plotting a nice design can be great fun.

Finally, be sure to choose a design that appeals to you, remembering not to take on too much. You can always add to the design later.

When you've identified what you want to draw, sketch it on graph paper. This should be in a rectangular area that is forty blocks across and twenty-five blocks high (corresponding to the screen's color blocks). You might try lightly pencil-sketching the design first. That way, you can erase and rearrange the shapes until you're completely satisfied with the placement of each.

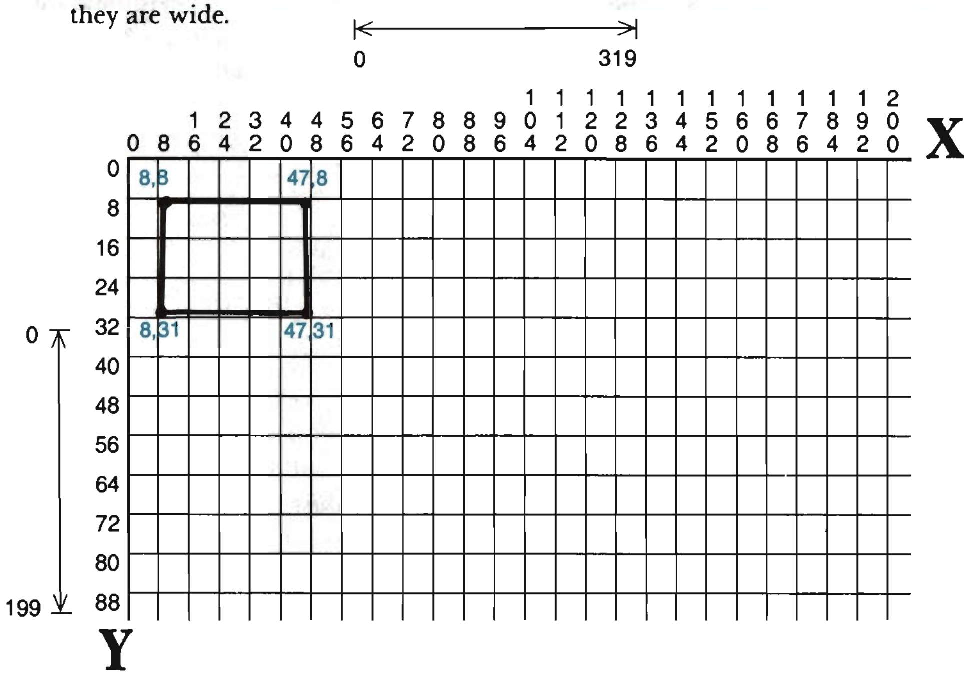
Another approach is to loosely tear the shapes out of construction paper—nothing fancy, just the general form of each object. Next, try arranging them on paper: chair in front of bed; chair beside bed; chair on top of bed; bed on top of chair, etc. When you find an arrangement that you like, draw it on your graph paper.

When the design is complete, you are ready to define and describe the first shape. Briefly, the shape's description will be read into memory and placed in two "arrays," or computer lists. One list will describe the shape's points, and the other list will describe the shape's lines. In order to do this, you must gather the information for these two lists. Because your design is sketched on graph paper, this can be done very easily.

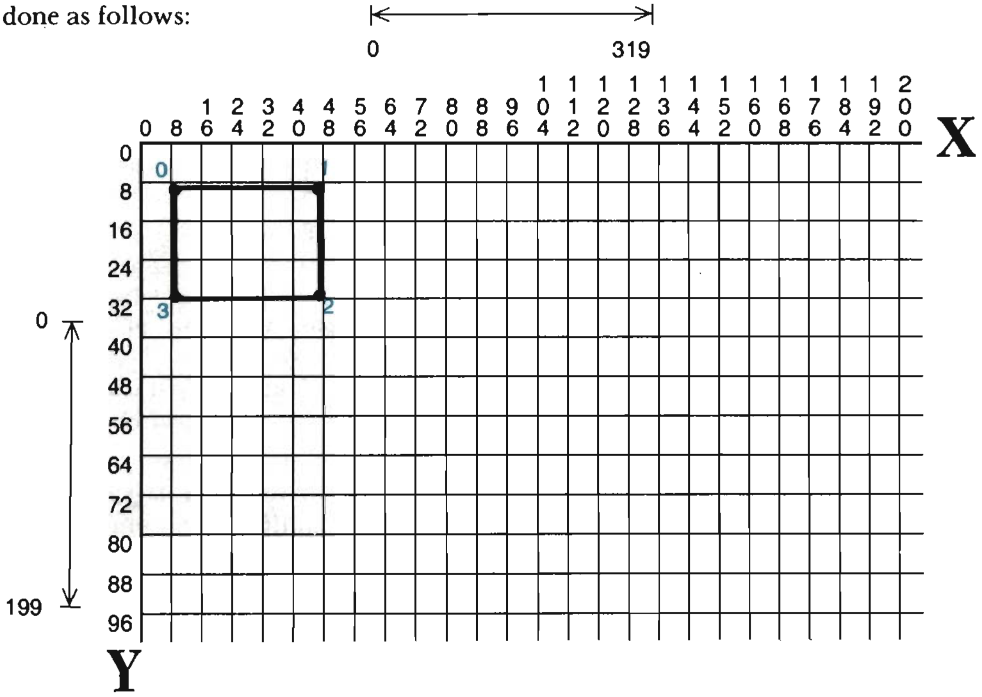
We will use a rectangle as our example shape. Sketched on graph paper, this shape might look like that shown below. Notice that for each endpoint (starting/ending point of a line) we have clearly noted the X,Y coordinates.

> *Note:* Throughout this book we will use graph paper examples similar to the one that follows. Each square represents an 8 x 8 pixel block on your screen. The numbers across the top represent the X location of every 8th pixel column. The numbers down the side indicate the Y location of every 8th pixel row. Marking your graph paper in this manner makes it easy to approximate the X,Y location of any point. Notice that this grid is *elongated* to reflect the

elongated screen pixels. All pixels are approximately 1.234 times higher than they are wide.



After sketching the shape, you assign a number to each of its endpoints. It doesn't matter which endpoint is assigned which number. All that matters is that you start with the number 0 (zero) and do *not* skip numbers. For the rectangle, this could be done as follows:



Now you can gather the lists. On the first list, write down the X,Y coordinates for each endpoint in the shape. This is done down columns: one column for X coordinates, and one column for Y coordinates. Each row holds the X,Y coordinates of one endpoint. Again using the rectangle as our example, this point list would look like this:

Point Data

	X	Y
Endpt #0	8	8
Endpt #1	47	8
Endpt #2	47	31
Endpt #3	8	31

Notice that the endpoints are listed in ascending order. That is, Endpoint #0 and its coordinates are on the first row, Endpoint #1 and its coordinates are on the second row, and so on. This is very important, as you will later see. The endpoints *must* be listed from Endpoint #0 on up, and each *must* be listed in X,Y form (X location before Y location).

That completes the information needed for the first list, which is called your "point data" list. In complex shapes with many endpoints, this list could become long. Always double-check your shape to make sure each endpoint has been duly recorded on the list before continuing.

The point data list only describes your shape's endpoints, and can be thought of as an empty "connect-the-dots" board. You also need a list that describes how those endpoints are connected together to form the shape. You need a "line data" list, set up in the following manner:

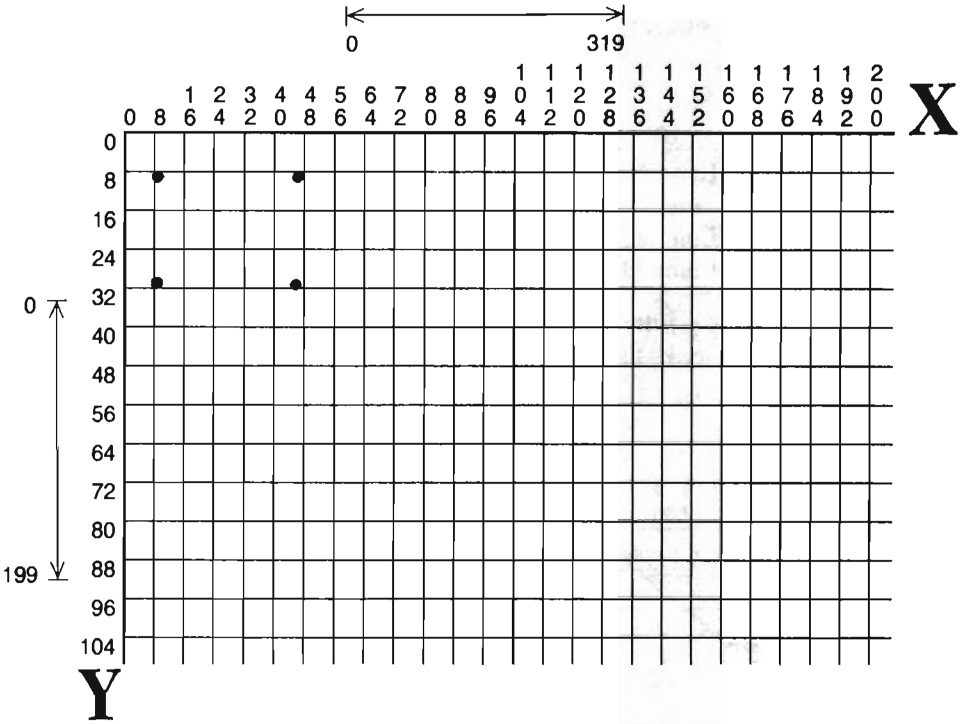
Line Data

	"FROM"	"TO"
Line #0	0	1
Line #1	1	2
Line #2	2	3
Line #3	3	0

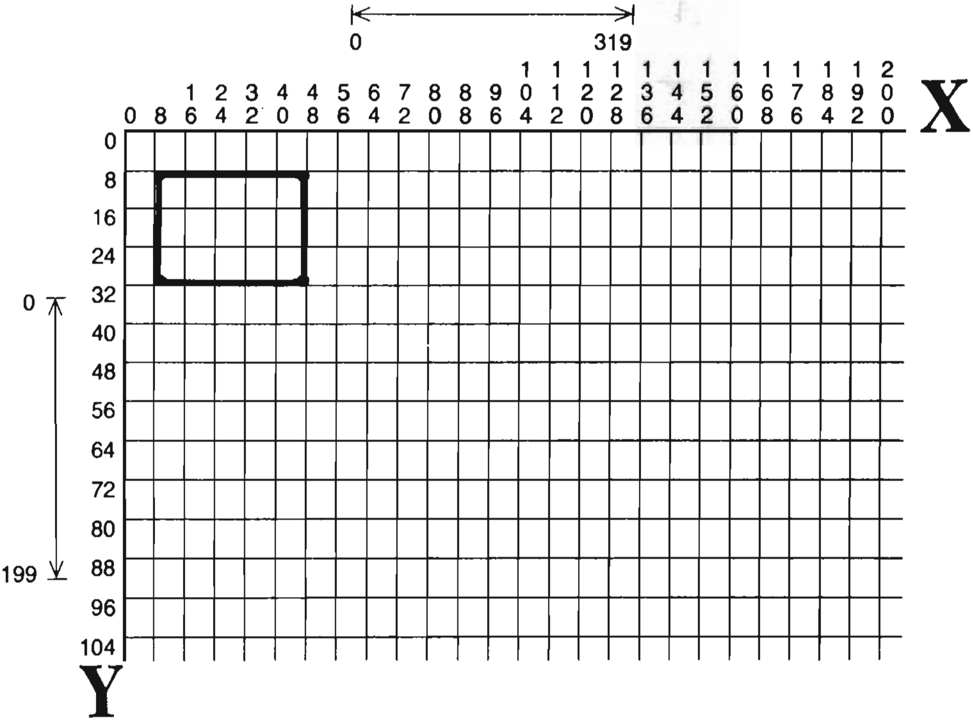
All lines are plotted "from" one endpoint "to" another. All you need to do is record the "from" and "to" endpoint numbers for each line in the shape you are drawing. By looking at the endpoint numbers assigned on your first list, you can easily record the endpoint numbers used by each line in the shape. This is how the line data list above was compiled.

Notice that each line was assigned a number. These numbers must start at 0 for the first line listed, and increase by one for each additional line. That completes the information needed on the line data list.

Let's see how the computer can use these two lists to DRAW A SHAPE. Looking at the point data list, it finds each endpoint within the shape:



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

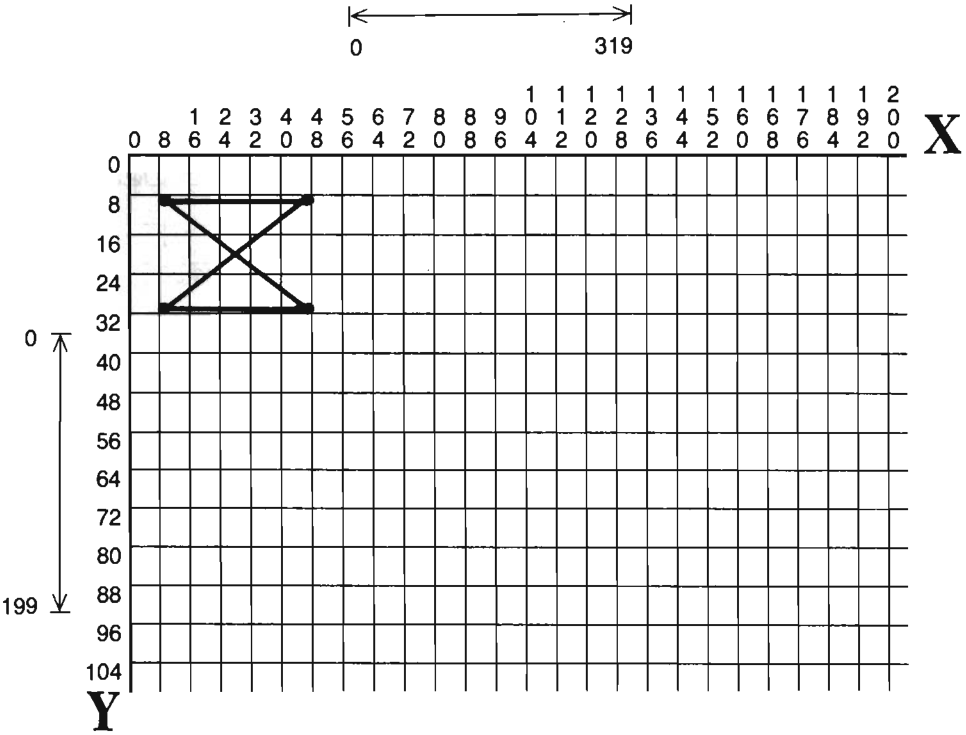


Look what happens if we keep the same point data, but change the line data to this:

Line Data

	"FROM"	"TO"
Line #0	0	1
Line #1	1	3
Line #2	3	2
Line #3	2	0

Even though the point data remained the same, the shape that is now described by these data lists looks like this:



Once you have gathered the two data lists, you are on your way. By entering these lists into your main routine the shape can be retrieved, plotted, moved, and/or duplicated simply by having the computer read each list. The next step is to enter these lists into the program.

Entering Data Lists in the Program

You will enter the data lists into the main routine as DATA statements. Data statements begin with the word DATA, and contain a succession of words and/or numbers, each separated by a comma. You need to enter DATA statements that contain the following information, in the exact order as listed here:

(1) The name of your shape.

(2) The number of points in the shape (start counting at 0).

(3) The number of lines in the shape (start counting at 0).

(4) The point data.

(5) The line data.

To see how this is done, let's define the rectangle. First, type RUN 172 and press **RETURN** to ZAP your current main routine. You will enter a new main routine that draws shapes using DATA statements. Again, the point and line data lists for our rectangle are:

Point Data

	X	Y
Endpt #0	8	8
Endpt #1	47	8
Endpt #2	47	31
Endpt #3	8	31

Line Data

	"FROM"	"TO"
Line #0	0	1
Line #1	1	2
Line #2	2	3
Line #3	3	0

Add the main routine lines below that name this shape and give its count of points and lines:

```
1ØØ3 REM:      SHAPE LIBRARY      :
1ØØ4 REM:::::::::::::::::::::::::
1ØØ6 DATA "RECTANGLE", 3, 3
```

There are several things to notice about these new programming lines. First, a REM statement has titled this section of the program SHAPE LIBRARY. This is because lines 1006 through 1999 are going to be reserved for your "library" of shapes. It is important that all shape descriptions be placed at the beginning of your main routine, and that they remain grouped together. Setting aside approximately 1,000 program lines dedicated to shape storage should be sufficient for almost any picture.

Second, notice that the shape's name ("RECTANGLE") is the first data item given within the DATA statements. The program will search for this name whenever you retrieve the shape. If the name is not the first item listed, the RETRIEVE A SHAPE tool will *not* be able to find the shape. Upper-case and lower-case matters when naming your shapes. If we were to name this shape "rectangle," we could only retrieve it by searching for "rectangle" (as opposed to perhaps "RECTANGLE"). The name must be within quotes, but may be any length or construction you desire.

Give the shape's count of points and lines in the same DATA statement. These values must immediately follow the shape's name, with the count of points given before the count of lines. In addition, these values are "zero-based" values; that is, the count starts at *zero*. Because our rectangle has a true count of four endpoints (1-2-3-4), the zero-based count is three (0-1-2-3). Similarly, because the true count of lines is four, the zero-based count of lines is three. It is very important to enter these zero-based counts correctly.

The next DATA statement to enter contains the point data of our shape. Enter this into your main routine:

```
1ØØ8 DATA 8, 8, 47, 8, 47, 31, 8, 31
```

Look back at the point data list and compare it to the DATA statement above. Although the endpoint numbers (0, 1, 2, and 3) are not typed into the program, you can see how they are used to properly list each endpoint:

```
           Endpt #Ø  Endpt #1  Endpt #2  Endpt #3
             X, Y      X, Y      X, Y      X, Y

1ØØ8 DATA   8, 8,     47, 8,    47,31,     8,31
```

When entering your point data, always put endpoint #0's coordinates first, then endpoint #1's, then endpoint #2's, and so on. Also, be sure to list each X coordinate *before* each Y coordinate, separating all the data with commas. The last data item in any DATA statement should end *without* a comma.

Finally, enter the line data. For our rectangle, type this DATA statement:

```
1Ø1Ø DATA  Ø, 1, 1, 2, 2, 3, 3, Ø
```

Notice how the line data is entered, beginning with line #0 and continuing through line #3:

```
           Line #Ø  Line #1  Line #2  Line #3
            Fr,To,   Fr,To,   Fr,To,   Fr,To

1Ø1Ø DATA   Ø, 1,     1, 2,    2, 3,    3, Ø
```

These three DATA statements completely define and describe our RECTANGLE shape. DATA statements can be a maximum of 2 screen lines in length, making it necessary to break the data up for more complex shapes. As long as you begin each line with the word DATA, and enter all the data in the proper order, you should have no problems.

Retrieving and Drawing Your Shapes

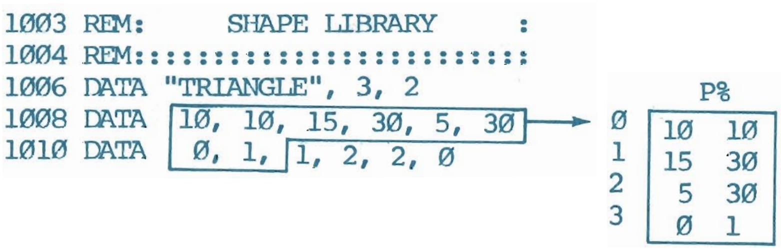
Currently, you shape is only defined in the main routine. To actually use it in a picture, the DATA statements must be READ into memory, and then sorted into points and lines. You accomplish this with the RETRIEVE A SHAPE tool.

This tool reads the point data into a P% array, and the line data into an L% array. These arrays are "computer lists" forming temporary storage places for the shape's description. Each time you retrieve a new shape, the previous shape description is over-written, and thus erased.

When you retrieve a shape, the computer always begins reading at the very *first* DATA statement in your program. DATA statements serve many purposes, and it is possible that you might use them in your program for something other than shape storage. That is why it is important that your shape data be placed at the very beginning of your program. That way, they get read first whenever you call the RETRIEVE A SHAPE tool.

When the computer reads shape data, it first makes a note of the shape's name. It then looks to see how many points the shape has. Based on *this* number, the computer fills the P% array with that many pairs of data items (X,Y). Next, it checks to see how many lines the shape should have. Based on that, it places that number of endpoint number pairs into the L% array. Finally, if this was indeed the shape to retrieve, the computer stops. However, if this was not the shape to retrieve, the next set of DATA statements is read into P% and L%—erasing the first set. This process continues until the correct shape has been read into P% and L%.

Always make sure you have correctly given the number of points and lines for each shape. Look at the diagram below to get an idea of what can go wrong if you don't. In it, the shape is a "TRIANGLE", and it is noted as having four endpoints (0-1-2-3), and three lines (0-1-2). A triangle, of course, only has three endpoints. Since the computer has no idea what a triangle is, one with four endpoints won't stop it. Instead, it immediately reads four sets of data into P%.

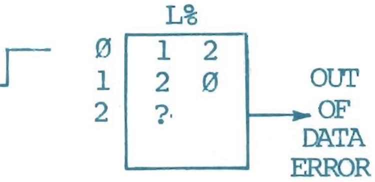


```
1ØØ3 REM:       SHAPE LIBRARY       :
1ØØ4 REM:::::::::::::::::::::::::
1ØØ6 DATA "TRIANGLE", 3, 2
1ØØ8 DATA  1Ø, 1Ø, 15, 3Ø, 5, 3Ø
1Ø1Ø DATA   Ø, 1, 1, 2, 2, Ø
```

	P%	
Ø	1Ø	1Ø
1	15	3Ø
2	5	3Ø
3	Ø	1

Next, it tries to fill L%. Since there are suppose to be three lines in the shape, it tries to place three sets of data items into L%. Finding only two pairs of data left, the program is abandoned and an OUT OF DATA ERROR occurs.

```
1ØØ3 REM:      SHAPE LIBRARY      :
1ØØ4 REM::::::::::::::::::::::::::
1ØØ6 DATA "TRIANGLE", 3, 2
1ØØ8 DATA  1Ø, 1Ø, 15, 3Ø, 5, 3Ø
1Ø1Ø DATA   Ø, 1, 1, 2, 2, Ø
```



If you understate the number of points, not all of them will be placed in P%. Instead, those not placed in P% are the first data items placed into L% when it is filled. This will produce a distorted, if not totally unrecognizable, shape when drawn.

Type the RETRIEVE A SHAPE subroutine into your program:

```
8ØØ REM::::::RETRIEVE A SHAPE
8Ø1 RESTORE
8Ø2 READ S$,ND,NL
8Ø3 FOR I = Ø TO ND
8Ø4 READ P%(I,Ø),P%(I,1):P%(I,2)=1
8Ø5 NEXT I
8Ø6 FOR I = Ø TO NL
8Ø7 READ L%(I,Ø),L%(I,1)
8Ø8 NEXT I
8Ø9 ON ABS(S$<>SE$)GOTO 8Ø2: RETURN
```

You need a way to test your RETRIEVE A SHAPE tool to see if you entered it properly. One sure way is to try drawing the shape. Type the DRAW A SHAPE tool below:

```
9Ø REM::::::DRAW A SHAPE
92 FOR J = Ø TO NL
93 E1%=L%(J,Ø): E2%=L%(J,1)
94 X1=P%(E1%,Ø): Y1=P%(E1%,1)
95 X2=P%(E2%,Ø): Y2=P%(E2%,1)
96 GOSUB 5Ø
97 NEXT J
98 RETURN
```

Don't worry that there is no line numbered 91. You will be adding it in a later chapter, when modifications to this tool will be necessary. For now, type the tool exactly as shown.

Finally, add these main routine lines that get you into and out of graphics mode, and then call upon the tools that will produce the rectangle:

```
2ØØØ GOSUB 1Ø: C=7: GOSUB 3Ø
2Ø1Ø REM::FIND/DRAW REC
2Ø2Ø DIM P%(99,2), L%(99,1)
```

```
2Ø3Ø SE$="RECTANGLE": GOSUB 8ØØ
2Ø4Ø C=2: GOSUB 9Ø
6ØØØ GET A$
6Ø1Ø IF A$ ="←" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
```

These lines will be explained in a moment. First RUN the program to check it. A red rectangle on a yellow background screen should appear near the top left side of your screen. Because you are using machine language to draw the points and lines, this will happen quickly.

Stop the program by pressing ←. If you had any problems, check the main routine, and then the new subroutines. Make sure the program works properly before continuing.

Let's go over the program now.

Line 2000 takes you into graphics mode. It then sets the color variable to a color code of yellow, and sets all pixels to this yellow background color.

Line 2020 gives the DIMensions (sizes) of your point data list and your line data list. You begin by assigning a name to each list. We have chosen the name "P" for the Point data list, and "L" for the Line data list. The "%" tells the computer that the lists will store integer values. It is important to *dimension* (set the dimensions of) your lists one time in the program. When you do, always use the names P% and L%.

The numbers in parentheses are called *subscripts*. These tell the computer the size of each list. The first number tells how many *rows* the list needs. This is a zero-based number, so the 99 for each list means they will both need 100 rows.

The second number tells how many *columns* each list will need. This is also a zero-based number, so the 1 for L% means it will need two columns, and the 2 for P% means it will need three columns. Although your Points list (P%) will never have more than 2 columns, it *must* be dimensioned with 3 columns for technical reasons (this is explained in a later chapter, when you have learned more about the workings of the DRAW A SHAPE tool).

Given the size (dimensions) of each list, the computer will set aside enough memory space to store them (no more and no less). That memory space will not be used for anything else. If you specify that a list only needs 10 columns, and then enter 11 columns worth of data, you will get a "BAD SUBSCRIPT ERROR" because not enough memory space was set aside for 11 columns.

Obviously, our rectangle does not use 100 rows for either list (P% or L%). To find out how many rows it does need, look back at the last endpoint number and last line number on each list. The rectangle's last endpoint was #3, and last line was #3. So, we could have dimensioned these lists as DIM P%(3,2),L%(3,1). The reason we dimensioned them for 100 rows is very important: you can only dimension a list *one time* in a program. The P% and L% lists will be used repeatedly for each shape you draw in the program. Thus, the number of endpoints and lines will always be changing. By setting P% and L% to 100 rows, most shape descriptions will fit on the lists.

What this DIM statement reduces to is this:

—You must have it in your program;
—You should only have it once;
—The 99's only need to be changed for shapes involving more than 100 points or 100 lines. Otherwise, enter DIM P%(99,2),L%(99,1) and forget about it.

Line 2030 sets the search variable SE$ to "RECTANGLE". This variable must be set to your shape's name in order to retrieve it. Immediately after that, Tool 800 RETRIEVE A SHAPE is called.

Line 2040 draws the shape once it has been retrieved. This is done by changing the color so that the shape can be seen, and then calling Tool 90, DRAW A SHAPE.

The loop beginning at line 6000 allows you to exit from the graphics by pressing ←. This loop causes the computer to continually watch the keyboard, noting keypresses as they occur. When you press the arrow key, the TEXT tool is called and the program ENDs.

Try something different. Change your shape's line data to the following and then RUN the program again:

```
1Ø1Ø DATA Ø, 1, 1, 3, 3, 2, 2, Ø
```

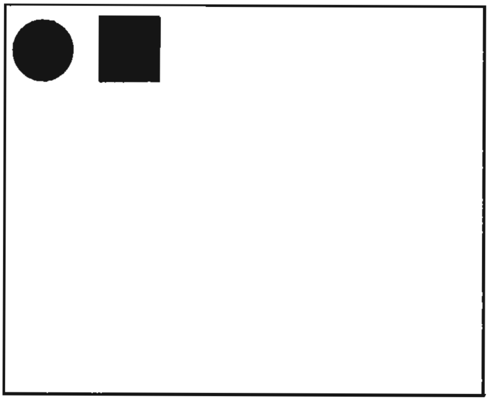
You will produce the straight-sided, hourglass figure discussed earlier. Try these lines:

```
1ØØ6 DATA "RECTANGLE", 3, 5
1Ø12 DATA Ø, 3, 1, 2
```

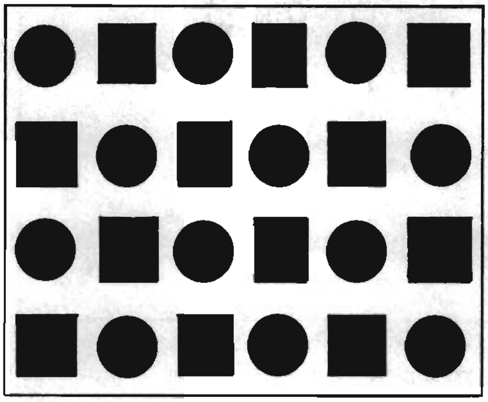
Notice that we had to change line 1006 to reflect the two new plotted lines we are adding in line 1012.

RUN the program. You have a new figure—a rectangle with an X through it. This shows another advantage of using DATA statements to store shape descriptions. Minor additions or changes to the data allows considerable flexibility to the way a shape will finally appear on the screen.

Basic shapes, such as your rectangle, can be used to create interesting patterns. Take the circle and square sketched below, for example.



By repeating these shapes across your screen, an appealing abstract design can be achieved:



Shapes can be irregular looking, like an ink blob or a bubbly cloud. They can be organic and natural looking, like a leaf or a sea shell. Shapes can also be geometric in appearance, as in the sketches above. Basic geometric shapes include the circle, square, and triangle. Geometric shapes are simple figures made of lines that connect at each end, but never cross over each other.

Change your rectangle data back so that it *does* define a rectangle:

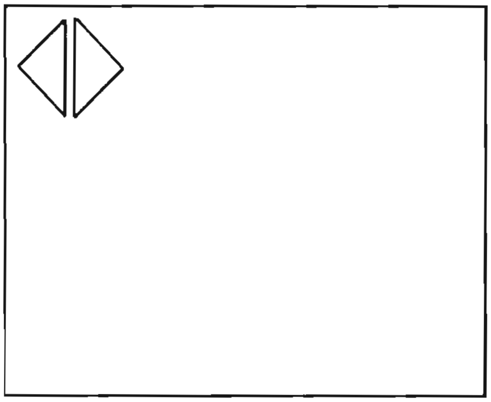
```
1ØØ6 DATA "RECTANGLE", 3, 3
1Ø1Ø DATA Ø, 1, 1, 2, 2, 3, 3, Ø
1Ø12
```

You should delete line 1012 entirely.

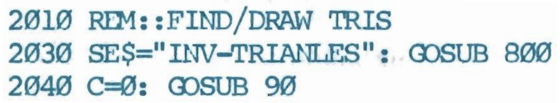
Now, enter these DATA statements into your SHAPE LIBRARY:

```
1Ø2Ø DATA "INV-TRIANGLES", 5, 5
1Ø22 DATA 3Ø, 8,3Ø,4Ø,16,23
1Ø24 DATA 33, 8,33,4Ø,47,23
1Ø26 DATA  Ø, 1, 1, 2, 2, Ø
1Ø28 DATA  3, 4, 4, 5, 5, 3
```

These DATA statements define the two basic triangles sketched below. You have entered them into the library as one shape; that is, they can only be retrieved together since they are defined together.



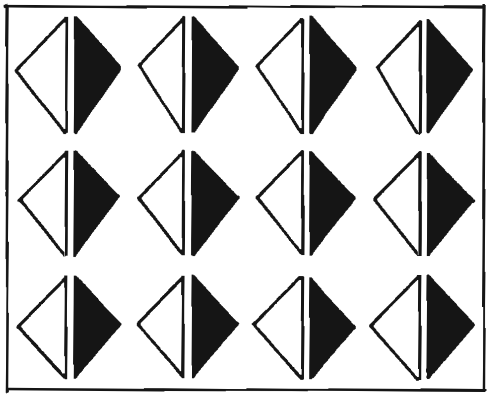
Change your main routine as follows in order to plot the new shape:



```
2Ø1Ø REM::FIND/DRAW TRIS
2Ø3Ø SE$="INV-TRIANLES": GOSUB 8ØØ
2Ø4Ø C=Ø: GOSUB 9Ø
```

RUN the program. The inverse triangles will be plotted in black on your screen.

Repeating and painting these two shapes creates an altogether different design than that created by the circle/square combination:



Shapes have different visual characteristics. For example, a circle, which is continuously curved and smooth, is much different than the pointed lines and angular directions of a triangle. A picture takes on different meaning depending on the kind of shape used. When you repeat similar shapes across a picture, you achieve a feeling of harmony.

In later chapters you will learn how to repeat a shape around the screen, as well as how to paint in shapes with color. You might want to retrieve the "RECTANGLE" or "INV-TRIANGLES" shape to experiment with at that time.

Below are the tool boxes for Tools 90 and 800. After that, we present a section on multi-color.

TOOL 90 :::::: DRAW A SHAPE

```
90 REM::::::DRAW A SHAPE
92 FOR J = Ø TO NL
93 E1%=L%(J,Ø): E2%=L%(J,1)
94 X1=P%(E1%,Ø): Y1=P%(E1%,1)
95 X2=P%(E2%,Ø): Y2=P%(E2%,1)
96 GOSUB 5Ø
97 NEXT J
98 RETURN
```

What It Does: This tool will draw the shape described in the P% and L% arrays. These arrays are most easily filled with Tool 800. Thus, you should always RETRIEVE A SHAPE (see next tool box) before drawing it.

Example Use: To use this tool, three steps are necessary:

(1) The RETRIEVE A SHAPE tool must be used to fill P% and L% with the proper shape description.
(2) You must set the color variable C to the desired plotting color.
(3) You need a GOSUB 90 statement to call this tool.

Technical Description: To draw a shape, use this loop:

```
FOR J = Ø TO NL
```

NL is the number of lines in the shape (0-based) as found by Tool 800. Thus, this loop will process once for each line in the shape. The first time through the loop, a line will be drawn between the points represented by the first entry in the L% array:

```
L%(Ø,Ø), L%(Ø,1)
```

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

```
L%(1,Ø), L%(1,1)
```

L%(J,0) and L%(J,1) are the two endpoints that determine the line to plot (where J is replaced with a number by the loop). These endpoints were previously stored in the P% array by the RETRIEVE A SHAPE tool.

```
93 E1%=L%(J,Ø): E2%=L%(J,1)
```

The program line above retrieves the two endpoint #'s of the current line to draw.

```
94  X1 = P%(E1%,Ø):Y1=P%(E1%,1)
95  X2 = P%(E2%,Ø):Y2=P%(E2%,1)
```

These lines look at P% to find out the actual X,Y coordinates of each retrieved endpoint number.

That is all the information necessary to draw a line in the shape. Tool 50 is then called.

```
96  GOSUB 5Ø
97  NEXT J
```

This is the bottom of the loop.

```
98  RETURN
```

This returns the computer to the main routine once all lines in the shape have been plotted.

TOOL 800 :::::: RETRIEVE A SHAPE

```
8ØØ REM::::::RETRIEVE A SHAPE
8Ø1 RESTORE
8Ø2 READ S$,ND,NL:
8Ø3 FOR I = Ø TO ND
8Ø4 READ P%(I,Ø),P%(I,1):P%(I,2)=1
8Ø5 NEXT I
8Ø6 FOR I = Ø TO NL
8Ø7 READ L%(I,Ø),L%(I,1)
8Ø8 NEXT I
8Ø9 ON ABS(S$<>SE$)GOTO  8Ø2: RETURN
```

What It Does: This tool fills the P% and L% arrays with the shape description named by SE$. It is necessary to fill these arrays if a shape is to be drawn, or later scaled, rotated, or translated.

Example Use: You must take several steps to use Tool 800:

(1) Draw the shape on graph paper.
(2) Assign a number to each endpoint, starting with Endpoint #0.
(3) Write down the X,Y coordinates of each endpoint, starting with Endpoint #0. Do this in the following form:

Point Data

	X	Y
Endpt #0	8	8
Endpt #1	47	8
Endpt #2	47	31
Endpt #3	8	31

(4) Write down the "FROM" and "TO" endpoint numbers of each line in the shape. Number these lines, beginning with Line#0, using the following format:

Line Data

	"FROM"	"TO"
Line #0	0	1
Line #1	1	2
Line #2	2	3
Line #3	3	0

(5) Within lines 1005-1999 (your Shape Library), insert a DATA statement that names the shape, and gives its zero-based count of points and lines. For example:

```
1ØØ5 DATA "RECTANGLE", 3, 3
```

(6) Immediately following the above DATA statement, insert DATA statement(s) that give the X,Y coordinates for each endpoint in the shape. Be sure to enter Endpoint #0's coordinates first, then Endpoint #1's, Endpoint #2's, etc. The X coordinate of an endpoint should always come before the Y coordinate. Use this format:

```
           Endpt.Ø   Endpt.1   Endpt.2   Endpt.3
1ØØ8 DATA   8, 8,     47, 8,    47, 31,   8, 31
```

(7) Immediately following your point data, insert your line data. This will be DATA statements having the "FROM" and "TO" endpoint numbers of each line to plot. Be sure to list Line #0's endpoint numbers first, then Line #1's, Line #2's, etc. The format for this is:

```
              LineØ   Line1   Line2   Line3
1Ø1Ø DATA     Ø, 1,   1, 2,   2, 3,   3, Ø
```

(8) If this is the *first* shape your program is going to retrieve, you must dimension the P% and L% arrays in the main routine. This can usually be done with a program line containing the following:

```
2Ø2Ø DIM P%(99,2), L%(99,1)
```

If any shape in your program has more than 100 points, then you should change the 99 that follows P% accordingly. If any shape has more than 100 lines, then you should change the 99 that follows L% accordingly.

(9) Set the variable SE$ to the shape's name as listed in your DATA statements, and then call the RETRIEVE A SHAPE tool:

```
2Ø3Ø SE$="RECTANGLE": GOSUB 8ØØ
```

Technical Description: The computer has a "pointer" which keeps track of the data items that have already been read into memory by a

READ statement. This pointer moves to each successive data item, as each one is read into memory. Thus, it is always "pointing" to the next data item to read. This is why, regardless of where a READ statement appears in your program, it will always read the first DATA items listed in the program, or pick up where the last READ statement left off.

RESTORE is a basic command which resets the pointer to the beginning of the DATA statements in your program. Thus, RESTORE is the key to the RETRIEVE A SHAPE subroutine, which needs to always begin searching for a specified shape from the beginning of the DATA statements.

Line 802 retrieves the name, number of points, and number of lines of the specified shape in the DATA statement. If any of these data items are incorrect, the subroutine will not function properly. Lines 803 through 805 read the correct number of points. Lines 806 through 808 read the correct number of lines. Line 809 checks to make sure that the shape just retrieved matches the one requested in SE$. If the retrieved shape matches the requested shape, then the subroutine returns the computer to the main routine. If not, the subroutine goes to line 802 to get the next shape in the program.

Multi-Color

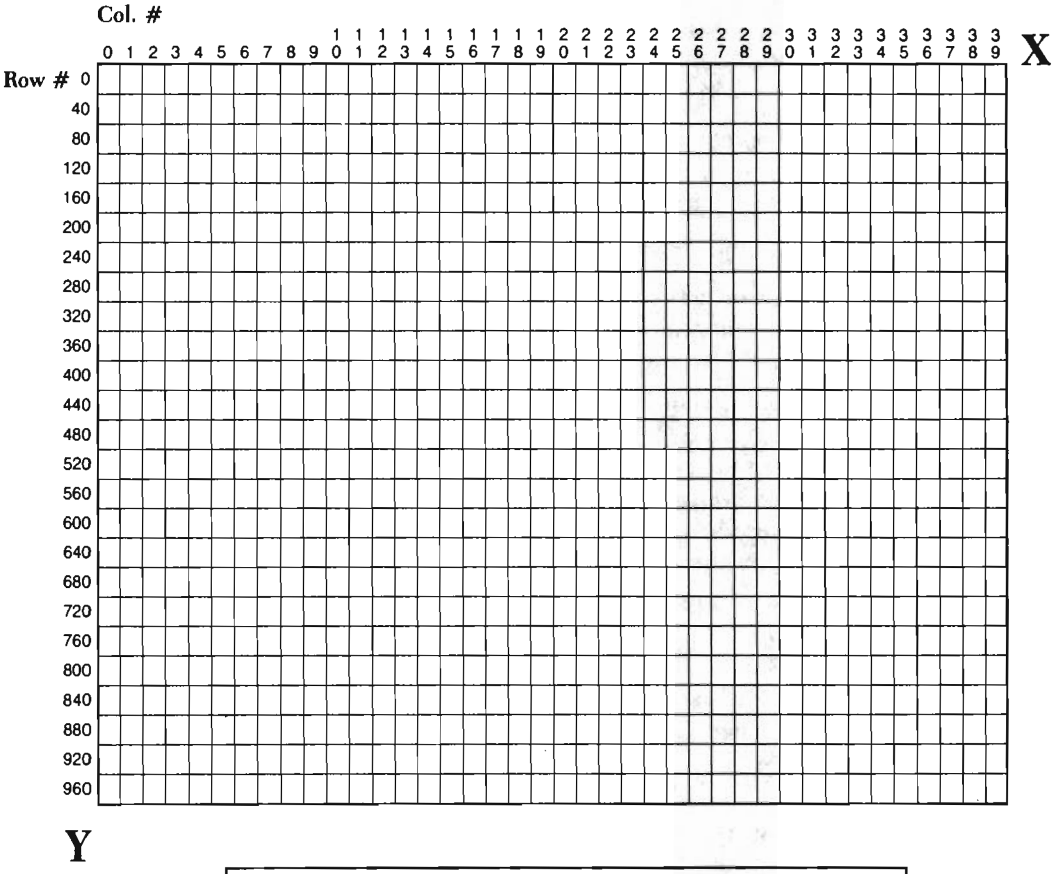
Until now, we have said very little about multi-color. However, you have had access to it for quite some time. By setting the variable MU to 1 (MU=1) before going into graphics mode, your picture will be displayed in multi-color. Change the following line in your program and RUN it:

```
2000 MU=1: GOSUB 10: C=7: GOSUB 30
```

Notice how the shapes have changed. Each sloping line is a little more rough and jagged. This is because multi-color cuts your vertical resolution *in half*. This means that every two pixel columns are now treated as one, producing a somewhat chunky effect any time you plot a diagonal line.

That is the only disadvantage to consider when choosing between high resolution graphics and multi-color. The advantage to using multi-color is the increased number of colors you can use per color block. Your screen is divided up into 1,000 color blocks, as shown here:

TOP OF SCREEN



COLOR BLOCK # = COL. # + ROW #

MEM. LOC. = 17408 + COLOR BLOCK #

COLOR MEMORY

(The X,Y PIXEL POINTS grid we have been using throughout this book has been divided up into these color memory blocks.)

Using high resolution, you are able to have only one foreground color and one background color in each block. With multi-color, you can still have only one background color per block, but the limit on foreground colors jumps to three per block. This is substantial, and often necessary in more detailed designs. To give you an idea of how multi-color can be of use, add the program lines below:

```
2001 C=1: X1=Ø: Y1=Ø
2002 X2=319: Y2=199: GOSUB 5Ø
2003 C=2: X1=16Ø: Y1=Ø
```

```
2004 X2=160: Y2=199: GOSUB 50
2005 C=0: X1=319: Y1=0
2006 X2=0: Y2=199: GOSUB 50
2007 GOTO 6000
```

With MU still set equal to one, RUN the program. Three lines, each a different color, will be plotted through the center of your screen.

Stop the program and change line 2000 back to:

```
2000 GOSUB 10: C=7: GOSUB 30
```

RUN the program again. You will quickly see the problem. The block in which the center point resides can only have two colors in high resolution graphics. The program, however, is trying to use four colors in it. Think how this gets resolved.

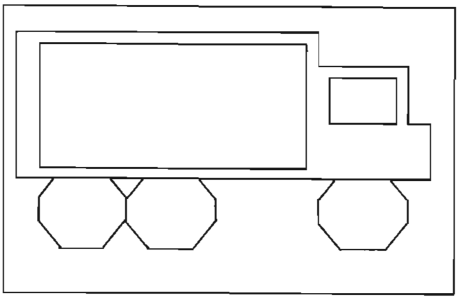
There is one background color given in the program. This is allowed in high resolution, so it is used to paint the background. Next, the first line is plotted. Because two colors are permitted anywhere on the screen, this second color causes no problems. However, as each additional line is plotted, sections of lines running through the center block are changed to the current plotting color. This keeps the center block within the limit of one foreground and one background color.

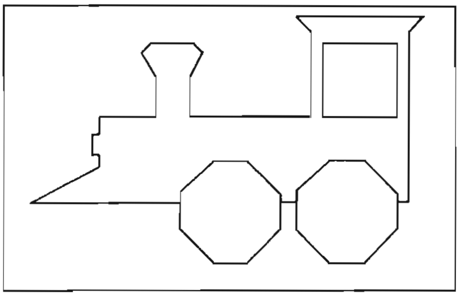
Try working with a more complex shape. To do this, you need to erase the main routine. This could be done by running the ZAP routine as is, but that would erase your library of shapes (the ZAP routine currently zaps all lines after line 1002). Fortunately, there is a way to modify the ZAP routine so that it begins zapping at a different line number. Change line 172 to this:

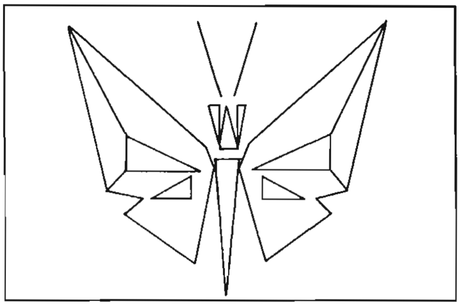
```
172 A = 256: B = 2049: C = 2000
```

The only change is that C = 1003 becomes C = 2000. Zapping will now take place beginning at line 2000. Type RUN 172 and press **RETURN**. Wait as lines 2000 and higher are deleted from the program. When the job is done, LIST your program to make sure no lines before 2000 were erased. If so, you will need to re-type them.

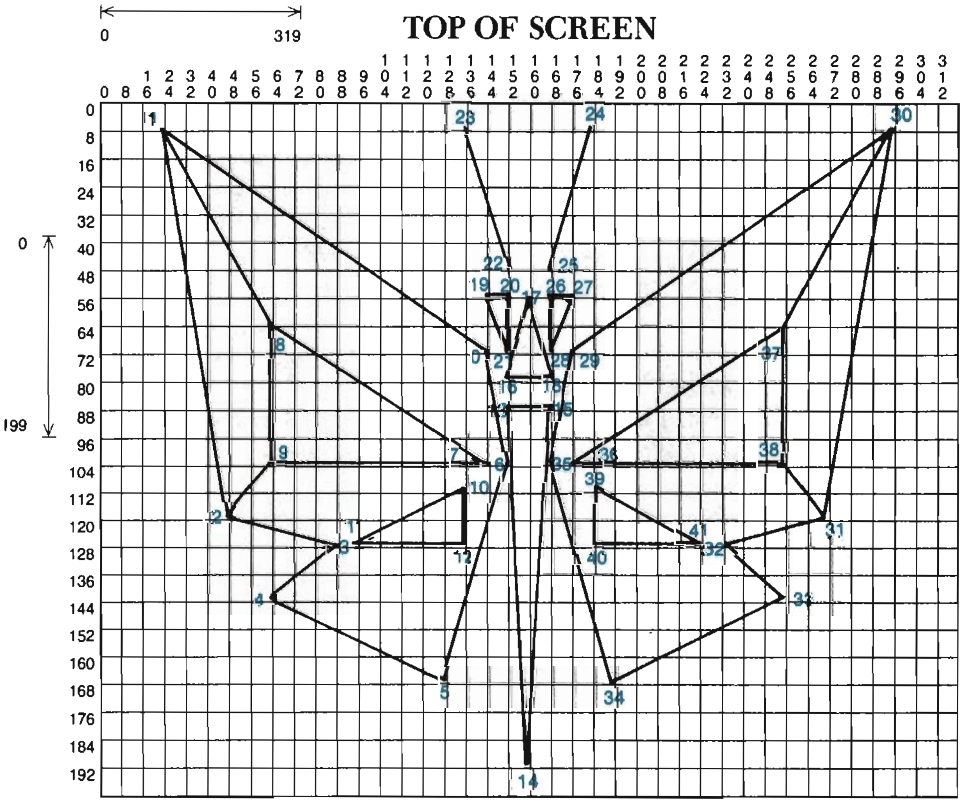
The three sketches below show whole objects composed of many different lines and pieces. By grouping some of the more basic shapes together (triangles, squares, octagons, etc), you can form larger objects. These objects are viewed as whole shapes rather than separate pieces.







The butterfly is a nice, angular object that will appear quite different in multicolor than in high resolution. Look at the butterfly sketched on our X,Y PIXEL POINTS grid:



X, Y PIXEL POINTS

Each endpoint on the grid has been assigned a number, ranging from endpoint #0 through endpoint #41. (Thus, there are 41 endpoints in our zero-based count.) If you were to total the number of lines, starting with line #0, you would find that there are 44 of them. Type the following program line to name the new shape and give its count of points and lines:

```
1Ø3Ø DATA "BUTTERFLY",41,44
```

The point data follows. Before you begin typing it, notice how the data is neatly organized. All of the commas from one DATA statement to the next are carefully lined up. This was not done merely for appearance sake. Arranging DATA statements in this manner will actually help you locate typing errors. Since each DATA statement is of equal length, and each data item is spaced to line up with the one above it, you can easily see if a data item was added or omitted, or if a single digit was added/omitted. Try entering the DATA statements exactly as shown here.

```
1Ø32 DATA 143, 71, 23,  7, 47,119
1Ø34 DATA  87,127, 63,143,127,167
1Ø36 DATA 151,1Ø3,143,1Ø3, 63, 63
1Ø38 DATA  63,1Ø3,135,111, 95,127
1Ø4Ø DATA 135,127,151, 87,159,191
1Ø42 DATA 167, 87,151, 79,159, 55
1Ø44 DATA 167, 79,143, 55,151, 55
1Ø46 DATA 151, 71,151, 47,135,  7
1Ø48 DATA 183,  7,167, 47,167, 55
1Ø5Ø DATA 175, 55,167, 71,175, 71
1Ø52 DATA 295,  7,271,119,231,127
1Ø54 DATA 255,143,191,167,167,1Ø3
1Ø56 DATA 175,1Ø3,255, 63,255,1Ø3
1Ø58 DATA 183,111,183,127,223,127
```

Check your typing one final time for mistakes. Next, you need to type the line data. This has also been organized into neat columns. Type it just as listed:

```
1Ø6Ø DATA  Ø, 1, 1, 2, 2, 3, 3, 4
1Ø62 DATA  4, 5, 5, 6, 6, Ø, 7, 8
1Ø64 DATA  8, 1, 8, 9, 9, 2, 9, 7
1Ø66 DATA 12,1Ø,1Ø,11,11,12,22,23
1Ø68 DATA 2Ø,19,19,21,21,2Ø,16,17
1Ø7Ø DATA 17,18,18,16,13,15,15,14
1Ø72 DATA 13,14,25,24,26,27,27,28
1Ø74 DATA 26,28,35,29,29,3Ø,3Ø,31
1Ø76 DATA 32,31,32,33,34,33,35,34
1Ø78 DATA 29,35,36,37,37,3Ø,37,38
1Ø8Ø DATA 36,38,39,41,4Ø,41,39,4Ø
1Ø82 DATA 38,31
```

That completes the butterfly's description. The main routine lines that will draw this shape should be typed as follows:

```
2Ø1Ø GOSUB 1Ø: C=14: GOSUB 3Ø
2Ø2Ø DIM P%(99,2), L%(99,1)
2Ø3Ø SE$="BUTTERFLY": GOSUB 8ØØ
2Ø4Ø C=2: GOSUB 9Ø
6ØØØ GET A$
6Ø1Ø IF A$= "←" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
```

Notice that we are going to paint the background a light blue (C=14 in line 2010). Recall that this will make the text "invisible" when we return to text mode, *unless* we change the color of the text letters in advance. Press **CTRL** and **2** at the same time to change the typing color. The computer will then display all text in white from now on.

> IMPORTANT: There will be times when you run a program that paints the background light blue, without first changing your text letters to white. On return to the text mode, the entire screen will be light blue (giving the appearance that you are still in the graphics mode). Type POKE 53281,1 **RETURN** when this happens, and the text background will automatically change to white.

Remember that this program may begin by loading "M/L" when run, so be sure to have the appropriate disk/tape available. Although it is not necessary, it's a good idea to save your program before running it since so much typing is involved. Save it under any name you like, and then RUN the program.

The butterfly should be quickly plotted in red across your screen. This interesting shape has many possibilities, and is one of the nicer additions to your library. Because it is made up of an assortment of smaller shapes, it will be a good design to experiment with in Chapter 3, where you learn how to paint shapes.

If you have any difficulties, try pressing ← to return to text mode. If that doesn't work, press **RUN/STOP** and carefully type GOSUB 20 **RETURN** (you won't be able to see your typing, so type carefully). Check the following list of possible problems/solutions:

Is there an Error Message on your text screen?

—If an OUT OF DATA ERROR occurs, check the DATA statement in line 1030. Make sure the count of points and lines is correct. Another possibility is that you forgot one or more data items, or even an entire DATA statement. Finally, make sure you set SE$ to the correct shape name. Remember, upper-case/lower-case counts when setting this variable to the shape named in the DATA statements.

—If a BAD SUBSCRIPT ERROR occurs, check the DIM statement in line 2020. If that's okay, check over your data.

—If a FILE NOT FOUND error occurs, you forgot to insert the disk/tape containing your "M/L" file.

—If a SYNTAX ERROR occurs, the problem is probably in the line number given in the error message. However, if this error occurs in a DATA statement, then check the number of points and number of lines you gave for *all* shapes.

Is the butterfly being plotted irregularly?

—Check that your line data is connecting the right points together.

—Check the DATA statement in line 1030. Make sure you have given the correct, zero-based count of points and lines.

Was only one line plotted off course?

—Check your line data (lines 1060 through 1082). One line may be plotted with an incorrect endpoint number. If that is not the problem, check your endpoint data.

Was nothing plotted?

—Check that C equals 2, and not 14, in line 2040. Also, check that tool 90 is called in this line.

—Make sure that tool 800 was called in line 2030.

When you have a long list of DATA statements that contain a typing error somewhere, a friend is always an asset. While you are looking at the DATA statements on the screen, have your friend read them out of our text (or, later, from your handwritten list). This goes much faster than trying it on your own, and you usually spot the problem quicker.

Once this program is working properly, try it out in multi-color. Add this line:

```
2000 MU=1
```

RUN the program. The butterfly should be plotted in the same location and color, but should appear a little chunkier and less refined. Stop the program, and delete line 2000.

There is one more thing you should know about multi-color: you *must* keep the program in a continuous loop after drawing a multi-color picture. This is because the foreground color of the text screen is stored in the same memory location as the third foreground color of multi-color. The text screen sometimes scrolls a couple of lines after a program is completely finished executing. This will cause the third color of your multi-color picture to scroll up a few lines in the picture. Ending the program with a continuous loop (lines 6000 through 6020) keeps the computer from scrolling the text screen until you are ready to return to text mode.

Multi-color is most useful in pictures having a concentration of colors that fall into one color block. In one short section we can't provide you with a main routine that would do justice to multi-color. As you work more and more with Commodore graphics, you will undoubtedly come across pictures that will only work in multi-color. By first sketching each picture on graph paper, within a 40 x 25 block area, you can determine ahead of time if multi-color will be necessary. If you draw a picture in high resolution graphics, and then run into a color block problem, try setting MU equal to 1. When you run the program again, you may find that the problem is eliminated.

Because of the loss of vertical resolution, we will not use multi-color to any great extent again. Nevertheless, you can use it at any time simply by setting the MU variable equal to one before GOSUB 10. Your screen will be a little different than that described here, but you should be able to follow along easily enough.

Clip a Shape

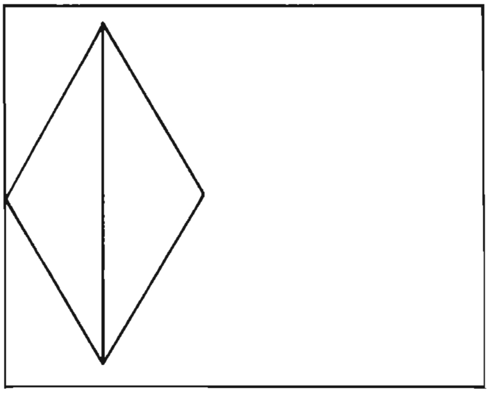
The final tool for this chapter is the CLIP A SHAPE tool. Although not as exciting as some of the others, it is nonetheless just as important. This is a long tool, so type it carefully:

```
70 REM::::::CLIP A SHAPE
71 S1=Ø: S2=Ø
72 IF X1< XL THEN S1=1
73 IF X1> XH THEN S1=2
74 IF Y1> YH THEN S1=S1+4
75 IF Y1< YL THEN S1=S1+8
76 IF X2< XL THEN S2=1
77 IF X2> XH THEN S2=2
78 IF Y2> YH THEN S2=S2+4
79 IF Y2< YL THEN S2=S2+8
8Ø IF S1=Ø AND S2=Ø THEN 5Ø
81 IF S1 AND S2 THEN RETURN
82 S=S1: IF S=Ø THEN S = S2
83 IF S AND 1 THEN Y=Y1+(Y2-Y1)*(XL-X1)/(X2-X1):X=XL:GOTO 87
84 IF S AND 2 THEN Y=Y1+(Y2-Y1)*(XH-X1)/(X2-X1):X=XH:GOTO 87
85 IF S AND 4 THEN X=X1+(X2-X1)*(YH-Y1)/(Y2-Y1):Y=YH:GOTO 87
86 IF S AND 8 THEN X=X1+(X2-X1)*(YL-Y1)/(Y2-Y1):Y=YL
87 IF S=S1 THEN X1=X: Y1=Y: GOTO 71
88 X2=X: Y2=Y: GOTO 71
```

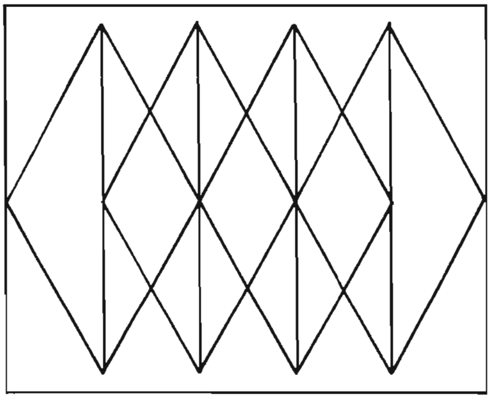
Lines 81 through 84 will wrap around on your screen. Just keep typing when this happens. There are many X's, Y's, X1's, and Y1's, so be careful. It's very easy to mix up these variables.

Before testing this tool, SAVE your program under "GRAPHICS2". This is important, because if you entered the tool incorrectly you could jam up the computer. In that event, the only solution would be to turn the computer off and then on again—losing any unsaved portions of your program.

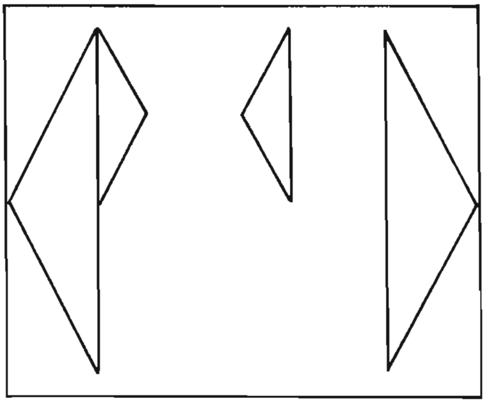
To test this tool, add yet another shape to your library. The shape shown below is larger, but similar, to the inverse triangles you plotted earlier.



This shape makes a particularly attractive pattern when repeated across the screen in an overlapping fashion. When the shapes overlap, new shapes are created.



By pulling apart this design, you can see that it is actually made with one simple shape—a triangle (see the sketch below). A simple shape like a triangle can be transformed by changing its size, its placement, and by rotating it to a new position.



Although you don't yet have the tools to transform shapes in these ways, you should be getting an idea of how you can use your Shape Library in the future.

Add this shape's data to your Shape Library:

```
1090 DATA "SPLIT-TRIANGLE", 3, 4
1092 DATA   0,   0, 15, 23,   0, 47,-15, 23
1094 DATA   0,   1,  1,  2,   2,  3,  3,  0
1096 DATA   0,   2
```

You may be wondering about the *negative* coordinate. This is necessary because we want to plot the shape partially off the screen, giving the CLIP A SHAPE tool an opportunity to clip something. Under normal circumstances, you should never intentionally plot off the screen.

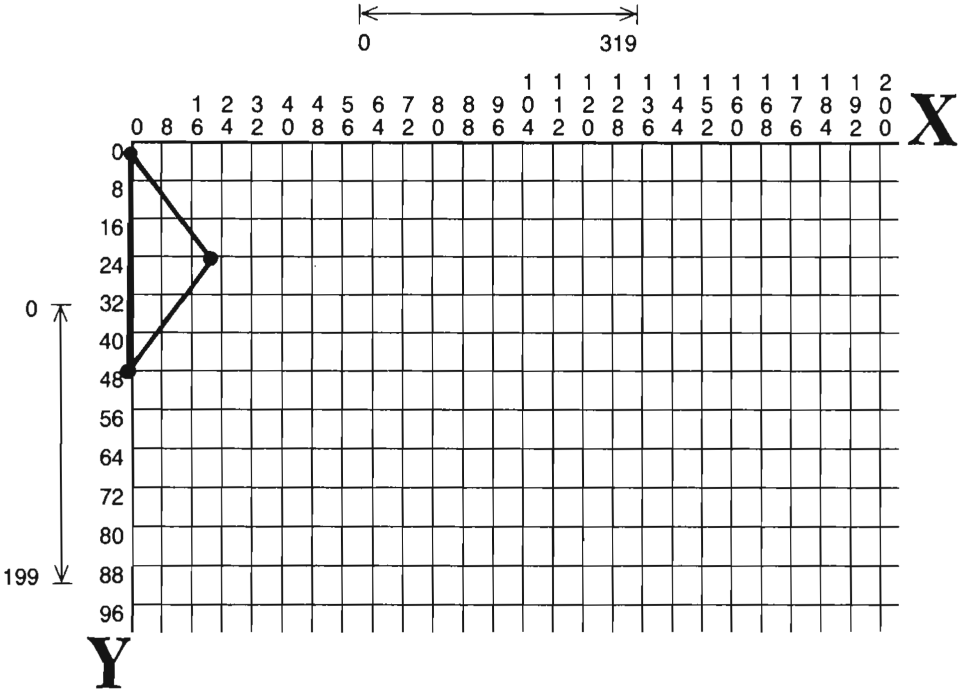
Change your main routine so that this shape is retrieved instead of the butterfly:

```
2030 SE$="SPLIT-TRIANGLE": GOSUB 800
```

Finally, change line 96 in the DRAW A SHAPE tool so that it automatically calls the CLIP A SHAPE tool (the CLIP A SHAPE tool now calls the PLOT A LINE tool):

```
96 GOSUB 70
```

RUN the program. It will take a few moments to clip parts of the shape, and only the right side of the shape will finally be plotted. This will include the center dividing line (see sketch below). If the shape is plotted in this manner, with no visible side effects, you have probably entered this tool correctly.



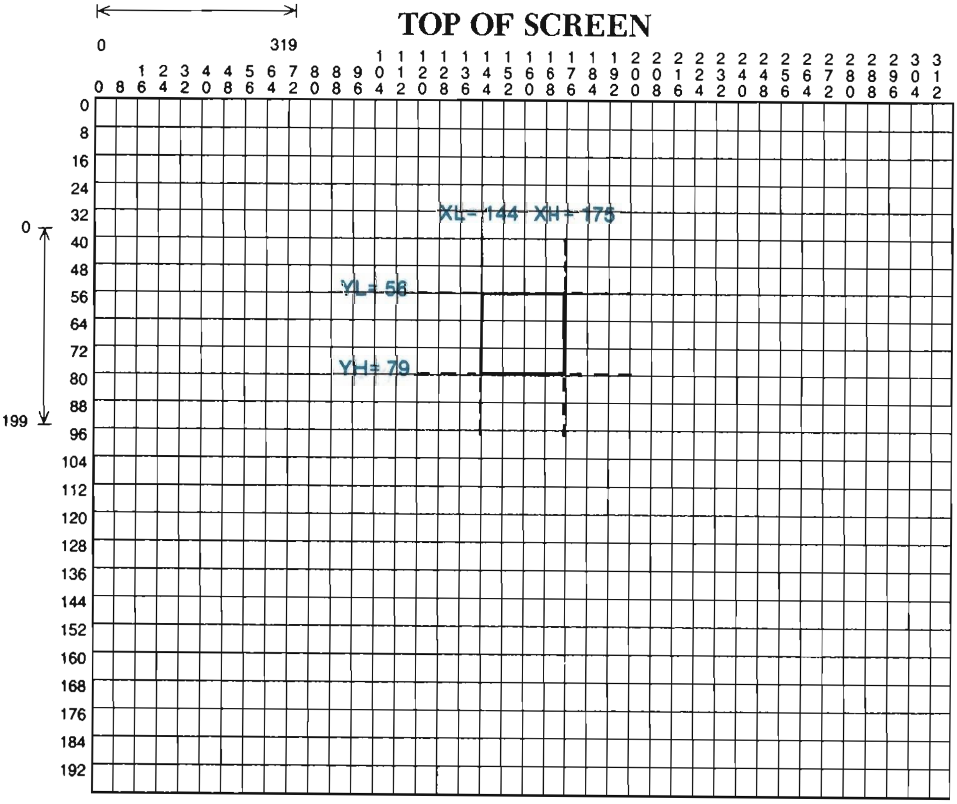
To obtain more conclusive evidence that your tool is working, change the program lines back so that they retrieve the butterfly:

```
2Ø3Ø SE$="BUTTERFULY": GOSUB 8ØØ
```

Next, you will change the clipping boundaries. Recall that in Chapter 1 you set the variables XL, XH, YL, and YH. This was done within Tool 30, on line 32. If you look at lines 72 through 79, and 83 through 86, you will find that these variables are used by the CLIP A SHAPE tool. The purpose of these variables is to establish the "clipping boundaries." XL (X-Low) and XH (X-High) define the left and right boundaries, in terms of an X coordinate. YL (Y-Low) and YH (Y-High) define the top and bottom boundaries, in terms of a Y coordinate. Change the variables as shown below:

```
2Ø35 REM:::::::SET VIEWING AREA
2Ø36 XL=144: XH=175
2Ø37 YL=56: YH=79
```

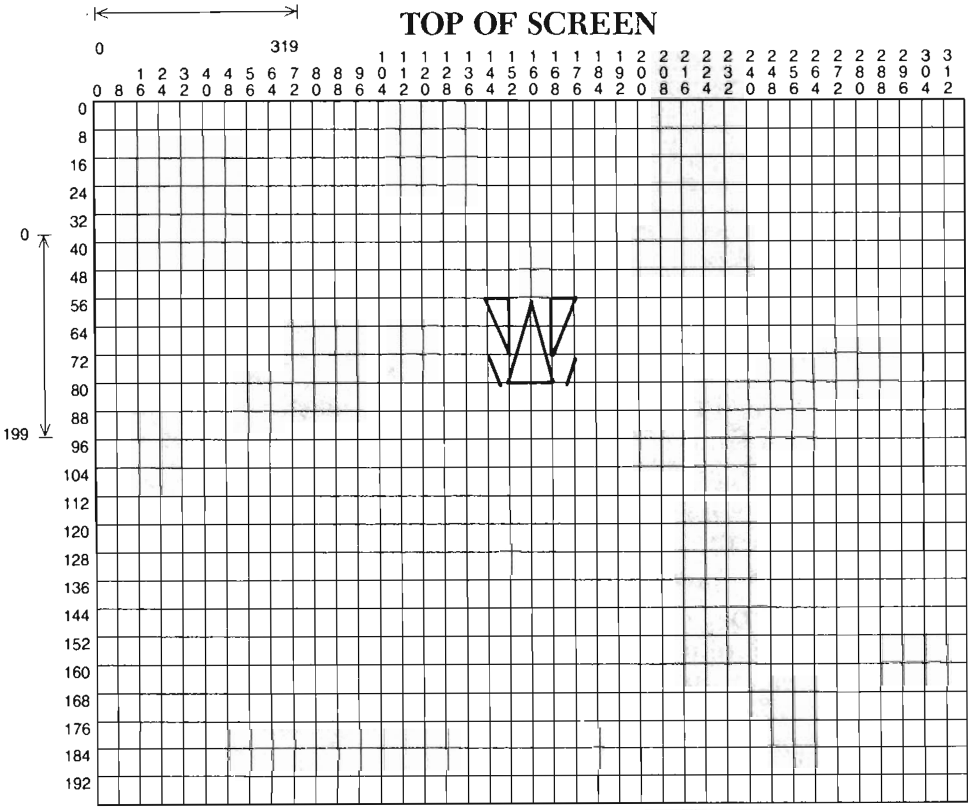
The boundaries are now set so that any points outside of the X range of 144 (Low) to 175 (High), or outside the Y range of 56 (Low) to 79 (High) will be clipped. Thus, only the rectangular area outlined below will be used for plotting. Any part of the butterfly falling outside of this rectangle will be clipped.



X, Y PIXEL POINTS

Run the program, and watch where clipping occurs now.

There are many lines to clip, so be patient. If your final screen resembles the drawing below, you can be certain you entered the clipping tool properly. If the shape was clipped, but not as shown below, check that you set your boundaries correctly in lines 2036 and 2037. Any other problems indicate an error in the tool.



X, Y PIXEL POINTS

When your program clips properly, be sure to delete lines 2035 through 2037 from your program. Keep in mind, though, that these variables can be re-set at any time to change the clipping boundaries to a new rectangular area.

> IMPORTANT: You must make sure XH is set to a higher value than XL, and that YH is set to a higher value than YL when setting these variables.

Because Tool 90 (DRAW A SHAPE) automatically calls the CLIP A SHAPE tool, you don't have to worry about including a GOSUB 70 statement in your main routine.

TOOL 70 :::::: CLIP A SHAPE

```
7Ø REM::::::CLIP A SHAPE
71 S1=Ø: S2=Ø
72 IF X1< XL THEN S1=1
73 IF X1> XH THEN S1=2
74 IF Y1> YH THEN S1=S1+4
75 IF Y1< YL THEN S1=S1+8
76 IF X2< XL THEN S2=1
77 IF X2> XH THEN S2=2
78 IF Y2> YH THEN S2=S2+4
79 IF Y2< YL THEN S2=S2+8
8Ø IF S1=Ø AND S2=Ø THEN 5Ø
81 IF S1 AND S2 THEN RETURN
82 S=S1: IF S=Ø THEN S=S2
83 IF S AND 1 THEN Y=Y1+(Y2-Y1)*(XL-X1)/(X2-X1): X=XL: GOTO 87
84 IF S AND 2 THEN Y=Y1+(Y2-Y1)*(XH-X1)/(X2-X1): X=XH: GOTO 87
85 IF S AND 4 THEN X=X1+(X2-X1)*(YH-Y1)/(Y2-Y1): Y=YH: GOTO 87
86 IF S AND 8 THEN X=X1+(X2-X1)*(YL-Y1)/(Y2-Y1): Y=YL
87 IF S=S1 THEN X1=X: Y1=Y: GOTO 71
88 X2=X: Y2=Y: GOTO 71
```

What It Does: This tool will clip each line in the shape being drawn, so that any portion of it not falling on the screen is not plotted.

Example Use: This tool is called with a GOSUB 70 statement after defining and retrieving a shape. This book has the DRAW A SHAPE tool call the CLIP A SHAPE tool automatically. Using this setup, you never have to worry about calling the CLIP A SHAPE tool.

Technical Description: This subroutine will "clip" a shape (one line at a time), until it lies only on the visible screen. The visible screen extends from X = 0 to 319 and Y = 0 to 199. As long as both endpoints of each line in the shape fall within this range, each corresponding line will be on the visible screen.

In the chapters on rotation, scaling, and translation, we will see that it isn't always easy to estimate whether each line in a shape will fall within the visible screen area after the program is run. Therefore, we must check each line in the shape ahead of time, before it is drawn, trimming each one accordingly.

The first step to clipping is to *classify* the endpoints of the line in the shape we are working on. The visible screen area is class 0. If an endpoint is there, then its class is 0 also. If the endpoint is to the left of the visible screen, it will have a class of 1. If it is to the right of the visible screen, it will have a class of 2. There are nine classifications in all, set up like this:

```
         |          |
    9    |    8     |   1Ø
         |          |
---------+----------+---------
         |    Ø     |
    1    |(visible  |    2
         | screen)  |
---------+----------+---------
         |          |
    5    |    4     |    6
         |          |
```

Notice that:

West = 1 (X<0)
East = 2 (X>319)
South = 4 (Y>199)
North = 8 (Y<0)
S.W. = South + West = 5 (X<0, Y>199)
S.E. = South + East = 6 (X>319, Y>199)
N.W. = North + West = 9 (X<0, Y<0)
N.E. = North + East = 10 (X>319, Y<0)

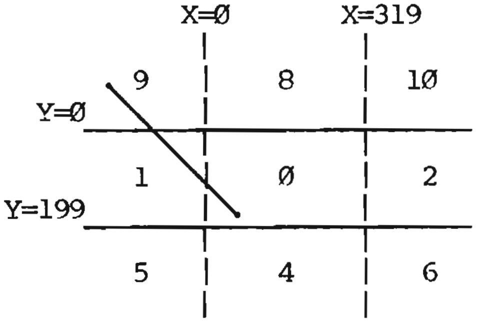
Lines 71-79 in this subroutine will classify each endpoint of the line being clipped.

Once the endpoints are classified, a couple of quick checks need to be performed. First, we must check whether both endpoints are class 0. If so, then the line doesn't need to be clipped: it falls entirely on the visible screen. The line would hence be plotted by the computer and, if you are drawing a shape, the next line to plot is looked at and classified.

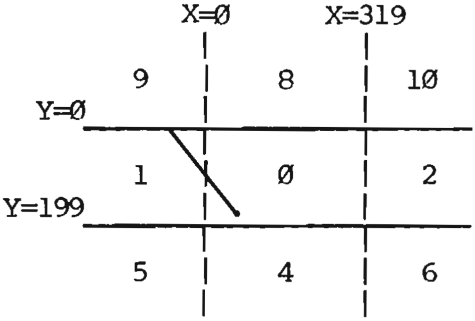
When one or both endpoints of a line do not fall into class 0, a bit of Boolean Logic is used to see if the line falls *entirely* off of the visible screen. Suppose, for example, that one endpoint is in class 5 (SW), and the other is in class 6 (SE). If you look back at the classification diagram, you will see that there is no way a straight line can pass through the visible screen area if its endpoints fall into these classes. This is also the case when trying to connect a line between class 1 and class 9 or between class 2 and 10. When both endpoints are found to lie off the visible screen to start with, the whole line is clipped, and then the next line in the shape is looked at.

There will be times when a line obviously needs to be clipped. Some lines need to be clipped several times before the line falls only on the visible screen. After a line is clipped once, it is sent back to be re-classified with its new endpoints. If the line is still classified outside of area 0, it is re-clipped. This process continues until each endpoint gets a classification of 0, or the line is found to fall entirely off the screen.

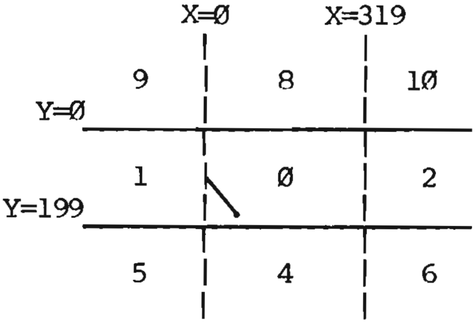
Here is an example of a line to clip:



The first time through the clipping loop, the line is clipped to the Y=0 boundary:



The second time through the loop, the line is clipped to the X=0 boundary, and can be plotted:



Summary

You have already learned in two chapters what might be the entire contents of a beginner's book. That's quite an accomplishment. To your credit, you can now:

—enter and exit the graphics mode with ease
—create both high resolution and multi-color displays
—plot points and lines
—define shapes with data statements
—retrieve shapes for use with other tools
—clip and draw shapes

Some things to keep in mind about your new tools are:

—The DATA statements that define your shape must accurately give the zero-based number of endpoints and number of lines.

—A shape must be retrieved before it can be drawn.

—When retrieving a shape, you must set SE$ to the exact name that the shape was given in the DATA statements.

—When drawing a shape, C must be set to a color code representing the plotting color.

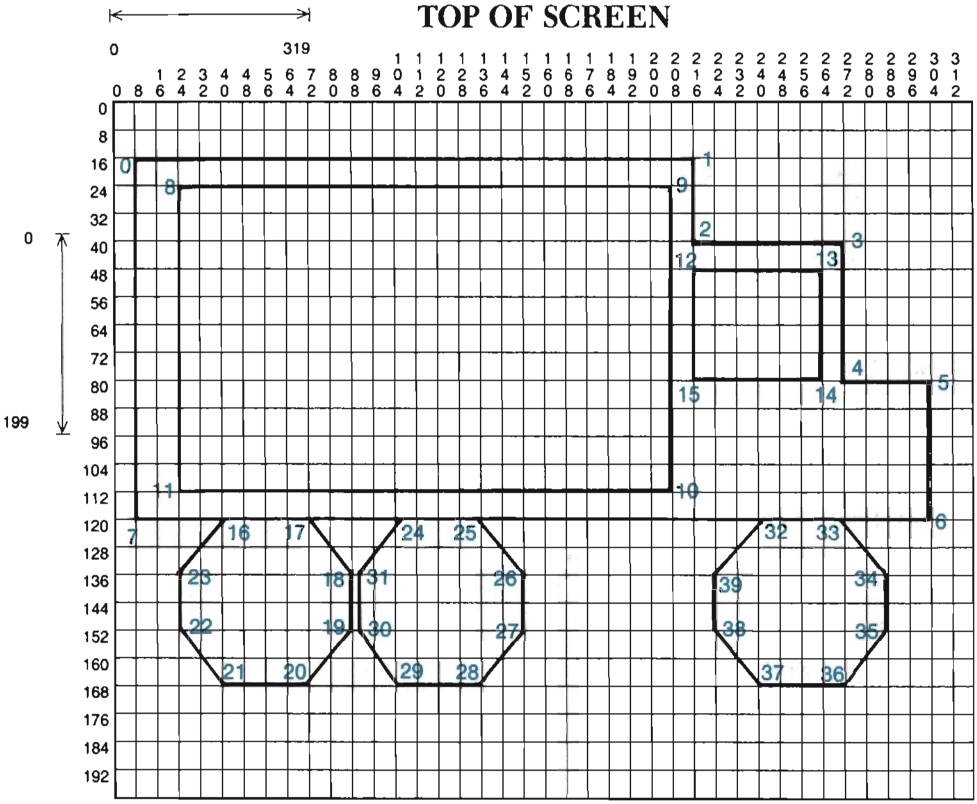
—The program must place the computer in an "endless" loop after drawing a picture in multi-color.

SAVE your program under the filename "CHAPTER 2". Then take the time to completely familiarize yourself with these new tools. To help you along, we have provided an exercise below. Try working through this exercise *after* saving the program, and before beginning Chapter 3.

Exercise

The truck shown earlier will be the shape you will now try to draw. It has been sketched on our X,Y PIXEL POINTS grid below. Beneath the sketch, you are given its Point data list and its Line data list.

Enter the necessary DATA statements to define the truck shape. Begin on line 1100, skipping line numbers as you go (1100, 1102, 1104, etc) to allow room for possible corrections later.



X, Y PIXEL POINTS

In the main routine, change line 2010 so that C is set to 4 (Purple). Change line 2030 to retrieve you new shape. Finally, change line 2040 so that C is set to 1 (white).

Point Data List

	X	Y
Endpt.#Ø	8	16
Endpt.#1	215	16
Endpt.#2	215	4Ø
Endpt.#3	271	4Ø
Endpt.#4	271	8Ø
Endpt.#5	3Ø3	8Ø
Endpt.#6	3Ø3	119
Endpt.#7	8	119
Endpt.#8	24	24
Endpt.#9	2Ø7	24
Endpt.#1Ø	2Ø7	111
Endpt.#11	24	111
Endpt.#12	216	48
Endpt.#13	263	48
Endpt.#14	263	79
Endpt.#15	216	79
Endpt.#16	4Ø	12Ø
Endpt.#17	71	12Ø
Endpt.#18	87	135
Endpt.#19	87	151
Endpt.#2Ø	71	167
Endpt.#21	4Ø	167
Endpt.#22	24	151
Endpt.#23	24	135
Endpt.#24	1Ø4	12Ø
Endpt.#25	135	12Ø
Endpt.#26	151	135
Endpt.#27	151	151
Endpt.#28	135	167
Endpt.#29	1Ø4	167
Endpt.#3Ø	88	151
Endpt.#31	88	135
Endpt.#32	24Ø	12Ø
Endpt.#33	271	12Ø
Endpt.#34	287	135
Endpt.#35	287	151
Endpt.#36	271	167
Endpt.#37	24Ø	167
Endpt.#38	224	151
Endpt.#39	224	135

Line Data List

	FROM	TO
Line#Ø	Ø	1
Line#1	1	2
Line#2	2	3
Line#3	3	4
Line#4	4	5
Line#5	5	6
Line#6	6	7
Line#7	7	Ø
Line#8	8	9
Line#9	9	1Ø
Line#1Ø	1Ø	11
Line#11	11	8
Line#12	12	13
Line#13	13	14
Line#14	14	15
Line#15	15	12
Line#16	16	17
Line#17	17	18
Line#18	18	19
Line#19	19	2Ø
Line#2Ø	2Ø	21
Line#21	21	22
Line#22	22	23
Line#23	23	16
Line#24	24	25
Line#25	25	26
Line#26	26	27
Line#27	27	28
Line#28	28	29
Line#29	3Ø	31
Line#3Ø	31	24
Line#31	32	33
Line#32	33	34
Line#33	34	35
Line#34	35	36
Line#35	36	37
Line#36	37	38
Line#37	38	39
Line#38	39	32
Line#39	29	3Ø

The solution to this exercise is shown below.

Solution

```
11ØØ DATA "TRUCK", 39, 39
11Ø2 DATA   8, 16,215, 16,215, 4Ø
11Ø4 DATA 271, 4Ø,271, 8Ø,3Ø3, 8Ø
11Ø6 DATA 3Ø3,119,  8,119, 24, 24
11Ø8 DATA 2Ø7, 24,2Ø7,111, 24,111
111Ø DATA 216, 48,263, 48,263, 79
1112 DATA 216, 79, 4Ø,12Ø, 71,12Ø
1114 DATA  87,135, 87,151, 71,167
1116 DATA  4Ø,167, 24,151, 24,135
1118 DATA 1Ø4,12Ø,135,12Ø,151,135
112Ø DATA 151,151,135,167,1Ø4,167
1122 DATA  88,151, 88,135,24Ø,12Ø
1124 DATA 271,12Ø,287,135,287,151
1126 DATA 271,167,24Ø,167,224,151
1128 DATA 224,135
113Ø DATA  Ø, 1, 1, 2, 2, 3, 3, 4
1132 DATA  4, 5, 5, 6, 6, 7, 7, Ø
1134 DATA  8, 9, 9,1Ø,1Ø,11,11, 8
1136 DATA 12,13,13,14,14,15,15,12
1138 DATA 16,17,17,18,18,19,19,2Ø
114Ø DATA 2Ø,21,21,22,22,23,23,16
1142 DATA 24,25,25,26,26,27,27,28
1144 DATA 28,29,3Ø,31,31,24,32,33
1146 DATA 33,34,34,35,35,36,36,37
1148 DATA 37,38,38,39,39,32,29,3Ø
2Ø1Ø GOSUB 1Ø: C=4: GOSUB 3Ø
2Ø2Ø DIM P%(99,2), L%(99,1)
2Ø3Ø SE$="TRUCK": GOSUB 8ØØ
2Ø4Ø C=1: GOSUB 9Ø
6ØØØ GET A$
6Ø1Ø IF A$="←—" THEN GOSUB 2Ø:END
6Ø2Ø GOTO 6ØØØ
```

Chapter Three

PAINTING SHAPES

Bright, Vivid, Gorgeous, Rich, Exotic, Florid, Rainbowy, Kaleidoscopic. These are just some of the synonyms you can find listed in your thesaurus for the word "colorful."

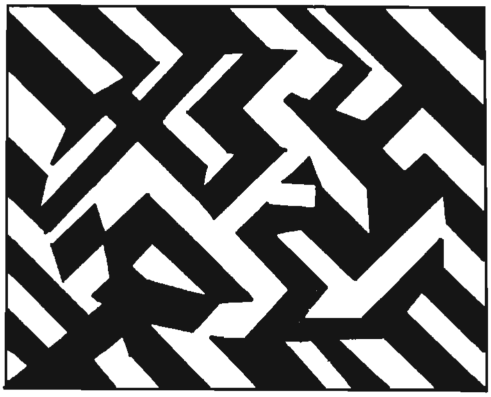
Faded, Washed-Out, Dull, Toneless, Weak, Wan, Fallow, Ashen, Sickly. These are synonyms that can be found for the word "colorless."

Notice how your mind reacts to the different lists of adjectives. Words like Bright, Vivid and Rich can actually perk you up, producing a lively, energetic feeling. On the other hand, reading words like Fallow, Ashen, and Sickly produces a somewhat hollow, dismal feeling. In the same way, colors (or the lack thereof) have a strong impact on what people feel when viewing your work.

There is no set of rules regarding what colors will produce what effect and when. It usually depends both on the design itself and the person viewing it. One person might view a picture full of browns and yellows as dreary. Someone else might associate those same colors with the outdoors—light, sunny, airy. The point is that colors *do* matter.

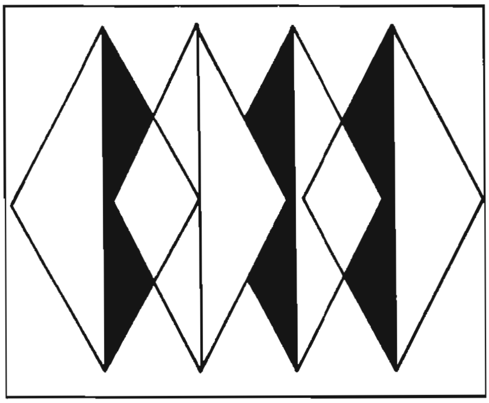
If you have painted shapes before, you have already seen the pleasing change that takes place as each color is added to the picture. Because your Commodore has a 16-color display capability, you can experiment with a large assortment of color combinations before carefully choosing those that appeal to you most.

If you only have a black and white monitor, don't give up on this chapter yet. A change in tone alone can significantly enhance the appearance of any design. Take the sketch below:



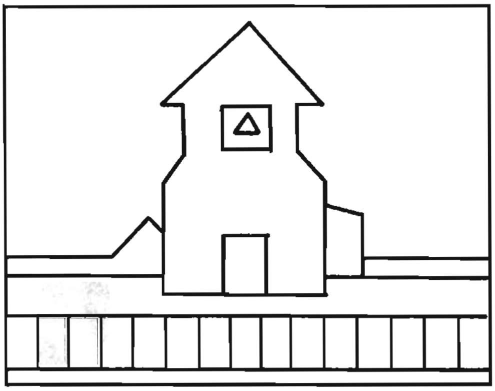
A dark tone was applied to all the varied shapes in this sketch. Observe how these dark toned shapes were arranged and organized into a pattern. Although the individual shapes themselves are very different, the tone and pattern creates "order." This order also results from the balance between the light and dark shapes.

In the next drawing, notice how the smaller triangles have been pronounced by their darker tones. This helps distinguish them from the rest of the picture, adding both depth and variety to the design.

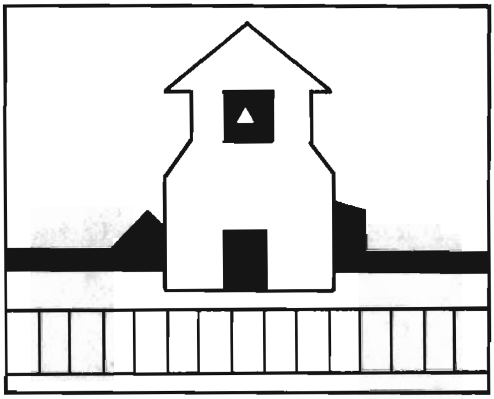


Look once again at the above sketch. Notice that the overlapping pattern is not consistent from one shape to the next. This tiny shift from the expected is another, more subtle method of adding variation to a design.

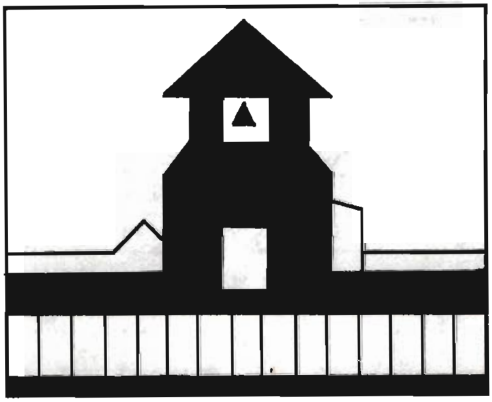
Finally, *contrast* can be used to create different visual results. The following picture of a schoolhouse is constructed of many lines and shapes. The next step would be to fill these shapes in with color. When coloring shapes, you should give careful consideration to what you desire as a final image.



In the design below, most of the shapes have been painted white.



A totally new design can be achieved by reversing the white and black areas:



Compare the results of the two sketches above. The first might represent the schoolhouse during the day, while the second might be a night scene. Or perhaps they are simply two different schoolhouses. Notice that certain shapes stand out with some colors, and recede into the distance with other colors. One thing is clear: the use of contrast can dramatically alter a drawing.

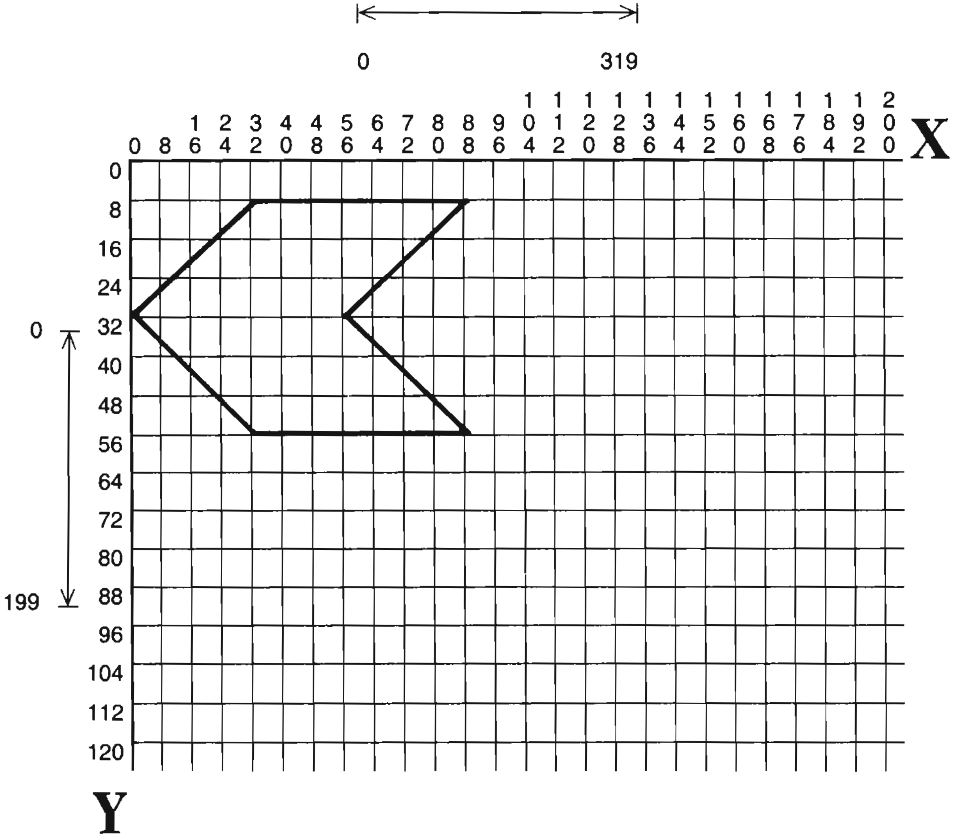
With a little imagination (and a PAINT A SHAPE tool), you will soon be fashioning your *own* exotic, florid, and rainbowy displays!

Painting Simple Shapes

Load the "CHAPTER 2" program into memory. Next, add the PAINT A SHAPE subroutine to your tool kit:

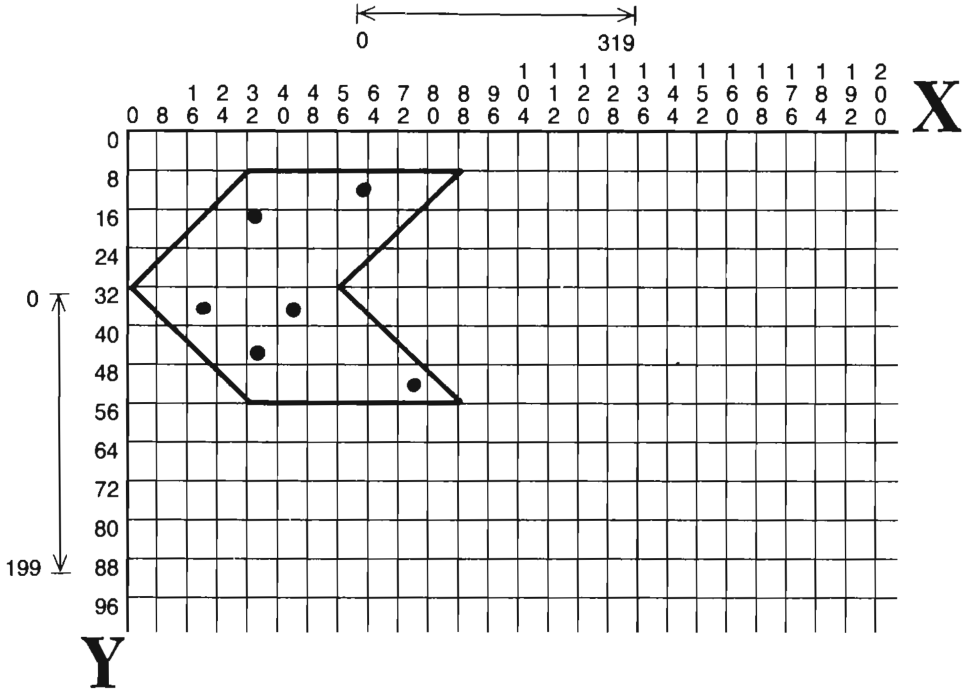
```
6Ø REM::::::PAINT A SHAPE
62 SYS 49748,X,Y,C,MU
63 RETURN
```

A shape is made up of three or more plotted lines. These lines connect together at each end, forming an enclosed outline. Below is six-sided outline shape.

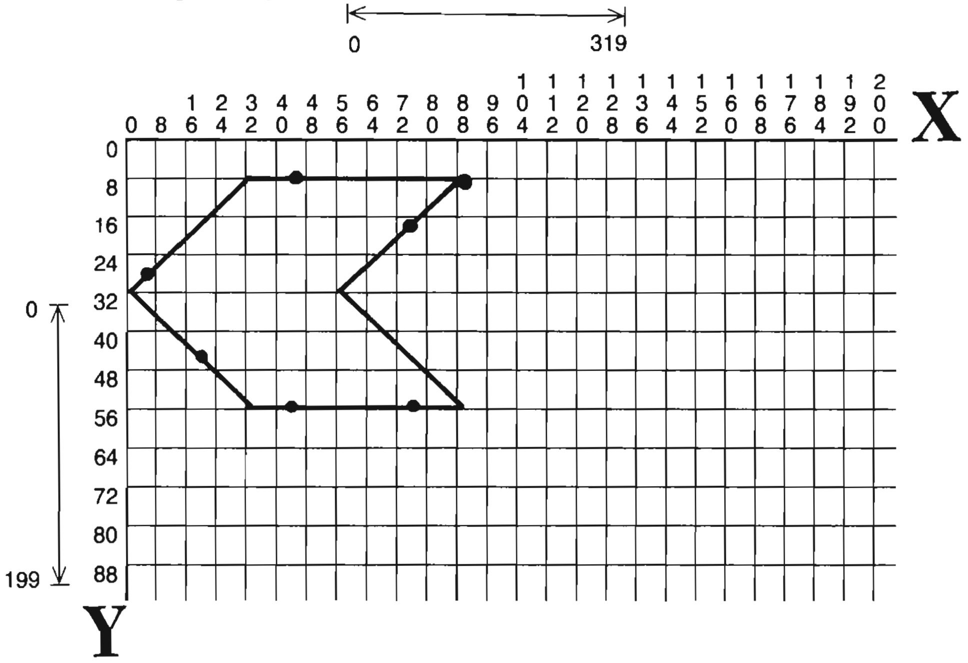


The computer needs to know *where* the inside of this outline shape is in order to paint it. A computer is not as smart as some might think. It has no "eyes," and cannot look at the shape to find the inside area. You must direct the computer there. By giving the X,Y location of *one* pixel point inside the outline, you position the computer in the area to paint.

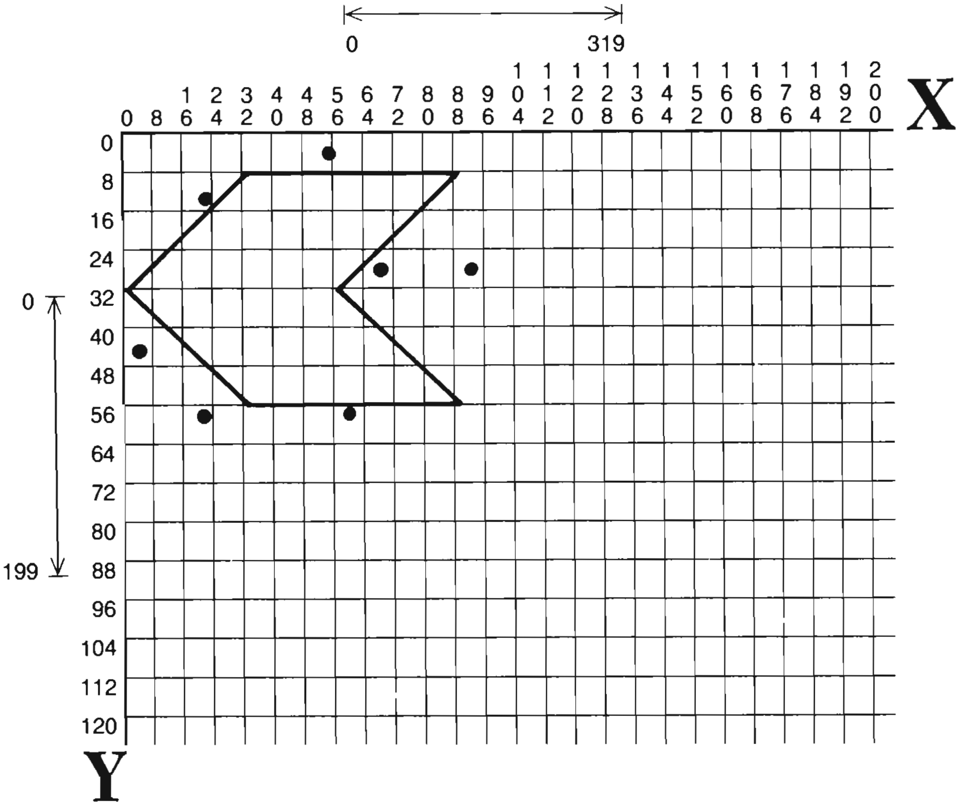
In our six-sided shape, there are many pixel points that fall inside the outline. Although only one needs to be identified for painting purposes, several are shown below:



A pixel point that falls *on* the outline instead of inside the outline should not be identified for painting:



Finally, identifying a pixel point that falls *outside* the outline will only serve to paint the outside area of the shape:



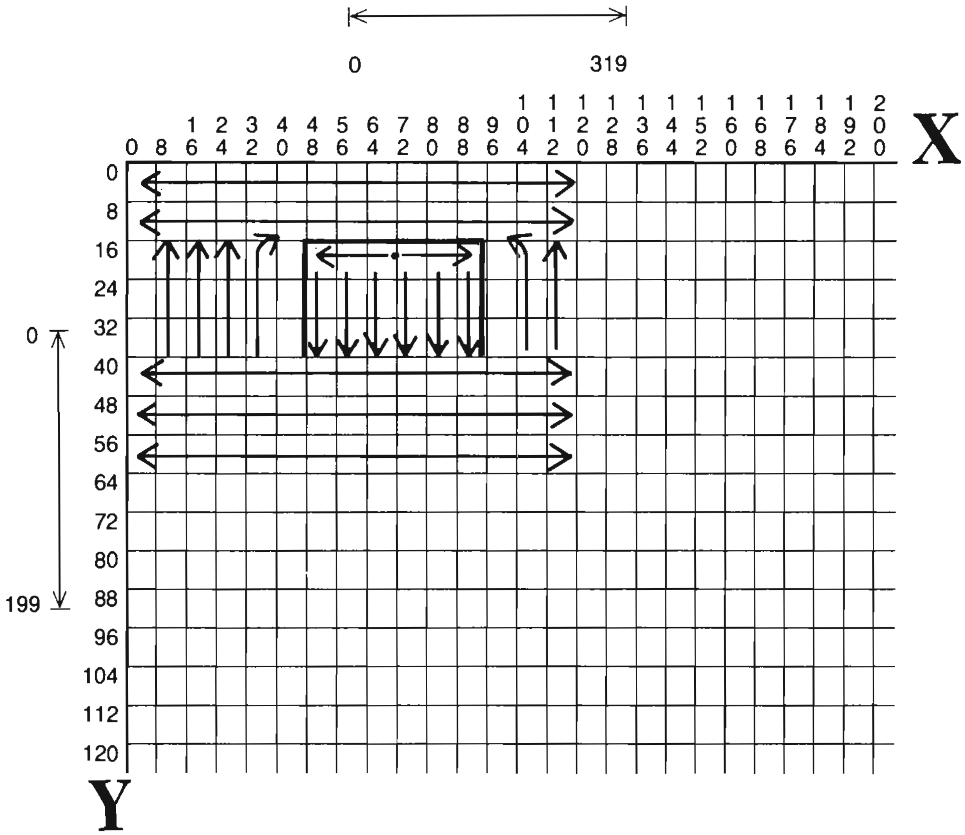
Each pixel that falls inside your shape is a "paint point." It is important that you correctly identify one of your shape's paint points prior to calling Tool 60.

Once inside the shape, your new tool can begin painting every background pixel it contacts. (Note: If you start it off on a foreground pixel, it won't paint anything— it will think it is done.) The computer paints in every direction. It will only quit painting in a certain direction when one of two things happens:

(1) The edge of the screen is reached.

(2) A foreground pixel is reached.

Think of this as if you are painting with buckets of paint. The outline shape acts as a barrier or wall that will contain each color spilled inside it. If the outline is not completely enclosed, the paint will run out into the surrounding areas. In the diagram below, the small dot is the X,Y position where the paint originally gets "spilled." The arrows show how the PAINT A SHAPE tool paints inside, and then also outside the shape. Just as real paint flows outside of an open container, so too will your painting tool.



That is basically all there is to painting shapes. In terms of programming, the main routine must:

(1) RETRIEVE the shape (SE$="?": GOSUB 800).

(2) DRAW the shape (C=?:GOSUB 90).

(3) Set the color variable to the desired painting color (C=?).

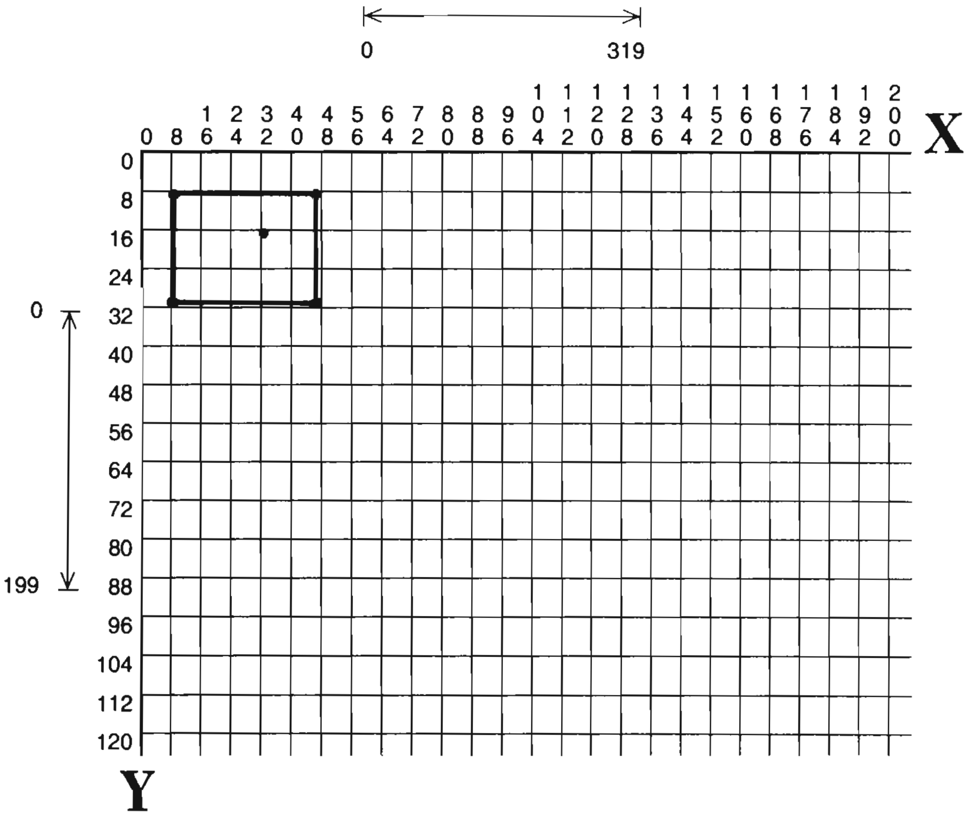
(4) Give the X,Y location of one paint point *within* the shape to paint (X=?:Y=?).

(5) Call the PAINT A SHAPE tool (GOSUB 60).

Let's apply these five steps to the RECTANGLE stored in your library. Change the main routine as follows:

```
2Ø1Ø GOSUB 1Ø:C=14: GOSUB 3Ø
2Ø3Ø SE$="RECTANGLE": GOSUB 8ØØ
2Ø4Ø C=1Ø: GOSUB 9Ø
```

The rectangle falls on the screen as shown below. Based on this, a paint point has been selected.



Using the X,Y coordinates of the paint point, you can paint the rectangle. Add the following to your program:

```
2Ø5Ø X=32:Y=16: GOSUB 6Ø
```

RUN the program and watch how the shape is painted. Painting starts on the row where the paint point is located, and flows to the right until a "barrier" (outline) is met. The computer then moves into the next row and paints it. The painting stops when all background pixels within the shape have been painted.

Press ← to return to text mode. We can put a hole in the outline shape by making a couple of small changes to this shape's data. Change lines 1006, 1008 and 1010 to the following:

```
1ØØ6 DATA "RECTANGLE",4,3
1ØØ8 DATA  8, 8,47, 8,47,2Ø,47,31, 8,31
1Ø1Ø DATA  Ø, 1, 1, 2, 3, 4, 4, Ø
```

This alters the shape's description: the bottom right-hand corner of the rectangle now is not closed.

RUN the program. It's amazing the effect such a small change can have. Don't hold your breath to see how this one turns out. The computer will continue painting until the entire screen area is painted red. Press **RUN/STOP** and tap **RESTORE** to break out of the program. Now, change lines 1006-1010 back to their original contents:

```
1ØØ6 DATA "RECTANGE",3,3,
1ØØ8 DATA  8, 8,47, 8,47,31, 8,31
1Ø1Ø DATA  Ø, 1, 1, 2, 2, 3, 3, Ø
```

Try another shape. Change lines 2030 and 2040 so that the INV-TRIANGLES shape is retrieved and drawn in black:

```
2Ø3Ø SE$="INV=TRIANGLES": GOSUB 8ØØ
2Ø4Ø C=Ø: GOSUB 9Ø
```

Lines 2050 and 2060 below will paint each side of this shape a different color:

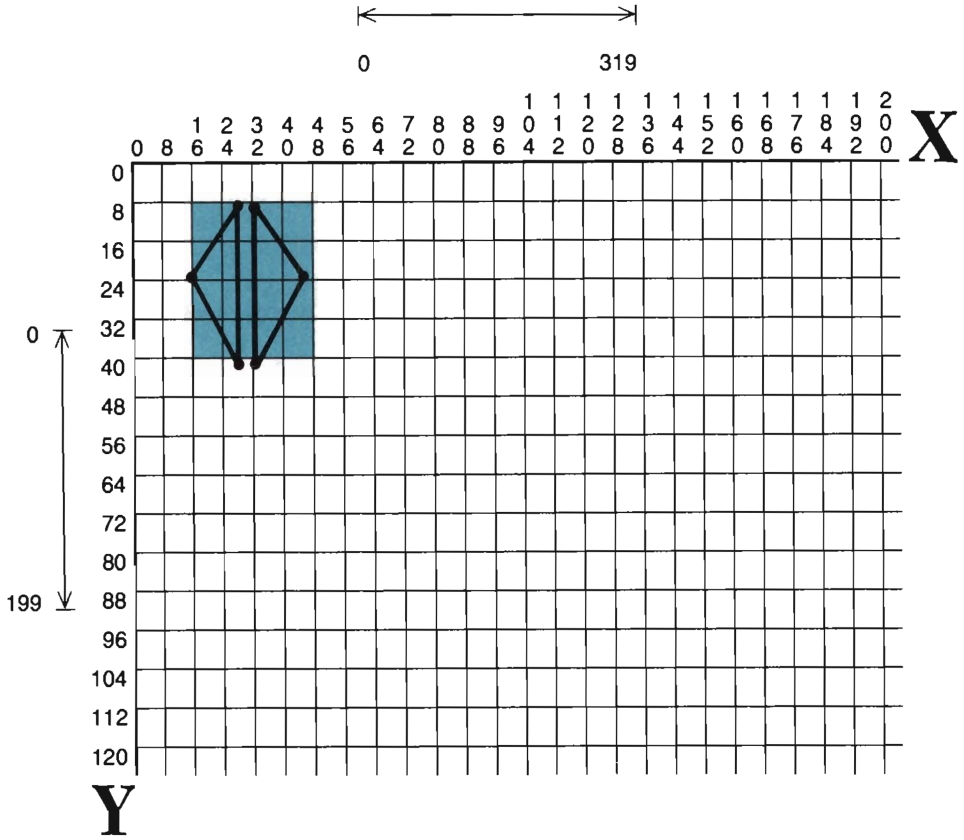
```
2Ø5Ø X=28: Y=2Ø: C=2: GOSUB 6Ø
2Ø6Ø X=34: Y=2Ø: C=8: GOSUB 6Ø
```

Again, RUN your program. The shape will be drawn, and then each side painted. The left side should be painted in red, and the right side in orange. These colors go particularly well together, and the light blue provides a nice background for them.

Before you stop this program, look closely at the bottom two points of these triangles. They remain black (the drawing color) instead of being painted red and orange (the painting colors). Why?

The PAINT A SHAPE tool only paints up to a shape's outline pixels, without directly painting the outline itself. However, if an outline pixel is located in a color block where painting takes place, it will automatically change to the new foreground color.

Look at the diagram of our design below. Each painted color block has been shaded. Remember that only one foreground color can be put in each block. When the computer tried to paint a red foreground color in a block on the left, it ran into a problem: there were already some black outline (foreground) pixels in the block. To eliminate this problem, it simply changed the black pixels to the new foreground color of red. That, briefly, is how most of the outline pixels changed to the appropriate color.



Because the two bottom points are outside of any painted color block, they remained black after the shape was painted. There are two possible remedies to this problem. Press ← to learn what they are.

One way is to change the color the shape gets drawn in (line 2040). This works for a solid shape that only needs to be painted one color. You simply change line 2040 so that you outline the shape in the same color you plan to paint it.

Because ours is not a solid, single-colored shape, we must use a second method. We need to move those bottom points so that they fall in a color block where painting occurs. Change lines 1022 and 1024 so that the Y location of those points is moved up from row 40 to row 39:

```
1Ø22 DATA 3Ø, 8,3Ø,39,16,23
1Ø24 DATA 33, 8,33,39,47,23
```

RUN the program to see how well this "quick fix" worked.

Color block problems will be your biggest obstacle if not taken into account during the design stage. Some simple guidelines for preparing to draw a picture on the Commodore are:

(1) Sketch the entire picture on graph paper, similar to our X,Y PIXEL POINTS grid (see appendices for a copy).

(2) Use light colored pens or pencils to shade in the desired colors.

(3) Search each color block (grid block) for more than one foreground color. (If you are going to be using multi-color, look for more than three foreground colors per block.)

You will have to adjust the picture in some way if you find color blocks with too many foreground colors. This typically involves moving a shape up, down, right or left, or adjusting its size.

You might spend some time practicing with your new tool before going on to the next section.

Painting in Multi-Color

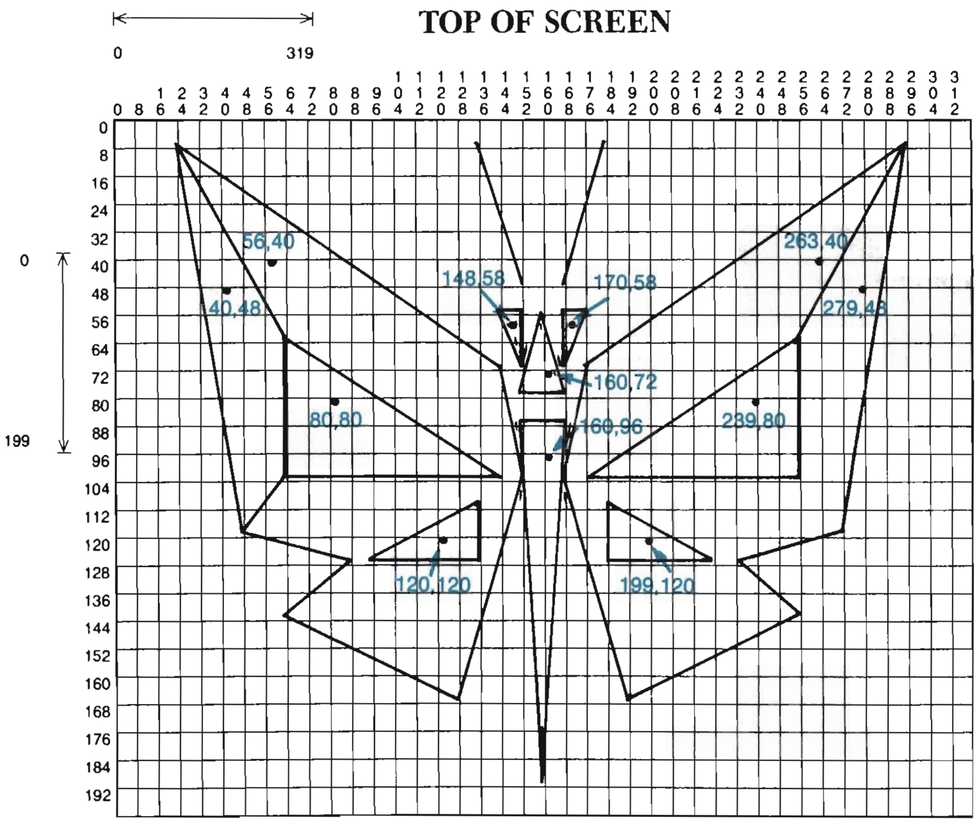
To use multi-color, you only need to set a single variable (MU). The difficulty, however, lies in determining when to use multi-color, as well as anticipating how it might affect your design.

Coincidentally, you have a very good example design to work with—the butterfly. The butterfly has many angles that cut through color blocks, thus causing problems in high resolution.

Change line 2030 to retrieve the BUTTERFLY:

```
2030 SE$="BUTTERFLY": GOSUB 800
```

The butterfly design is again shown below. In each area to paint, a paint point has been identified.



X, Y PIXEL POINTS

These paint points can be easily transferred to the program in the form of X,Y locations (see program listing below). After each new paint point has been established, a GOSUB 60 calls the paint routine. The color variable (C) is set to a new color code whenever necessary. Add these lines to your program now:

```
2Ø5Ø X=56: Y=4Ø: C=12: GOSUB 6Ø
2Ø6Ø X=263: GOSUB 6Ø
2Ø7Ø X=4Ø: Y=48: GOSUB 6Ø
2Ø8Ø X=279: GOSUB 6Ø
2Ø9Ø X=8Ø: Y=8Ø: C=8: GOSUB 6Ø
21ØØ X=239: GOSUB 6Ø
211Ø X=12Ø: Y=12Ø: GOSUB 6Ø
212Ø X=199: GOSUB 6Ø
213Ø X=16Ø: Y=96: C=Ø: GOSUB 6Ø
214Ø Y=72: GOSUB 6Ø
215Ø X=148: Y=58: C=2: GOSUB 6Ø
216Ø X=17Ø: GOSUB 6Ø
```

Notice that whenever the Y location remains constant from one paint point to the next, Y does not have to be set the second time. For example, the first paint point identified in the program is 56,40 (X=56:Y=40). The next paint point identified is 263,40 (X=253:Y=40). Since Y does not change from the first paint point to the second, Y is only set the first time. This same idea applies to the X location, which remains constant from line 2130 to line 2140.

RUN this program in high resolution to see what happens. Wait as the entire shape fills in with various colors. When the butterfly's eyes fill in with red, the painting is complete.

Beautiful? Hardly. Two obvious problems exist with this design painted in high resolution. First, bits and pieces of the outline weren't painted. Unfortunately, there is no "quick fix" for this one. You can't draw the butterfly in the painting color because there's more than one painting color. You can't move the bits and pieces inside painted color blocks, because there's no easy way to figure out the X,Y locations of those bits and pieces.

Another problem (again related to color blocks) is the chunky, block-like appearance of the orange wing areas. This problem results from trying to plot two colors (beige and orange) within the same color blocks. Notice that this problem occurs only where there are sloping lines.

Press ← to get back to the program. Change line 2010 so that MU equals 1 before calling the graphics tool:

```
2010 MU=1: GOSUB 10: C=14: GOSUB 30
```

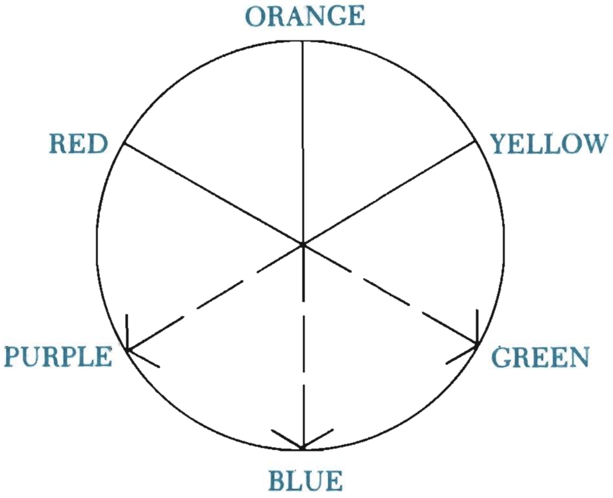
RUN the program to see if this helped. Beautiful? This time, yes. The colors all fall into place in multi-color. You've lost some vertical resolution, but gained much more in return.

There is still one problem, however, not taken care of by multi-color: the outline. Note that it is no longer randomly painted here and there; instead, it isn't painted at all. This is because multi-color permits three foreground colors in each color block. Since the design never violates this limit, all colors are used exactly where they are put.

It is not always undesirable to have a shape outlined in black, but often it is. The only way to get around this problem is to store each section of the design (e.g., each area to paint) as a shape by itself. This allows you to retrieve the section, draw it in its painting color, and then paint it in the same color. In the case of the butterfly, for example, the orange triangle on the top left side could be stored as a shape by itself. Later, it could be retrieved, drawn in orange, and then painted in orange. If this were done for each shape within the butterfly, you solve the outline problem.

We are not going to take the time here to section-out the butterfly. The important thing is to understand how multi-color can be of help, and where it can present problems. As we've mentioned, color blocks can be your greatest single design problem—especially where sloping lines are concerned.

Keep the butterfly on your screen, and examine for a moment the paint colors we chose. The orange is from the warm family of colors, while the blue is from the cool family of colors. If you look at the chart below, you will see that these two colors are directly opposite each other on the color wheel. Opposite colors are called "complements." Usually, when two complementary colors are used in equal proportions, they cancel each other out—producing a calmer looking design. Although we have not used these two colors in equal proportions, we have inserted a neutral color (beige) between the two. This heightens the neutrality of the design, again producing a greater calmness.



Warm and Cool Complements

Red and Green
Orange and Blue
Yellow and Purple

Press ← to return to text mode. Next, change the following two lines in your program and RUN it:

```
2040 C=5: GOSUB 90
2130 X=160: Y=96: C=5: GOSUB 60
```

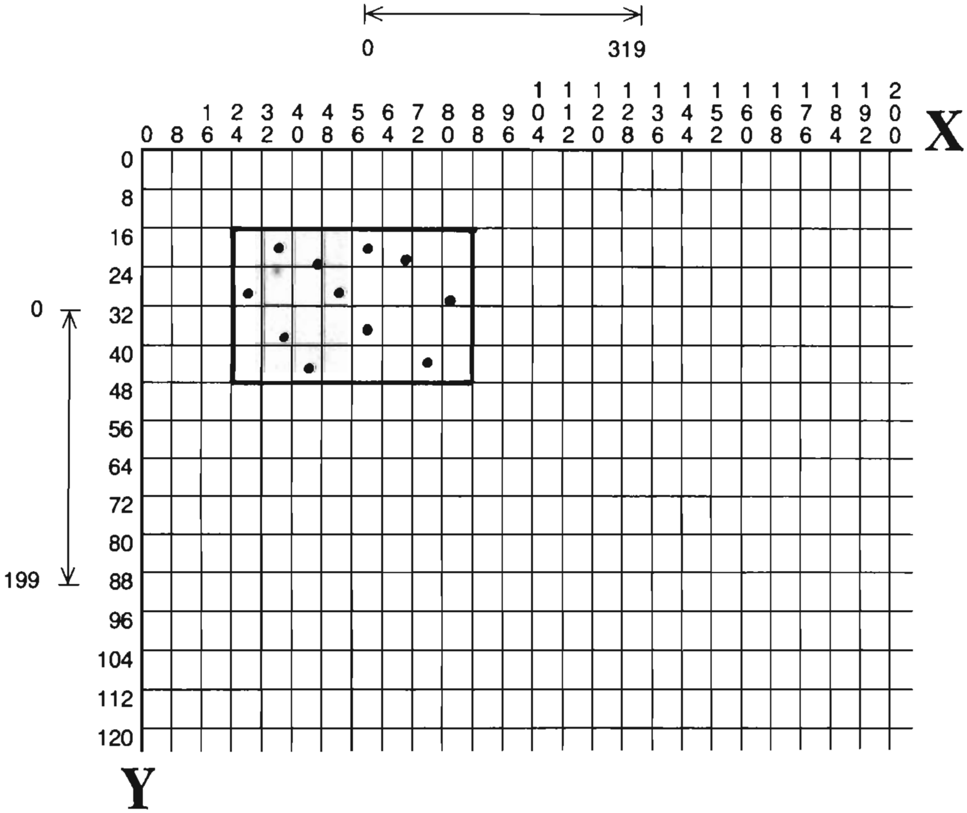
Not exactly something to hang on your wall, is it? Several problems can be found with this color selection. First, the green blends and meshes too much with the background color. A haziness appears that almost strains the eyes; it is compunded by the overall brightness of the colors. The orange color now dominates the screen, and makes the triangles appear on top of the wings instead of as a part of them. Finally, the eyes are being unnecessarily emphasized with the red color.

Choosing an attractive color combination will often be a matter of trying out several on the screen to see how they look. Unpleasant surprises, like that on your screen now, are bound to occur. If you have worked with colors for a long time, you may be able to visualize them before you put them on the screen. For many people, however, much of the selection will be through the process of trial and error.

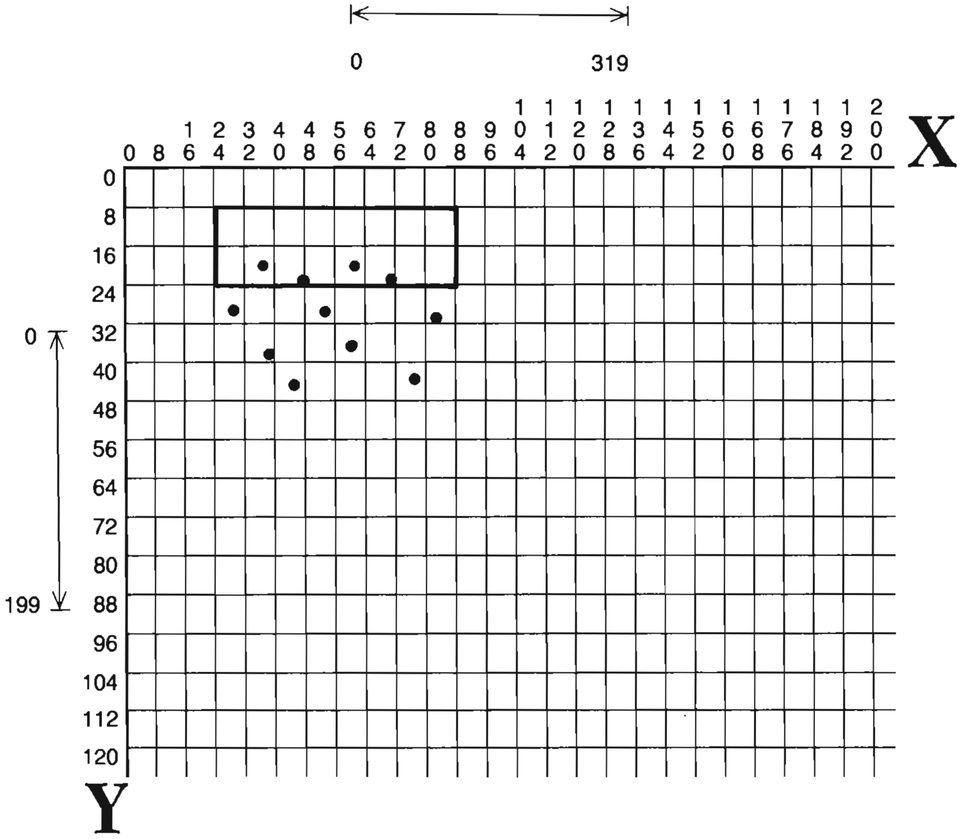
We suggest trying some of your own color combinations now. Try a blue, purple and cyan combination. Work with the shape's outline. (The color chart is listed in the appendices for quick reference.) As you discover combinations that you like, try to analyze why they appeal to you. Think, too, about the combinations you don't like. Through experience, you will begin to remember those combinations that particularly appeal to your tastes.

Storing Paint Points

You have learned what a paint point is, and how to identify it in the program (X=?:Y=?). This relatively simple task enables you to paint the inside area of a plotted shape. This is not the end to your painting lesson, though. In fact, if you skipped ahead to any other chapter, you would soon find yourself at a loss concerning painting. To see what we mean, look at some of the paint points in this rectangle:



Now, consider the upcoming chapter on scaling. In it, you will learn of a tool that can automatically scale a shape. Without going into detail, let us show you what happens when you scale this rectangle to half its height:



Notice that most of the original paint points now fall outside the shape—rendering them useless for painting purposes. Whereas the original rectangle could be painted by identifying the point at 82,42, the scaled rectangle cannot.

In this particular example, a paint point still can be easily identified. However, it won't be easy to do so in every case. For example, consider scaling the butterfly to half its height. It's difficult to visualize exactly how the butterfly would fall on the screen after it was scaled. Even harder to judge is where the new paint points would be located.

The solution is to define each paint point within the shape's DATA statements. As part of the shape's description, the paint points will be scaled, rotated, and moved with the rest of the shape.

A simple exercise shows how this works. Let's draw and paint an octagon. Take the time, if you like, to SAVE your current program under any file name *except* CHAPTER 3. Then, change line 172 to the following:

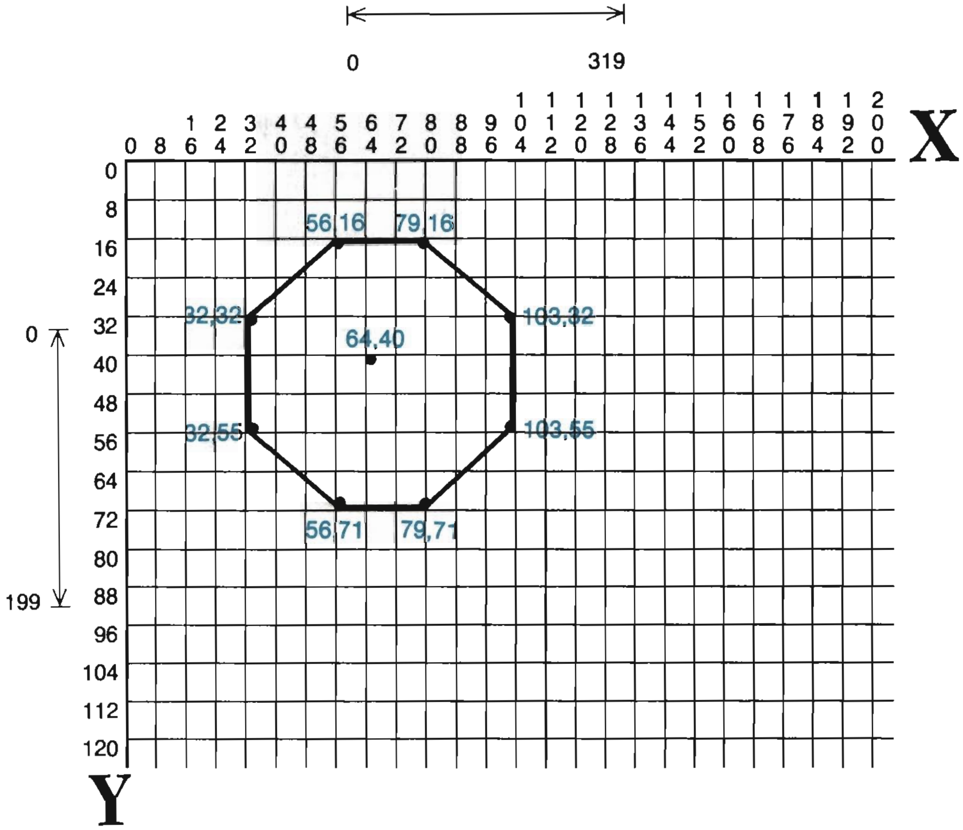
```
172 A=256: B=2049: C=1006
```

We are going to ZAP the Shape Library because it has become too bulky and cumbersome to use. Move the cursor to a free, blank line and type:

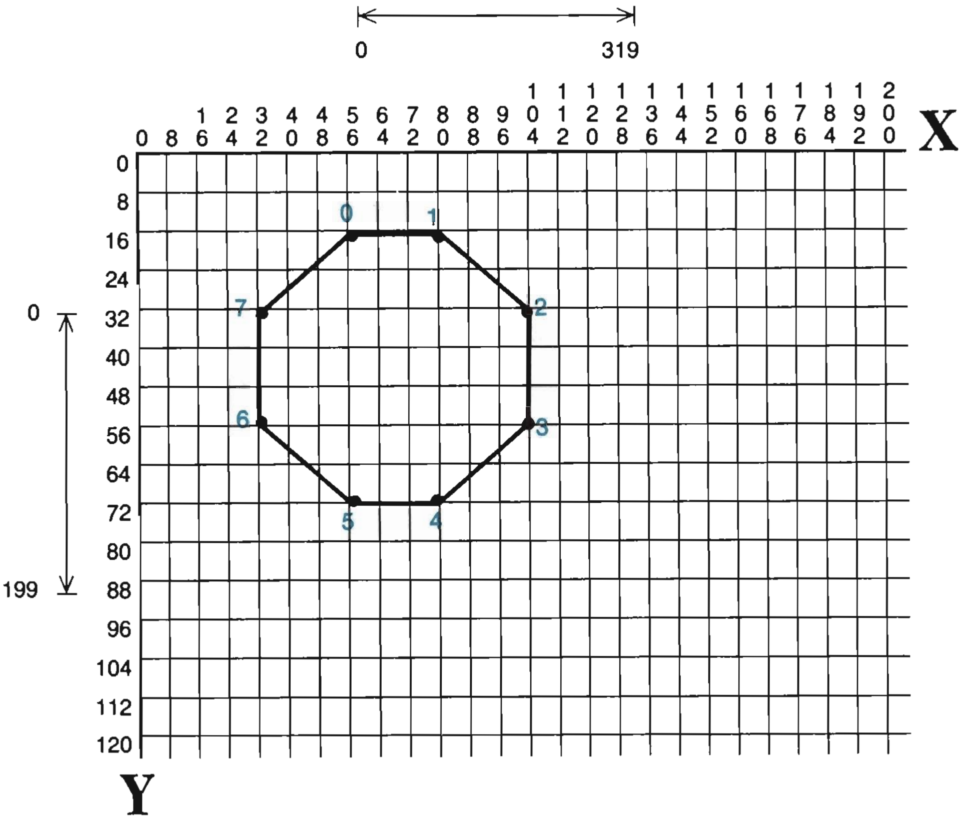
```
RUN 172
```

Press **RETURN**. When the zapping stops and the cursor reappears, LIST your program. Make sure that only the program lines after 1004 have been deleted.

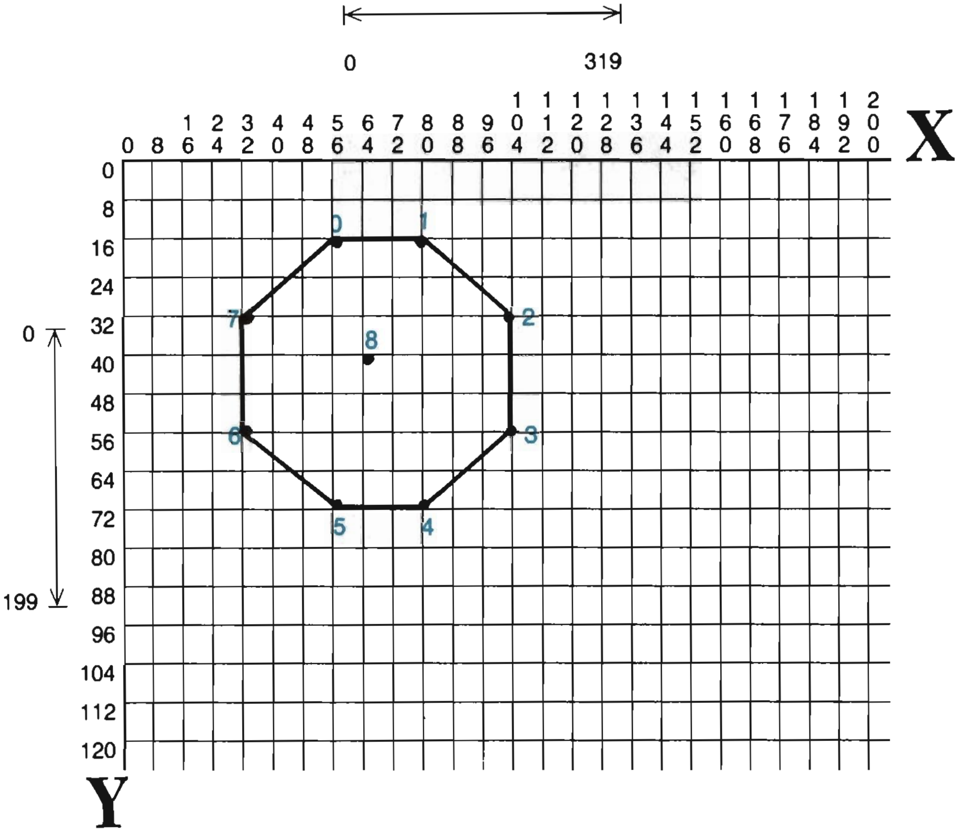
As you learned in the last chapter, the first step in drawing a shape is to sketch it on an X,Y PIXEL POINTS grid, noting the coordinates of each endpoint. This is still necessary, but you must now also identify each paint point as well:



The next step is to number each endpoint, starting with #0:



Once *all* endpoints have been numbered, you then number the paint points. Your paint points will be stored in the same array as your endpoints (P%), and so they are numbered together with the endpoints. Since the octagon's endpoints are numbered 0 through 7, its first and only paint point is numbered 8:



With this information, you can write both the Point data list and the Line data list:

Point Data List	X	Y
Endpt #0	56	16
Endpt #1	79	16
Endpt #2	103	32
Endpt #3	103	55
Endpt #4	79	71
Endpt #5	56	71
Endpt #6	32	55
Endpt #7	32	32
Endpt #8	64	40

Line Data List	"FROM"	"TO"
Line #0	0	1
Line #1	1	2
Line #2	2	3
Line #3	3	4
Line #4	4	5
Line #5	5	6
Line #6	6	7
Line #7	7	0

You can see that the Point data list includes both endpoints and paint points, all in consecutive order. The Line data list never uses point #8, so that point is free to be used as a paint point.

To translate all of this into program lines, type the following:

```
1ØØ5 DIM P%(99,2), L%(99,1)
1ØØ6 DATA "OCTAGON",8,7
```

Line 1005 dimensions P% and L%. (This DIM statement can appear anywhere in the program, provided it appears before the first GOSUB 800). The next line assigns a name to the shape: "OCTAGON". In addition, the zero-based count of points (endpoints *and* paint points), and the zero-based count of lines is given.

Next, enter the Point data:

```
1ØØ8 DATA  56, 16, 79, 16,1Ø3, 32
1Ø1Ø DATA 1Ø3, 55, 79, 71, 56, 71
1Ø12 DATA  32, 55, 32, 32, 64, 4Ø
```

Enter this data exactly as you are accustomed. The X,Y coordinates of Endpoint #0 start the DATA statements off. Then Endpoint #1's coordinates, Endpoint #2's coordinates, etc. Because these DATA statements must include all points, Paint Point #8's coordinates follow Endpoint #7's coordinates. That completes the first set of data.

The Line data is entered in the usual fashion:

```
1Ø14 DATA  Ø, 1, 1, 2, 2, 3, 3, 4
1Ø16 DATA  4, 5, 5, 6, 6, 7, 7, Ø
```

To draw and paint this shape, enter the following main routine lines:

```
2ØØØ GOSUB 1Ø: C=14: GOSUB 3Ø
2Ø1Ø SE$="OCTAGON": GOSUB 8ØØ
2Ø2Ø C=6: GOSUB 9Ø
2Ø3Ø PP=8: GOSUB 6Ø
6ØØØ GET A$
6Ø1Ø IF A$ = "←—" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
```

The shape is retrieved and drawn with Tools 800 and 90. Next, it is painted. However, instead of specifying an X,Y location inside the octagon, specify the number of the paint point as found on your point list (PP=8).

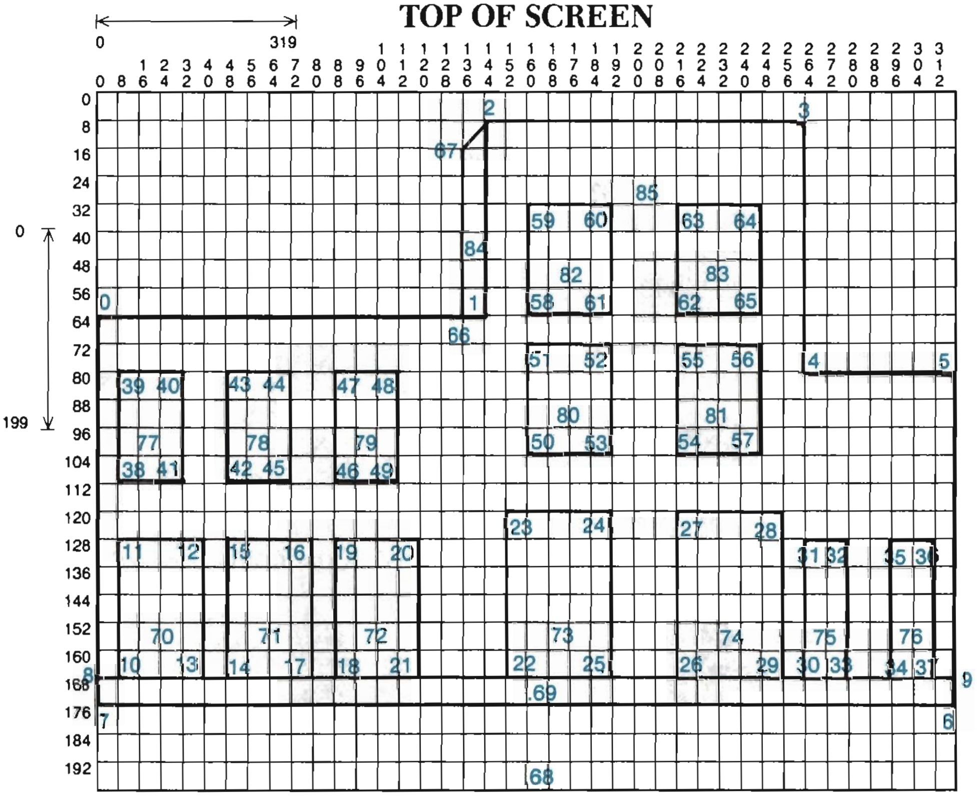
You need to modify the PAINT A SHAPE tool to take advantage of this new concept. Add line 61 as follows:

```
61 X=P%(PP,Ø): Y=P%(PP,1): IF X<Ø OR Y<Ø OR X>319 OR
   Y>199 THEN RETURN
```

This program will change the graphics background screen to light blue, so you must first change the text color to white (press **CTRL** and **2** at the same time). Then, RUN the program to see the modified tool at work.

Pre-defined paint points do not have to be listed at the end of the point data. For example, the sixth coordinate set could identify our shape's paint point, with the last two endpoint coordinates *following* it. Setting PP to 6 would then correctly find the paint point when the time came to paint the shape. You will make it easier on yourself if you keep your endpoints grouped together and your paint points grouped together. Stick to the convention of endpoints, followed by paint points.

This small exercise did not discuss shapes having more than one paint point, so let's try one more example:



X, Y PIXEL POINTS

This design is made up of 68 endpoints, 18 paint points, and 60 lines. Notice that we have numbered the endpoints from 0 through 67, and the paint points from 68 through 85.

ZAP your main routine in order to begin fresh (type RUN 172 and press **RETURN**), and then add these lines:

```
1008 DATA "BUILDING", 85, 59
```

The 68 endpoints, with 18 paint points added to it, gives a true point count of 86. Converting this to a zero-based count, we arrive at 85. The 60 lines renders a zero-based count of 59. Thus the point and line count for our "BUILDING" shape is 85,59.

Now enter the endpoint data:

```
1010 DATA   0, 64,144, 64,144,  8
1012 DATA 263,  8,263, 80,319, 80
1014 DATA 319,175,  0,175,  0,168
```

```
1Ø16 DATA 319,168,  8,168,  8,128
1Ø18 DATA  39,128, 39,168, 48,168
1Ø2Ø DATA  48,128, 79,128, 79,168
1Ø22 DATA  88,168, 88,128,119,128
1Ø24 DATA 119,168,152,168,152,12Ø
1Ø26 DATA 191,12Ø,191,168,216,168
1Ø28 DATA 216,12Ø,255,12Ø,255,168
1Ø3Ø DATA 264,168,264,128,279,128
1Ø32 DATA 279,168,296,168,296,128
1Ø34 DATA 311,128,311,168,  8,111
1Ø36 DATA   8, 8Ø, 31, 8Ø, 31,111
1Ø38 DATA  48,111, 48, 8Ø, 71, 8Ø
1Ø4Ø DATA  71,111, 88,111, 88, 8Ø
1Ø42 DATA 111, 8Ø,111,111,16Ø,1Ø3
1Ø44 DATA 16Ø, 72,191, 72,191,1Ø3
1Ø46 DATA 216,1Ø3,216, 72,247, 72
1Ø48 DATA 247,1Ø3,16Ø, 63,16Ø, 32
1Ø5Ø DATA 191, 32,191, 63,216, 63
1Ø52 DATA 216, 32,247, 32,247, 63
1Ø54 DATA 136, 63,136, 16
```

To make sure that the paint points are placed in the Point array, they must directly follow the endpoints. Type them as:

```
1Ø56 DATA 16Ø, 198, 16Ø, 172,  24, 152
1Ø58 DATA  6Ø, 152, 1ØØ, 152, 172, 152
1Ø6Ø DATA 235, 152, 271, 152, 3Ø4, 152
1Ø62 DATA  2Ø,  96,  6Ø,  96, 1ØØ,  96
1Ø64 DATA 175,  88, 232,  88, 175,  44
1Ø66 DATA 232,  44, 139,  4Ø, 2Ø2,  24
```

Finally, add the line data:

```
1Ø7Ø DATA  Ø, 1, 1, 2, 2, 3, 3, 4
1Ø72 DATA  4, 5, 5, 6, 6, 7, 7, Ø
1Ø74 DATA  8, 9,1Ø,11,11,12,12,13
1Ø76 DATA 14,15,15,16,16,17,18,19
1Ø78 DATA 19,2Ø,2Ø,21,22,23,23,24
1Ø8Ø DATA 24,25,26,27,27,28,28,29
1Ø82 DATA 3Ø,31,31,32,32,33,34,35
1Ø84 DATA 35,36,36,37,38,39,39,4Ø
1Ø86 DATA 4Ø,41,41,38,42,43,43,44
1Ø88 DATA 44,45,45,42,46,47,47,48
1Ø9Ø DATA 48,49,49,46,5Ø,51,51,52
1Ø92 DATA 52,53,53,5Ø,54,55,55,56
1Ø94 DATA 56,57,57,54,58,59,59,6Ø
1Ø96 DATA 6Ø,61,61,58,62,63,63,64
1Ø98 DATA 64,65,65,62,66,67,67, 2
```

That was a lot of typing, so review it carefully. Start by glancing over the line numbers. There should be one for every even number after 1004. Check to make sure that the data items are all lined up correctly. If one program line is longer or shorter than the others, it probably contains a typing error.

Type the main routine lines below that retrieve and draw the building:

```
2ØØØ GOSUB 1Ø: C=14: GOSUB 3Ø
21ØØ REM::DRAW BUILDING
211Ø SE$ = "BUILDING": GOSUB 8ØØ
212Ø C=1: GOSUB 9Ø
6ØØØ GET A$
6Ø1Ø IFA$=" ←" THEN GOSUB 2Ø:END
6Ø2Ø GOTO 6ØØØ
```

The program lines that set the paint point variable (PP) and call Tool 60 are shown below. If you glance back at our sketch of this design, you will see exactly where each of these paint points falls within the building. Go ahead and add them to your program, and then RUN it.

```
22ØØ REM::PAINT BUILDING
221Ø PP=68: C=11: GOSUB 6Ø
222Ø PP=69: C=12: GOSUB 6Ø
223Ø PP=7Ø: C=6: GOSUB 6Ø
224Ø PP=71: GOSUB 6Ø
225Ø PP=72: C=5: GOSUB 6Ø
226Ø PP=73: C=9: GOSUB 6Ø
227Ø PP=74: GOSUB 6Ø
228Ø PP=75: C=6: GOSUB 6Ø
229Ø PP=76: C=Ø: GOSUB 6Ø
23ØØ PP=77: GOSUB 6Ø
231Ø PP=78: C=4: GOSUB 6Ø
232Ø PP=79: GOSUB 6Ø
233Ø PP=8Ø: C=Ø: GOSUB 6Ø
234Ø PP=81: C=9: GOSUB 6Ø
235Ø PP=82: C=6: GOSUB 6Ø
236Ø PP=83: C=Ø: GOSUB 6Ø
237Ø PP=84: C=12: GOSUB 6Ø
238Ø PP=85: C=15: GOSUB 6Ø
```

Wait a few minutes for the entire design to be plotted and painted. When it is complete, take a few moments to study it.

> *Note:* If you are having any problems with this design, refer back to page XX. Although the line numbers have changed since then, the basic problems/solutions have not.

Several different colors have been used in this composition. The colors appear to be randomly selected because, to some extent, they were. Starting with a mixture of colors allows you to test the visual effect each lends to the design. One obvious effect is that the darker windows appear to be in rooms having a smaller light source. The brighter, pastel colors, on the other hand, almost give the rooms a glowing look.

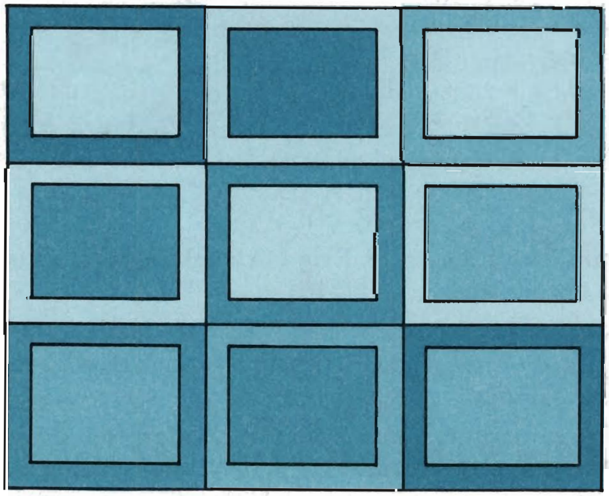
Observe also how the windows are of different sizes and placements. In the middle row of windows, the two on the right have been placed slightly higher than those on the left. Don't be surprised if you did not notice this until now. The human mind strives to maintain a degree of balance and order in all that it takes in visually. For this reason, it will often project balance into a design. Windows that are placed or sized slightly differently are often perceived as being identical. Look at the lower roofs at either side of the building. The lower roof on the right side is, at first, perceived as being the same height as the lower roof on the left side. It is only upon close inspection that this illusion becomes clear.

In the same way, the mind searches for balance in color. You can achieve color balance in several ways. One type of balance was displayed with the butterfly. In that example, balance and neutrality were achieved through contrasting colors.

You reach another type of balance by distributing colors proportionally. If one color does not quite seem to fit your design, don't discard it too quickly. It may be that there is too little or too much of it. Try adding some more of it to the design. As backwards as this might sound, it has been known to work. In fact, this "proportional" color balancing can even be carried into your living room. Has that orange loveseat always bothered you? Try adding orange somewhere else in the room. You might be pleasantly surprised at the results.

Finally, notice how the vanishing line adds perspective to the picture. Perspective is the three-dimensional representation of images as they would normally appear to the eye. By simulating the angle formed by the front and left sides of the building, a truer image is formed.

The sketch below offers another good design for testing and comparing different colors together.



Although the sketch is in black and white, this design places squares of various colors on several different background colors. You can test and compare nine foreground/background combinations at the same time. Some important considerations are:

Do the colors look nice together?
Do the colors run together or create a line where they meet?
Is there too much contrast between the colors?
Is there too little contrast between the colors?
Does one color dominate the other?
Do the colors clash?

Some color combinations produce blurred images, while others create a sharp, distinct contrast among the shapes. Contrast can be achieved through the use of light and dark colors, or warm and cool colors. The warm color family includes yellow, orange and red. The cool colors are blue, green and violet. Using yellow and violet together creates contrast.

TOOL 60 :::::: PAINT A SHAPE

```
6Ø REM::::::PAINT A SHAPE
61 X=P%(PP,Ø): Y=P%(PP,1): IF X<Ø OR Y<Ø OR
   X>319 OR Y>199 THEN RETURN
62 SYS 49748,X,Y,C,MU
63 RETURN
```

What It Does: This tool can paint any completely enclosed (outlined) shape already drawn on the screen.

Example Use: The steps to painting a shape outside its DATA statements are:

(1) Draw the outline shape.

(2) Set C to the desired painting color (C=?).

(3) Give the X,Y location of one paint point within the shape to paint (X=?: Y=?).

(4) Call the paint tool, skipping over lines 60 and 61 (GOSUB 62).

To define a shape's paint inside its DATA statements:

(1) Sketch the shape on a grid and number its Endpoints (#0, #1, #2,...#N).

(2) Locate each paint point on the sketch, and number them as a continuation of the endpoints (#M...#X).

(3) Write down your Point data, from Endpoint #0 thru Paint Point #X.

(4) Write down your Line data.

(5) Enter your Point and Line data into the program as DATA statements. Do this as explained in Chapter 2.

(6) Retrieve and draw the shape on the screen using Tools 800 and 90.

(7) Set C to the paint color (C=?).

(8) Set PP to the paint point within the area you are going to paint (PP=?).

(9) Call the paint tool (GOSUB 60).

Technical Description: This subroutine starts at the specified paint point and invisibly races to the left until it hits either the edge of the shape or the screen boundary. It then plots a straight line to the right until it hits the other side of the shape or the screen boundary. Along the way it looks up and down for any "interesting" points. An "interesting" point is an unplotted point which lies just to the right of a plotted point. The subroutine saves these interesting points for later.

When it has painted all the way to the right edge of the shape, it retrieves the last interesting point it saved. Starting at that point, it then races to the left again, repeating the search for more interesting points. It paints this row, moving right, just as it had the previous row. This process continues until it runs out of interesting points to save and has retrieved all the ones it has saved.

Surprisingly, this is all it takes to fill an object. The secret is keeping track of the interesting points. This process can be very intriguing to watch when painting irregular shapes. The PAINT A SHAPE routine seems to miss some sections of the shape, only to fill in those sections at the last possible instant. (See the machine language listing in the appendices for more information on this tool.)

Summary

Painting is a necessary skill to acquire in any graphics endeavor. Before you begin painting, you should consider any color block problems that might be created by your design. If multi-color would remedy the problem, set MU=1 before the GOSUB 10 statement.

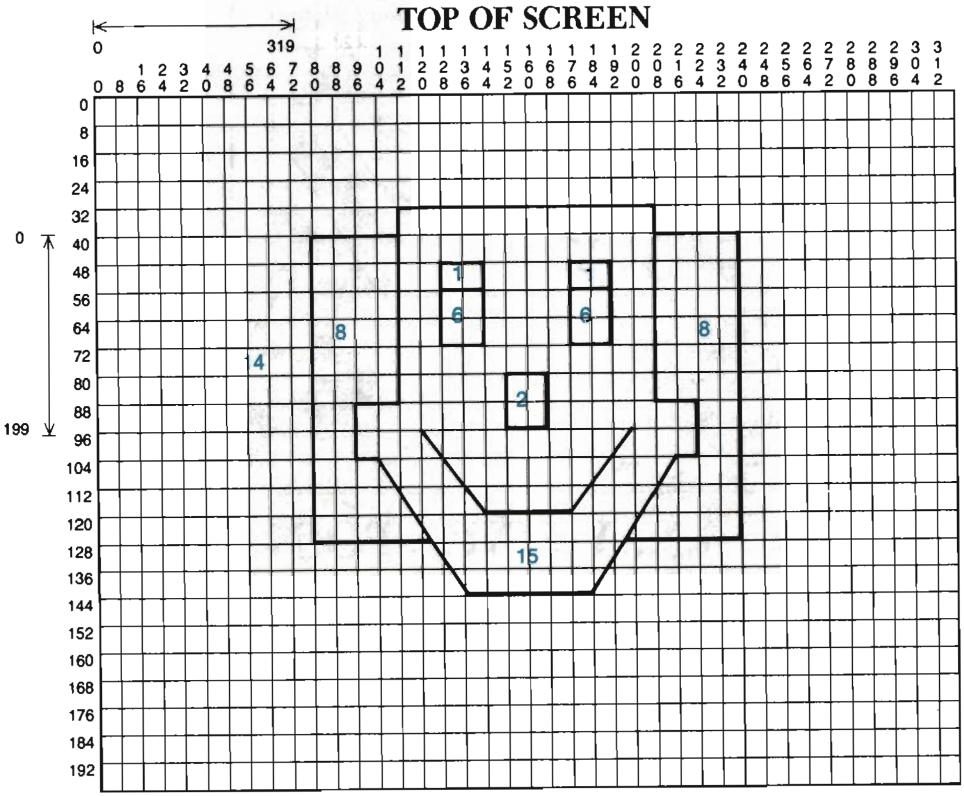
As you paint, consider the proportional balance of your colors. Check for colors that seem to dominate the screen, or colors that seem to get lost in the crowd. Also, consider the balance between the light colors and dark colors. Artists often turn their work upsidedown and sideways to make sure that a proper balance has been established. Instead of jeopardizing your monitor, tilt your head this way and that way to view your pictures from different angles.

Decide ahead of time if you need to pre-define your paint points. You can't do any harm by listing paint points in the DATA statements, and you will probably be glad you did. However, if you decide to paint an area not defined in the data statements, simply set X and Y, and call the paint tool with a GOSUB 62.

Save your program under the filename "CHAPTER 3" before moving on to the exercise.

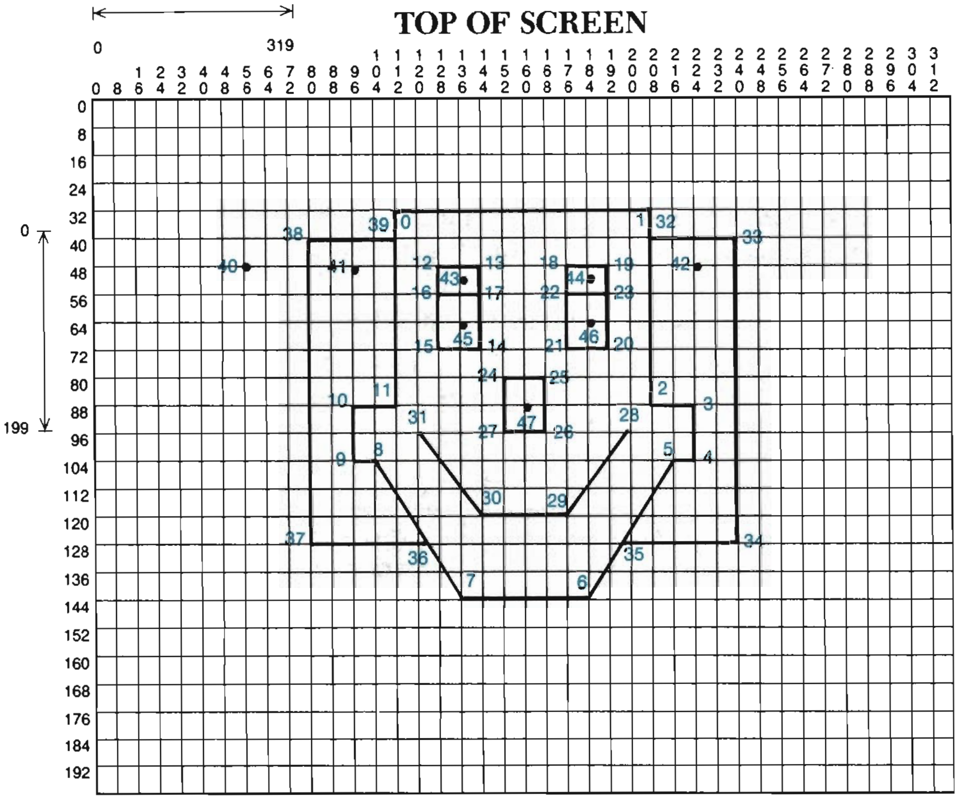
Exercise

This exercise is a little tough. Try it on your own; but if you get too frustrated, skip directly to the solution. The clown design shown below needs to be outlined in black, and then painted with the indicated color codes:



X, Y PIXEL POINTS

Below we have the same sketch, but with the endpoints and paint points numbered for you:



X, Y PIXEL POINTS

There is a color block problem with the mouth that needs to be solved. Also, the coordinates of points 35 and 36 can only be found through trial and error.

This design can be painted in high resolution, but it will take some thinking to figure out how. We are using a difficult example for the exercise to show you that the improbable is not necessarily the impossible. The solution is shown below. Our point coordinates may not match yours exactly, which is fine.

Solution

You were provided one hint to the color block solution—there was no paint point given within the clown's face, while one was given in the background area (at 56,48). This is because the clown's face will use the background color (15), enabling you to use a foreground color to draw the mouth. Thus, the blue "background" area is actually painted like any other shape. Look at the solution below, particularly line 2010, where the background color is set. Line 2040 uses paint point 40 to paint in the light-blue area behind the clown.

```
1140 DATA "CLOWN",47,34
1142 DATA 111, 31,208, 31,208, 87
1144 DATA 224, 87,224,104,216,104
1146 DATA 183,144,136,144,103,104
1148 DATA  95,104, 95, 87,111, 87
1150 DATA 128, 48,143, 48,143, 71
1152 DATA 128, 71,128, 56,143, 56
1154 DATA 176, 48,191, 48,191, 71
1156 DATA 176, 71,176, 56,191, 56
1158 DATA 152, 80,167, 80,167, 95
1160 DATA 152, 95,199, 96,175,119
1162 DATA 144,119,120, 96,208, 40
1164 DATA 239, 40,239,127,197,127
1166 DATA 121,127, 80,127, 80, 40
1168 DATA 111, 40
1170 DATA  56, 48, 96, 48,224, 48
1172 DATA 136, 50,184, 50,136, 64
1174 DATA 184, 64,160, 88
1176 DATA  0, 1, 1, 2, 2, 3, 3, 4
1178 DATA  4, 5, 5, 6, 6, 7, 7, 8
1180 DATA  8, 9, 9,10,10,11,11, 0
1182 DATA 12,13,13,14,14,15,15,12
1184 DATA 18,19,19,20,20,21,21,18
1186 DATA 16,17,22,23,24,25,25,26
1188 DATA 26,27,27,24,28,29,29,30
1190 DATA 30,31,32,33,33,34,34,35
1192 DATA 36,37,37,38,38,39
2000 GOSUB 10: C=15: GOSUB 30
2010 SE$ = "CLOWN": GOSUB 800
2020 C=0: GOSUB 90
2030 PP=40: C=14: GOSUB 60
2040 PP=41: C=8: GOSUB 60
2050 PP=42: GOSUB 60
2060 PP=43: C=1: GOSUB 60
2070 PP=44: GOSUB 60
2080 PP=45: C=6: GOSUB 60
2090 PP=46: GOSUB 60
2100 PP=47: C=2: GOSUB 60
```

Chapter Four

TRANSLATING SHAPES

This chapter introduces the first of three transformation tools. The word *transform* means to change the form or appearance. When you transform a shape, you change its appearance as plotted on the screen. The computer's capacity to store and retrieve information, coupled with its lightning speed, makes shape transformation quick and easy.

The ability to transform shapes is unique to computer art, and is one of the greatest benefits computer art has over the more traditional methods of drawing and painting. The realm of experimentation, once limited by the time required to re-paint an image, has expanded in a way never before imagined. Using oil paints, for example, you could not easily move a figure from the left of the canvas to the right—you must paint over the original figure and then completely re-paint it in the new position. With this chapter's TRANSLATE A SHAPE tool, you can easily move a figure.

Translate, as used here, means to move from one place to another. When you translate a shape, you plot it at a new screen location. The purpose of such a tool is two-fold. 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 particular direction. Up to this point, the only way to move a shape would be to re-define its coordinates within the DATA statements. With your new tool, only *one* X and *one* Y adjustment is required to re-plot an entire shape in a new location.

The second important feature of this tool is its ability to duplicate a shape. Once you've defined a shape with DATA statements, this new tool can duplicate it as often as you like, anywhere on the screen. To duplicate the shape, all you need to do is specify where you want the duplicate(s) to appear.

To set up for this chapter, load CHAPTER 3's program into memory. Next, check that line 172 has C set to 1008, and then run the ZAP routine. Finally, enter the necessary program lines that allow you to enter and exit high resolution graphics:

```
2ØØØ GOSUB 1Ø: C=2: GOSUB 3Ø
6ØØØ GET A$
6Ø1Ø IF A$ = "←" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
```

Test Plotting Your Shapes

The TRANSLATE A SHAPE tool affords you the freedom to plot a shape in several different screen locations before deciding on its final placement. Remember that your shapes are currently described as *single, whole* objects. Because of this, your new tool will only need to know how far horizontally (left or right), and how far vertically (up or down) you want a shape to be moved. With these two pieces of information, the tool can take any shape, of any size, and re-plot it where instructed.

Transformation tools, like TRANSLATE A SHAPE, require several small "supporting" tools to help get their various tasks done. A simply analogy will help to explain this. Suppose you have been hired as the head cake-froster at a neighborhood bakery. The cakes are pre-made, and your only task is to mix up various frostings and apply them to the cakes. Six steps are involved in frosting a single cake:

(1) Wash out your mixing bowl.

(2) Wash out your measuring cup (you only have one).

(3) Determine the first (or next) ingredient needed in the frosing mix and place it in the measuring cup.

(4) Pour the current contents of the measuring cup into the mixing bowl, and stir up the ingredients.

(5) Steps 2 through 4 are completed for each ingredient.

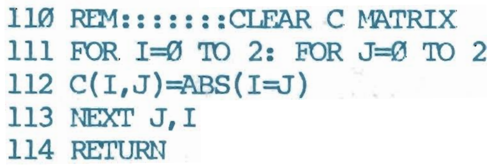
(6) When all ingredients have been combined together and sufficiently stirred, apply the frosting to the cake.

Admittedly, this analogy has its flaws. It does, however, serve its purpose.

For each step above, except step 5, there is a corresponding tool to be added to your tool kit. The cake to be frosted is equivalent to the shape you want to transform. The frosting in the mixing bowl is equivalent to the combined list of transformations (e.g., translate, scale, rotate) to be applied to the shape. Each ingredient represents an individual transformation request (e.g., translate).

The new tools will use matrices to hold and combine your transformation specifications. The word "matrix" is a mathematical term for a two-dimensional array. Two such matrices are required for transformations. One matrix, called the *C Matrix*, will be used as the "mixing bowl," and will hold the current list of transformations you have requested. The other matrix, called the *T Matrix*, will be used as the "measuring cup" to hold the last, single transformation request.

Below is the first tool you need to type. This tool, called CLEAR C MATRIX, will wash out your "mixing bowl" so that a new transformation recipe can be mixed. This tool needs to be called at the beginning of your program, as well as each time you want to apply a new set of transformations to one of your shapes.



```
11Ø REM::::::CLEAR C MATRIX
111 FOR I=Ø TO 2: FOR J=Ø TO 2
112 C(I,J)=ABS(I=J)
113 NEXT J,I
114 RETURN
```

The next tool to enter, shown below, is called CLEAR T MATRIX. This tool cleans out the "measuring cup" to prepare it for a new transformation ingredient.

As you will see in a moment, this tool is called from within the TRANSLATE A SHAPE tool.

```
120 REM::::::CLEAR T MATRIX
121 FOR I=Ø TO 2: FOR J=Ø TO 2
122 T(I,J)=ABS(I=J)
123 NEXT J,I
124 RETURN
```

This third tool is your TRANSLATE A SHAPE tool. Its task is to place your translation request into the measuring cup (T Matrix). Notice that the first thing it does is call Tool 120, to ensure that the measuring cup has first been properly cleaned out.

```
140 REM::::::TRANSLATE A SHAPE
141 GOSUB 12Ø
142 T(2,Ø)=XT: T(2,1)=YT
143 GOTO 13Ø
```

Next, the COMBINE MATRICES tool. This tool takes the transformation ingredient out of the measuring cup (T Matrix), places it into the mixing bowl (C Matrix), and stirs up the ingredients. This tool, as you can see in program line 143, is called after the computer completes Tool 140's task.

```
13Ø REM::::::COMBINE MATRICES
131 FOR I=Ø TO 2: FOR J=Ø TO 2
132 W(I,J) = C(I,J)
133 NEXT J,I
134 FOR S=Ø TO 2: FOR J=Ø TO 2
135 C(S,J)=Ø
136 FOR I=Ø TO 2
137 C(S,J)=C(S,J)+W(S,I)*T(I,J)
138 NEXT I,J,S
139 RETURN
```

The final step, applying the frosting to the cake, is done with the APPLY TRANSFORMS tool below. This tool will take the current list of transformations found in the C matrix and apply it to the Point data list stored in P%.

```
1ØØ REM::::::APPLY TRANSFORMS
1Ø1 FOR S=Ø TO ND: FOR J=Ø TO 2
1Ø2 R%(S,J)=Ø
1Ø3 FOR I=Ø TO 2
1Ø4 R%(S,J)=R%(S,J) + P%(S,I)*C(I,J)
1Ø5 NEXT I,J,S
1Ø6 RETURN
```

An important thing to know about the APPLY tool is that the newly transformed Point list is *not* returned to your P% array. Instead, it is placed in an R% array. The

"R" stands for Replica, and this array will hold a transformed replica (copy) of your shape's Point data list. As the transformations are applied to each point in P%, the newly transformed points are placed in R% for drawing and painting purposes.

This leads us to a couple of final adjustments to your tool box. First, some changes to the DRAW A SHAPE tool you've been using. Line 91 needs to be added so that Tool 100 is called when needed. In addition, lines 94 and 95 need to be corrected, so that a replica of the shape's Point list is used, rather than the Point list. Make the following addition/alterations to Tool 90:

```
91 GOSUB 1ØØ
94 X1=R%(E1%,Ø): Y1=R%(E1%,1)
95 X2=R%(E2%,Ø): Y2=R%(E2%,1)
```

Finally, the PAINT A SHAPE tool needs to be amended. Currently, this tool will retrieve paint points from the P% array. However, now you will be translating the points in your Point data list. The newly transformed points, including all paint points, will be stored in an R% array. Thus, we now want this tool to retrieve all paint points from the R% array. Change line 61 to the following:

```
61 X=R%(PP,Ø): Y=R%(PP,1): IF X<Ø OR Y<Ø OR X>319
   OR Y>199 THEN RETURN
```

That completes this chapter's contribution to your tool kit. The chart below gives a brief description of your new tools, together with the "frosting analogy" as it applies to each. As we begin using these new tools to create designs, you may want to refer to this chart as a reminder of how transformations are accomplished.

TOOL	BRIEF DESCRIPTION	"FROST CAKE" ANALOGY
110 CLEAR C MATRIX	Clears out C matrix, setting it up to hold a *list* of transforms.	Cleans out mixing bowl, getting it ready for next frosting mix.
120 CLEAR T MATRIX	Clears out T matrix, setting it up to hold the *next* transform request.	Cleans out measuring cup, getting it ready for next ingredient.
140 TRANSLATE A SHAPE	Places translate request into the T matrix.	Places next ingredient into measuring cup.
130 COMBINE MATRICES	Mathematically combines T and C matrices, placing results back into C matrix.	Pours measuring cup into mixing bowl and stirs up contents.
100 APPLY TRANSFORMS	Applies the C matrix to the points listed in P%, placing resulting points into R%.	Places the frosting on your cake.

Most of the above information is simply that—information. Out of the five new and two modified tools just entered, you need never think about any except:

(1) Tool 110 (CLEAR C MATRIX).

(2) Tool 140 (TRANSLATE A SHAPE).

(3) Tool 90 (DRAW A SHAPE).

(4) Tool 60 (PAINT A SHAPE).

The rest of your new tools are "support" tools. Support tools are called by the main tools as needed. If you were to glance through your tool box, you would find that:

—Tool 120 (CLEAR T MATRIX) is called by Tool 140
—Tool 130 (COMBINE MATRICES) is called by Tool 140
—Tool 100 (APPLY TRANSFORMS) is called by Tool 90

(Now, forget about tools 100, 120, and 130. There will never be an occasion where you will have to think about them or call them.)

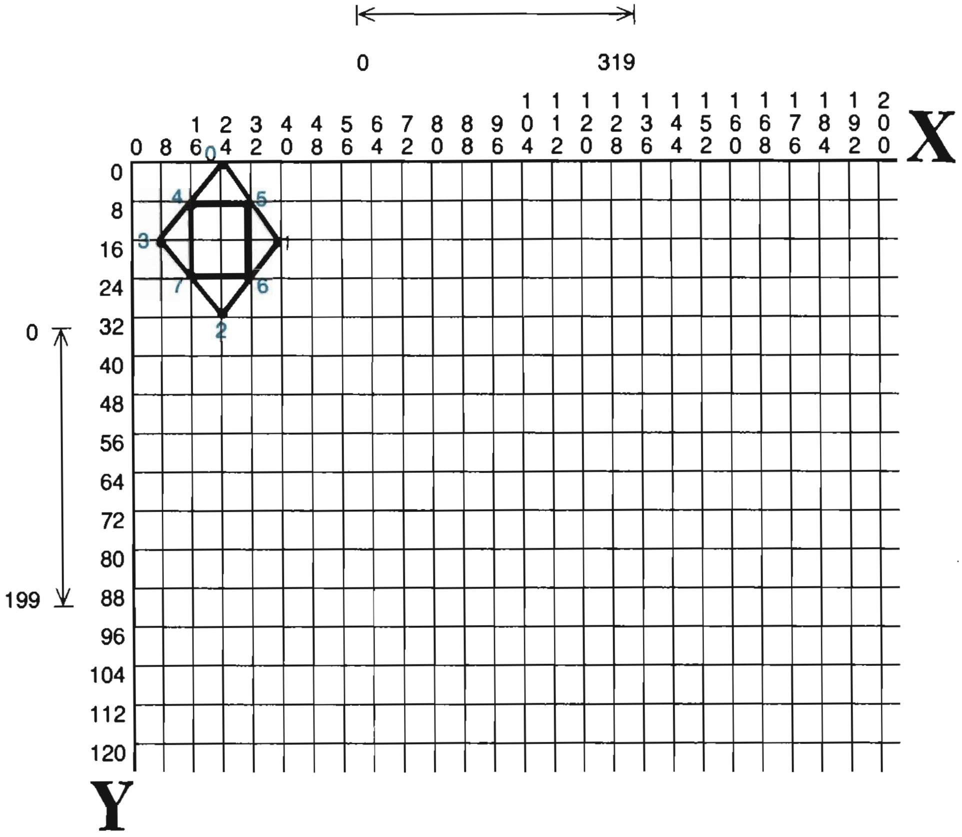
With the tools in place, you are ready to put them to use. The first step is to dimension all of the arrays/matrices. This includes the following:

—the P% array (Point data list; dimensioned as usual)
—the L% array (Line data list; dimensioned as usual)
—the R% array (Replica array; dimensioned same as P%)
—the T matrix (holds the current transformation; always dimensioned for three rows and three columns)
—the C matrix (holds complete list of transformations; always dimensioned for three rows and three columns)
—the W matrix (a temporary storage place for the contents of the C matrix while some matrix multiplication is done; always dimensioned for three rows and three columns)

Each of these arrays must be dimensioned once, but no more than once. Because of this restriction, it is not possible to dimension an array inside a subroutine. If, for example, the C matrix were dimensioned inside Tool 110, the program would try to dimension the C matrix each time Tool 110 was called. This would result in the program failing. Change line 1005 so that it will once and for all dimension the six required arrays:

```
1ØØ5 DIM P%(99,2), L%(99,1), R%(99,2), T(2,2), C(2,2), W(2,2)
```

The design you will work with is a simple quilting pattern, shown on the grid below. If you have an interest in quilting, this new tool works wonders with geometric patterns, and will prove a terrific design aide when planning your next quilt. (This is also true for tiling, stained glass, needlepoint, or any other design-oriented craft where repetitive patterns are often used.)



Although all of the endpoints have been appropriately numbered in the quilt design, no paint points have been identified. This is fine. You only need to identify paint points in shapes you plan to paint. We will not be painting this particular design.

The DATA statements that define this quilt design are given below. Type them into your Shape Library, then double-check your typing for accuracy:

```
1ØØ8 DATA "QUILT DESIGN", 7,7
1Ø1Ø DATA 24, Ø,39,16,24,31, 8,16
1Ø12 DATA 16, 8,31, 8,31,23,16,23
1Ø14 DATA  Ø, 1, 1, 2, 2, 3, 3, Ø
1Ø16 DATA  4, 5, 5, 6, 6, 7, 7, 4
```

Finally, add the following main routine lines that will plot this shape on the screen:

```
2Ø1Ø SE$="QUILT DESIGN": GOSUB 8ØØ
2Ø2Ø GOSUB 11Ø
2Ø3Ø C=1: GOSUB 9Ø
```

Notice that line 2020 calls the CLEAR C MATRIX tool. This matrix must be properly set at the beginning of each picture-drawing program.

Take the time to RUN the program. If the quilt design is plotted in white in the upper left-hand corner of the screen, press ← to return to text mode.

If you are having any problems, check the new subroutines and then the DATA statements. Correct any mistakes before continuing.

Now that you have seen where this shape is normally plotted, let's learn how to move it to other screen locations. First, specify *where* the shape is to be moved. This is done with "offset" values. Offset values specify how far a shape should be moved along the X axis (XT=?), and how far it should be moved along the Y axis (YT=?). Assigning XT (X-Translate) a value other than 0 will move the shape right (+XT) or left (-XT). If you set XT=5 and then call Tool 140 (TRANSLATE A SHAPE), the C matrix will be set to move a shape 5 pixels to the right. If you set XT equal to -5 and then call Tool 140, the C matrix will be set to move a shape 5 pixels to the left.

By the same token, assigning a value to the variable YT (Y-Translate) will move your shape up (-YT) or down (+YT) the screen. For example, setting YT equal to -56 and then calling Tool 140 moves your shape *up* 56 pixel rows. Setting YT equal to positive (+) 56 moves it *down* 56 pixel rows.

Try out this idea by modifying your program as follows:

```
    2030 XT=100: YT=50: GOSUB 140
    2040 C=1: GOSUB 90
```

When you RUN the program, you will find that the shape has been moved 100 columns to the right, and 50 rows down. In essence, each X coordinate in your shape's DATA statements had 100 added to it (the value assigned to XT), and each Y coordinate had the value 50 added to it (the value assigned to YT).

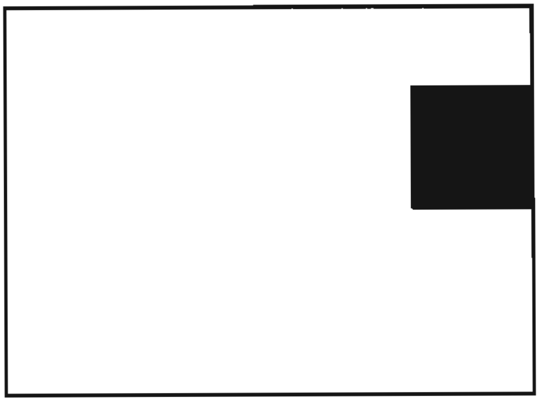
Change line 2030 to try another location:

```
    2030 XT=296: YT=16: GOSUB 140
```

RUN the program. As you probably guessed, the shape is plotted partially off the screen. Because you have a clipping tool, there is no danger of the "unseen" points ever really being plotted.

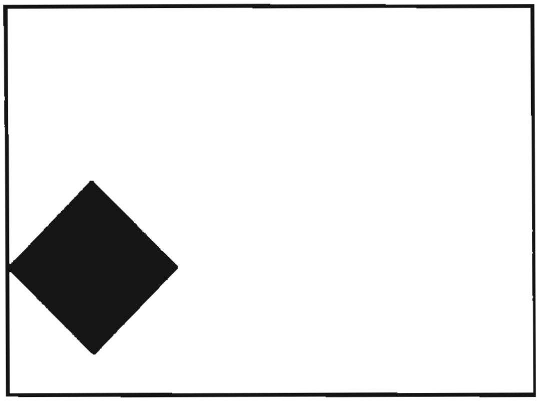
A shape's placement is an important consideration in your artwork. As you maneuver a shape around the screen, you will find that different visual effects occur. This can be shown with the black square sketched below.

The square is centered in the drawing area. Sketched in a predictable location, it would probably arouse little interest from passers-by. The centered square is a calm, unquestioned beginning to a composition. However, if you place that same square against the border, you raise the question of why it was placed there:



The square now seems to "peek" around the edge of the drawing area. Because it has been placed higher on the drawing board, it appears further back in space than before.

As demonstrated in the next sketch, placing a shape flush against the border creates tension. This tension is the result of your mind's interpretation of how the shape and border interact. In this particular design, the diamond can be interpreted as "resting" against the border. You have the impression that the shape is being pulled to the left by gravity, only to be stopped by the border.



Take some time, if you like, to try your own offset values. You might also want to take this opportunity to experiment with the PAINT A SHAPE tool. To do so, add paint points to your DATA statements and adjust the count of points in line 1008. (Don't delete this shape from your library or load another program. The quilt design is needed in the next section.)

Duplicating Shapes

It is fairly easy to understand how to duplicate shapes across and around the screen area. To copy a shape, all you need to do is specify where the copy should go (using XT and YT), and then draw the shape (GOSUB 90). Each time you offset a shape and draw it, a new copy will appear on the screen. As we discuss this use of the TRANSLATE A SHAPE tool, you will need to keep in mind one important rule about offsetting shapes:

> Until you clear the C Matrix (GOSUB 110) of any transformations it is storing, the XT and YT values will offset a shape relative to the *last* place it was located.

This will become clear in the next exercise. Currently, your program is set to do the following:

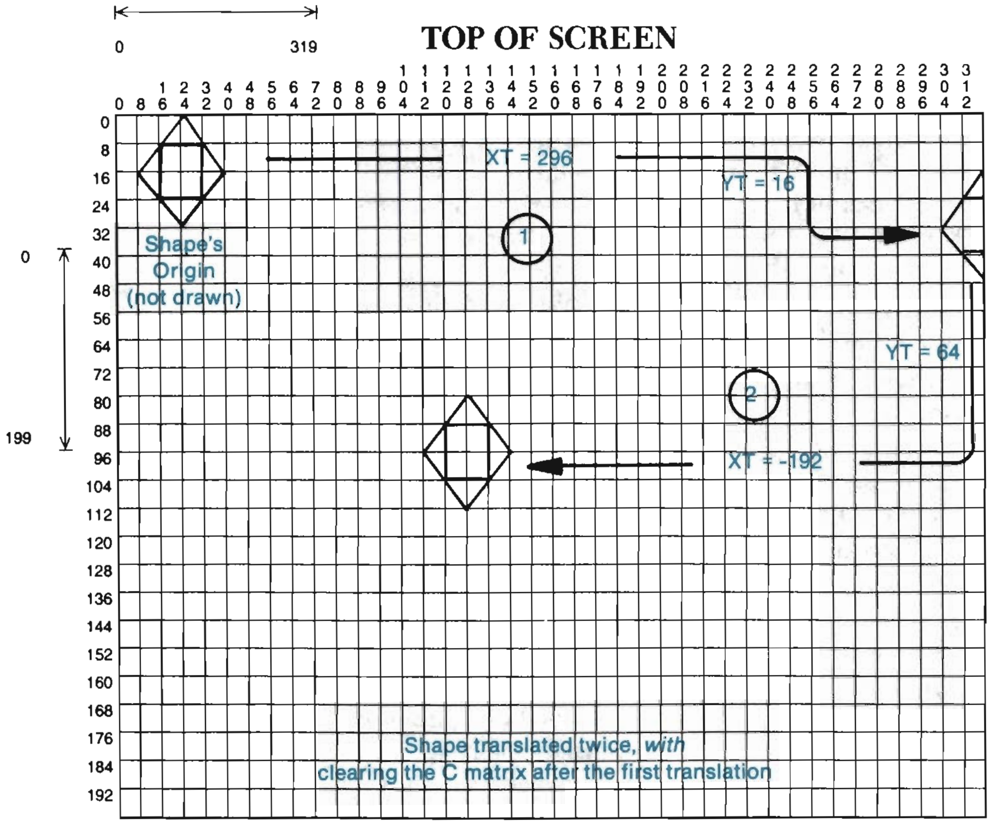
—Retrieve the quilt design (line 2010)
—Clear the C Matrix for a new set of transformations (line 2020)
—Offset the *last* location of the shape by 296 columns and 16 rows (line 2030)
—Draw the shape (line 2040)

Before the computer reads program line 2030, the shape is located at its origin (i.e., the original drawing location as described in the DATA statements). Once it reads line 2030, the computer translates the shape according to the XT and YT offset values. If the shape were translated again, but without clearing the C matrix first, it would be translated *away* from the location to which it was last translated. Until the C matrix is cleared, translating a shape will move it away from the last location to which it was moved. If you clear out the C matrix, the shape will again be translated away from its origin.

Look at the program lines below. These lines move your shape again, but without clearing the C matrix first. Notice that the shape will be drawn (GOSUB 90) at its new location. This will result in plotting two copies of the quilt design on the screen. Add these lines to your program and RUN it.

```
2050 XT=-192: YT=64:GOSUB 140
2060 GOSUB 90
```

The diagram below shows how the offset values were used to place each copy of the shape on your screen. In step one on the diagram, the shape is translated relative to the origin (last place it was located). It is translated according to the values of XT and YT. In step two, the new offset values are added to the old ones because the C matrix was never cleared out. This offsets the shape from the last place it was located, as opposed to its actual origin.



X, Y PIXEL POINTS

Obviously, the XT and YT values are cumulative. This means that each time you assign a value to XT and call Tool 140, that value is added on to any past value XT had. The same idea is true for YT. These variables will continue to accumulate until you clear out the C matrix. In our current example, XT is first set to 296 and YT is set to 16. Once that transformation is complete, XT and YT are re-set in preparation for the next transformation. The cumulation of these variables produces the following values for the second transformation:

XT equals 296 (old value) -192 (new value) = 112
YT equals 16 (old value) + 64 (new value) = 80

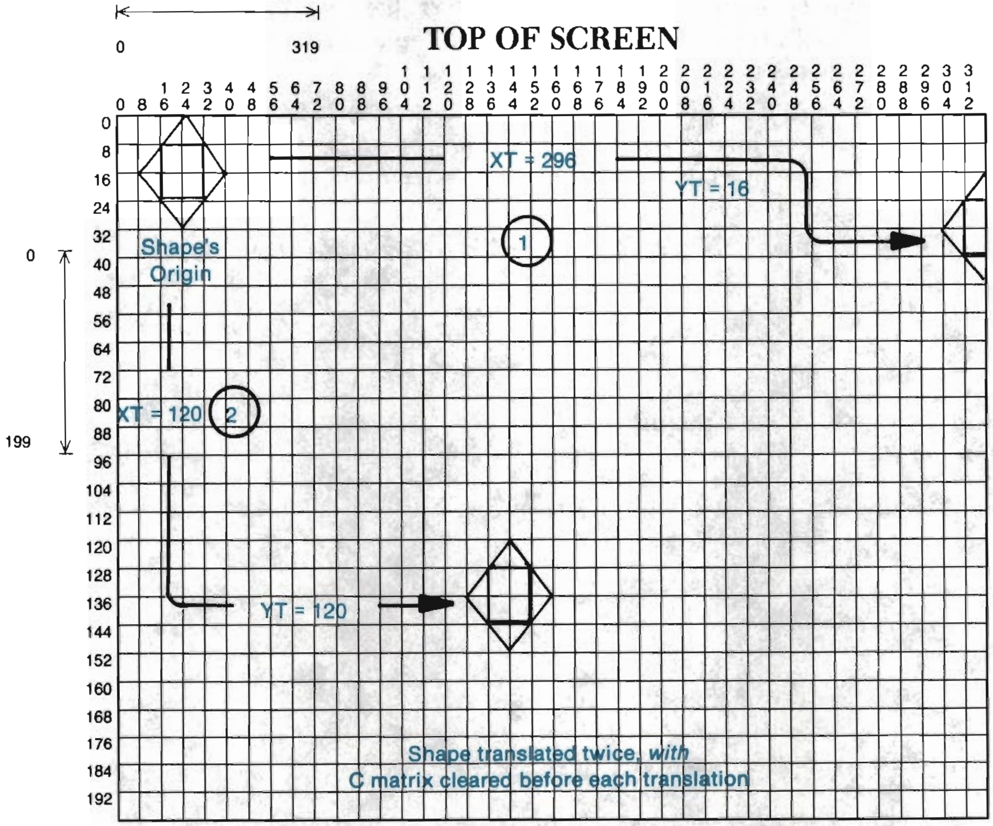
By making XT = New XT + Old XT, and YT = New YT + Old YT, shapes are moved away from their last location, and not necessarily their origin.

Change program lines 2050 and 2060 as shown below, adding program line 2070:

```
2Ø5Ø GOSUB 11Ø
2Ø6Ø XT=12Ø: YT=12Ø: GOSUB 14Ø
2Ø7Ø GOSUB 9Ø
```

RUN the program. In this exercise, both copies of the shape will be offset relative to the *origin* because the C matrix was cleared before each new translation. Press ← to see what we mean.

Look at the diagram below. In it, you are shown how clearing the C matrix changes the "starting location" from which each copy of the quilt design is translated. Each time the C matrix is cleared with a GOSUB 110, the XT and YT values are also cleared from the C matrix.



X, Y PIXEL POINTS

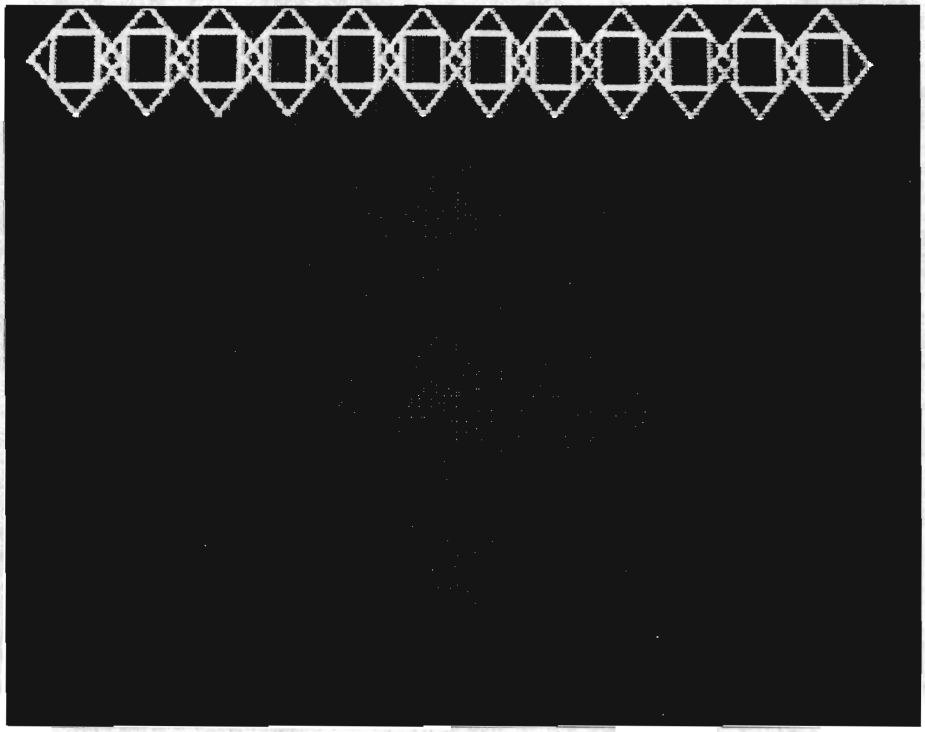
Delete lines 2010 through 2070 from your progam (do this by hand, and not with the ZAP routine, to retain lines 6000 through 6020). You need to get a better feel for what this new tool can accomplish. This can be shown with the following program lines:

```
2100 REM::::::RETRIEVE QUILT DESIGN
2110 SE$="QUILT DESIGN": GOSUB 800
2200 REM::::::CLEAR C MAT & SET COLOR
2210 GOSUB 110: C=1
2300 REM::::::TRANSLATION LOOP
2310 GOSUB 90
2320 XT=24
```

```
2330 FOR M=1 TO 11
2340 GOSUB 140
2350 GOSUB 90
2360 NEXT M
```

This program is set up to draw the shape at line 2310, and then begin a loop that will execute eleven times. Each time the loop is executed, the shape is translated along the X axis 24 places (XT is set at 24 and never changed). Each time the shape is translated, it is also drawn. The result will be 12 copies of the shape plotted across the screen. RUN the program.

The quilt design is 32 pixels wide. By translating it only 24 pixels to the right each time, you create an overlapping effect. This produces new, smaller shapes at the areas where the overlapping occurs. Your screen will display the following when the program is finished:



When the twelve copies are complete, stop the program and list lines 2100-2360. The main point of interest is that GOSUB 110 only appears once. This keeps XT set at 24, thus compounding the offset location of your shape by twenty-four places each time the loop is processed. If line 2340 contained both GOSUB 140 and GOSUB 110, the shape would be drawn eleven times in the exact same offset location:

If you find your are having problems drawing or translating a shape, check the following:

—Check your DATA statements for accuracy.

—Check to make sure GOSUB 110 is entered where necessary (this is an easy one to forget—a GOSUB 110 *must* appear at the beginning of the program).

—Make sure a DIM statement is dimensioning all 6 arrays (P%, L%, R%, T, C and W) as required.

—Make sure the color variable (C) has been changed to a drawing color after the background is painted.

—Look for zeroes typed as oh's, and ones typed as el's.

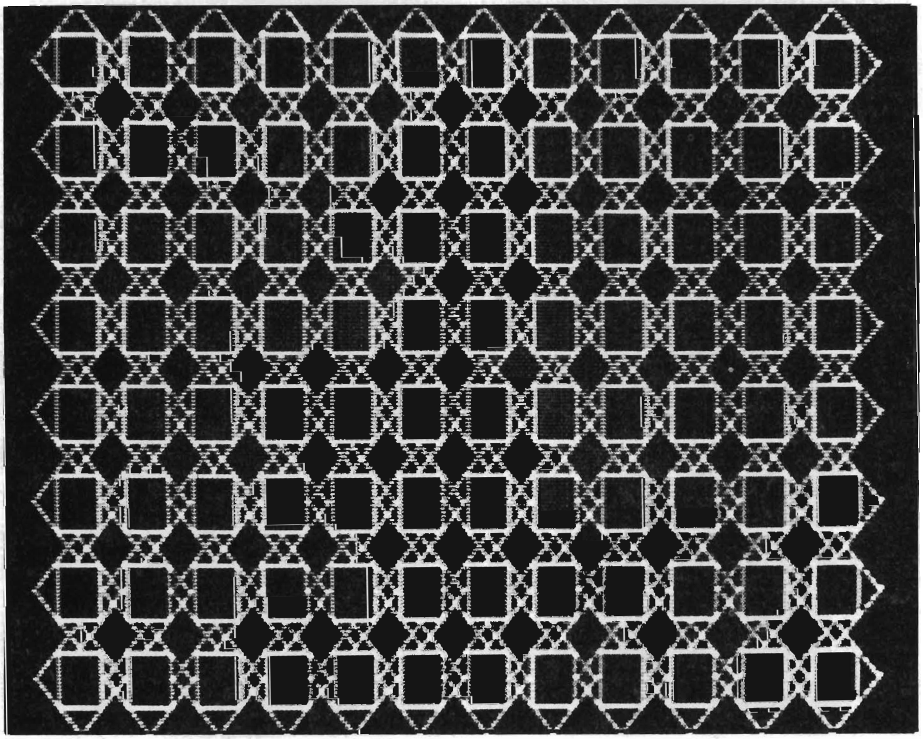
—Check that all necessary variables have been set before each tool is called (SE$, C, XT, YT, etc.)

If you ever find that only *one* point in your shape is plotted, you probably forgot to clear the C matrix. Remember: even if you don't transform a shape, you must clear the C matrix out at least once before the new DRAW A SHAPE tool can work properly.

Just for fun, let's add another loop that copies the design *down* the screen as well as across. Add the following program lines and then RUN the program again:

```
2305 FOR L=1 TO 8
2370 XT=-264: YT=24: GOSUB 140
2380 YT=0: NEXT L
```

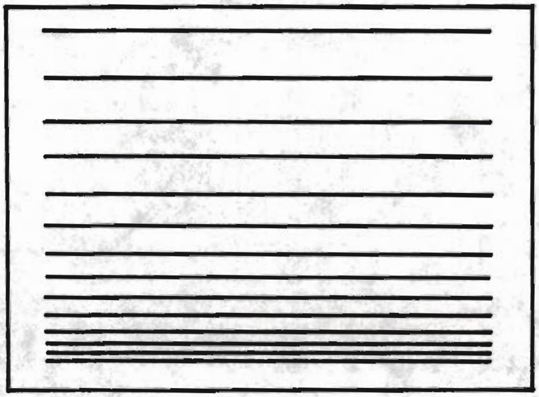
After a few minutes you should see a finished quilt. The overlapping that occurs between each column and each row produces smaller, unplanned shapes. This makes for a much more intricate and interesting pattern.



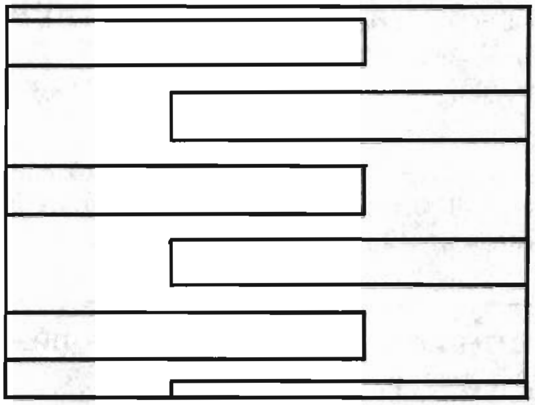
Stop the program so that we can discuss the addition of the "L" loop. This loop complicates the program, but it can be understood with a little thought. Lines 2305 and 2380 create the L loop, which is set to process eight times. Within this loop, the M loop—which draws a row of quilt patterns—is present. Once the first row has been drawn, line 2370 translates the shape back along the X axis 264 places to the left (-XT). This moves the shape back to the beginning of a row. Because YT is set to 24, the shape is moved down the screen twenty-four places (starting another row). Finally, line 2380 returns YT to 0 and calls for the next L. YT has to be returned to 0 to keep the shape from moving down the screen each time line 2340 is executed. (This might better be understood if you removed "YT=0" from line 2380 and ran the program again.)

Type: GOSUB 10 and press **RETURN.** The quilt again should be displayed on the screen. What you are viewing is an example of "repetition." Repetition is the translation of a shape at regular intervals. On your screen, each shape is spaced at an equal distance from the other shapes surrounding it. Using the same shape throughout the picture produces harmony and order in the arrangement. The geometric character of the shapes also reinforces this harmonic relationship.

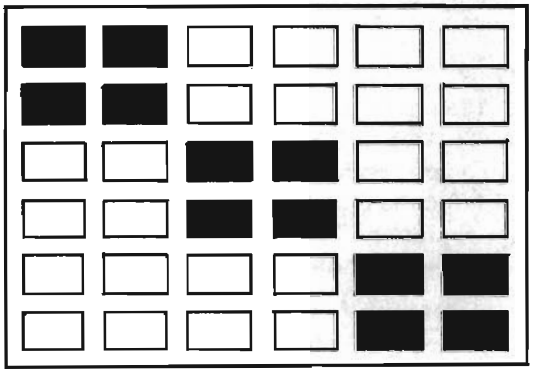
Variations can occur in a repetitive pattern by making slight changes in the surrounding space. In the design sketched below, the space between the shapes progressively increases toward the top. Although the spacing between the shapes varies, it is a planned, regular, and predictable arrangement.



The shapes in the next sketch are arranged in a logical, sequential order. The composition is symmetrically balanced, as both sides of the picture are similar.



Another example of translation is shown in the use of squares below. This arrangement is also sequential and orderly, and the spacing regular and predictable. This type of organization, in which both sides of the picture are equal, is called "formal balance." There's a visual cohesion and unity in this kind of composition. Variety has been provided by the change in light and dark color values.



You can use these designs as a starting base for many, many other designs to come. For more design ideas, try your local library and neighborhood book stores. Some of the best designs have already been figured out for you. All you need to do is look for them. Pay particular attention to designs using repetition—the TRANSLATE A SHAPE tool is ideal for them.

If you'd like to spend some time practicing, follow these steps:

(1) At the beginning of the program, clear the C matrix (GOSUB 110).

(2) Retrieve the shape to translate (SE$="?": GOSUB 800).

(3) If necessary, set the color variable to a drawing color (C=?).

(4) If the shape is to be offset from its origin, insert a GOSUB 110 statement to

clear the C matrix. (This is not necessary if no transformations have taken place since step one above).

(5) Set the variable XT to the distance right (XT=+?) or left (XT=-?) you want the shape moved. Set the variable YT to the distance up (YT=-?) or down (YT=+?) you want the shape moved.

(6) Call the TRANSLATE A SHAPE tool (GOSUB 140).

(7) *Draw the newly located shape* (GOSUB 90).

This chapter's tool boxes follow. The first three tool boxes discuss those tools that *you* have to call (Tools 90, 110, and 140) from within the main routine. The tool boxes for your supporting tools then follow.

TOOL 90::::::DRAW A SHAPE—MODIFIED

```
9Ø REM :::::: DRAW A SHAPE
91 GOSUB 1ØØ
92 FOR J = Ø TO NL
93 E1%=L%(J,Ø): E2%=L%(J,1)
94 X1=R%(E1%,Ø): Y1=R%(E1%,1)
95 X2=R%(E2%,Ø): Y2=R%(E2%,1)
96 GOSUB 7Ø
97 NEXT J
98 RETURN
```

What It Does: This tool begins by calling on Tool 100 to have any transformation requests applied to the shape to be drawn. The transformed shape is placed in R% and L%, and drawn from there. Because Tool 100 (which applies the C matrix to your shape) is called on, you must be certain that the C matrix is set to transform (or not transform) your shape appropriately. The shape to be drawn will be plotted in the color specified by the variable C's most current value.

Example Use: To draw a shape, three basic steps are necessary:

(1) The shape must be defined in DATA statements, and retrieved with Tool 800. To retrieve a shape, all arrays (P%, L%, R%, T, C and W) need to be properly dimensioned.

(2) The variable C must be set to a color code between 0 and 15 (see the color chart listed in the appendices). Be sure the color to draw is *different* from the screen's background color.

(3) A GOSUB 90 statement is necessary to call this tool.

Program lines similar to the following will draw the shape stored in R% and L%:

```
1ØØ6 DIM P%(99,2), L%(99,1), R%(99,2),
     T(2,2), C(2,2), W(2,2)
213Ø C=1: GOSUB 9Ø
```

Technical Description: This modified tool works almostly exactly the same as the original Tool 90 worked. One change, however, is the added GOSUB 100 in line 91. This GOSUB applies the C matrix to the P% array, with the resulting points being placed in the R% array. Also, a change had to be made in lines 94 and 95 so that the shape's points were retrieved from the R% array for drawing purposes.

Earlier in the book we told you that the P% (and now R%) array had to be dimensioned with three columns for "technical reasons." You are now equipped with enough information to learn why this is so. The P% array is multiplied with the C matrix by Tool 130 (COMBINE MATRICES). This C matrix has three rows. To multiply two matrices together, one of the matrices must have the same number of columns as the other matrix has rows. Thus, the P% array needs three columns. The R% array must also have three columns, since it will be storing the results of the transformation.

TOOL 110::::::CLEAR C MATRIX

```
11Ø REM::::::CLEAR C MATRIX
111 FOR I=Ø TO 2: FOR J=Ø TO 2
112 C(I,J)=ABS(I=J)
113 NEXT J,I
114 RETURN
```

What It Does: This tool clears the C matrix, setting it up for a list of transformation requests. This matrix must be set up for transformations at the very beginning of your program. If it is not set properly, your shapes cannot be drawn with the new DRAW A SHAPE tool. In addition, the C matrix must be cleared each time you want to begin transforming a new shape, or each time you want to transform the original shape of one that has already been transformed.

Example Use: This tool should be called from within the main routine by a GOSUB 110 statement. Again, it must be called each time the C matrix needs to be set back to "empty." Because this tool uses a matrix, the C matrix must be dimensioned somewhere within the main routine. This can be done at the beginning of your Shape Library, at line 1005. This DIM statement should be in the form of: DIM C(2,2).

Technical Description: When the C matrix is "cleared," its values are set to identify. The matrix form of the C array then looks like this:

C Matrix

	0	1	2
0	1	0	0
1	0	1	0
2	0	0	1

Matrix identity is defined as the state of the matrix before transformations have taken place. Matrix identity is always achieved by 1's in a diagonal pattern running from the upper left-hand corner to the lower right-hand corner of the matrix, and setting all other cells to zero.

The C and T matrices are mathematically combined each time a transformation is requested. Any time you multiply a matrix (for example, the T matrix) by a matrix set to identify (for example, the C matrix), you will arrive at the same matrix you started with (i.e., the T matrix).

TOOL 140::::::TRANSLATE A SHAPE

```
140 REM::::::TRANSLATE A SHAPE
141 GOSUB 12Ø
142 T(2,Ø)=XT: T(2,1)=YT
143 GOTO 13Ø
```

What It Does: This tool takes your transformation specifications and places them in the T matrix for use later by Tool 130 (COMBINE MATRICES). Transformations can not be placed directly into the C matrix, but must be mathematically placed there, one at a time, by using a temporary "storage" matrix. In this book, the T matrix is used as a temporary storage area.

Example Use: To translate a shape, you must have this subroutine tool and four supporting tools (Tools 100, 110, 120, and 130), as well as a modified DRAW A SHAPE tool (see Tool 90's tool box). All matrices and arrays required by the supporting tools must be dimensioned appropriately (arrays/matrices P%, L%, R%, T, C, and W).

Two offset variables (XT and YT) must be set to specify where the shape is to be moved. The value assigned to XT will determine how far left (-XT) or right (+XT) the shape is to be moved. The value assigned to YT will determine how far up (-YT) or down (+YT) the shape is to be moved. A GOSUB 140 statement will transform the shape. You will also need a GOSUB 90 statement to draw the shape on your screen. This can all be done in the form:

```
211Ø GOSUB 11Ø
212Ø XT=5Ø: YT=25: GOSUB 14Ø
213Ø C=1: GOSUB 9Ø
```

Tool 110 must be called before transforming and drawing a shape.

The XT and YT variables will offset a shape from the *last* location it was translated to. If a shape has never been translated, the values will offset the shape from its origin (i.e., the original location as described in the DATA statements). To clear out the C Matrix, so that a shape is again offset from its origin, a GOSUB 110 statement is necessary before calling the translate tool.

You should *always* clear out the C matrix before translating a new shape. The stored list of transformations in the C matrix will continue to apply to each shape you transform, until you specifically clear out this matrix with a GOSUB 110 statement.

*Technical Description:*To store an XT and YT translation, it is necessary to clear out the T matrix. XT and YT can then be placed in their proper positions in the T matrix, so that they can later be applied to the C matrix. When the XT and YT translations have been placed in the T matrix, the T matrix looks like this:

T Matrix

	0	1	2
0	1	0	0
1	0	1	0
2	XT	YT	1

The following tool boxes deal with "supporting" tools. Tools of this type are called automatically by "main" tools—they need never, and should never, be called by you.

TOOL 100:::::::APPLY TRANSFORMS

```
100 REM:::::::APPLY TRANSFORMS
101 FOR S=Ø TO ND: FOR J=Ø TO 2
102 R%(S,J)=Ø
103 FOR I=Ø TO 2
104 R%(S,J)=R%(S,J) + P%(S,I)*C(I,J)
105 NEXT I,J,S
106 RETURN
```

What It Does: This tool will apply all transformations stored in the C matrix to the list of points stored in P%. The resulting list of points is then placed in R% for use by the DRAW A SHAPE tool.

Example Use: This tool is automatically called by Tool 90 (DRAW A SHAPE).

Technical Description: The P% array can be thought of as a matrix with many rows. Since it is a matrix, it can be multiplied by the C matrix, with the results being placed in R%. (See COMBINE MATRICES tool box for information on matrix multiplication.) That is what this tool does. The shape stored in R% is the shape that is then drawn.

TOOL 120::::::CLEAR T MATRIX

```
120 REM::::::CLEAR T MATRIX
121 FOR I=Ø TO 2: FOR J=Ø TO 2
122 T(I,J)=ABS(I=J)
123 NEXT J,I
124 RETURN
```

What it Does: This tool clears the T matrix, which holds one transformation request at a time. This matrix must be cleared out before a new transportation request can be stored in it.

Example Use: This tool is automatically called by the TRANSLATE A SHAPE tool. You will find in later chapters that the other transformation tools also call this CLEAR T MATRIX tool automatically. Your main routin, however, *must* dimension the T matrix once (and only once) for three rows and three columns. To dimension the T matrix, the main routine should contain a program line set up in the form of:DIM T(2,2).

Technical Description: The T matrix is cleared in the same manner and for the same reason as the C matrix. It is set to "identity." In this state, the T matrix looks like this:

T Matrix

	0	1	2
0	1	0	0
1	0	1	0
2	0	0	1

TOOL 130::::::COMBINE MATRICES

```
130 REM::::::COMBINE MATRICES
131 FOR I=Ø TO 2: FOR J=Ø TO 2
132 W(I,J) = C(I,J)
133 NEXT J,I
134 FOR S=Ø TO 2: FOR J=Ø TO 2
135 C(S,J)=Ø
136 FOR I=Ø TO 2
137 C(S,J)=C(S,J)+W(S,I)*T(I,J)
138 NEXT I,J,S
139 RETURN
```

What It Does: This tool will take the current transformation request, stored in the T matrix, and mathematically combine it with the list of transformations already stored in the C matrix. The resulting updated list of transformations is placed back into the C matrix for use by Tool 100 APPLY TRANSFORM.

Example Use: This tool is an extension of the TRANSLATE A SHAPE tool, and is accessed by a GOTO 130 statement at the end of the translate tool. You will find in later chapters that each transformation tool sends the computer directly to this tool.

Technical Description: The C matrix is temporarily stored in a W matrix while matrix multiplication takes place. Once this is done, the W matrix and the T matrix are multiplied together, and the results are returned to the C matrix.

The T matrix is composed of three rows and three columns, and resembles this:

T Matrix

	0	1	2
0	T(0,0)	T(0,1)	T(0,2)
1	T(0,1)	T(1,1)	T(1,2)
2	T(2,0)	T(2,1)	T(2,2)

The W matrix looks like this:

W Matrix

	0	1	2
0	W(0,0)	W(0,1)	W(0,2)
1	W(0,1)	W(1,1)	W(1,2)
2	W(2,0)	W(2,1)	W(2,2)

There is a standard formula for multiplying two matrices together. Assuming that we want to combine matrices T and W, the formula is as follows:

	0	1	2

0	T(0,0)W(0,0)
	+T(0,1)W(1,0)
	+T(0,2)W(2,0)
		T(0,0)W(0,1)
		+T(0,1)W(1,1)
		+T(0,2)W(2,1)
			T(0,0)W(0,2)
			+T(0,1)W(1,2)
			+T(0,2)W(2,2)

1	T(1,0)W(0,0)
	+T(1,1)W(1,0)
	+T(1,2)W(2,0)
		T(1,0)W(0,1)
		+T(1,1)W(1,1)
		+T(1,2)W(2,1)
			T(1,0)W(0,2)
			+T(1,1)W(1,2)
			+T(1,2)W(2,2)

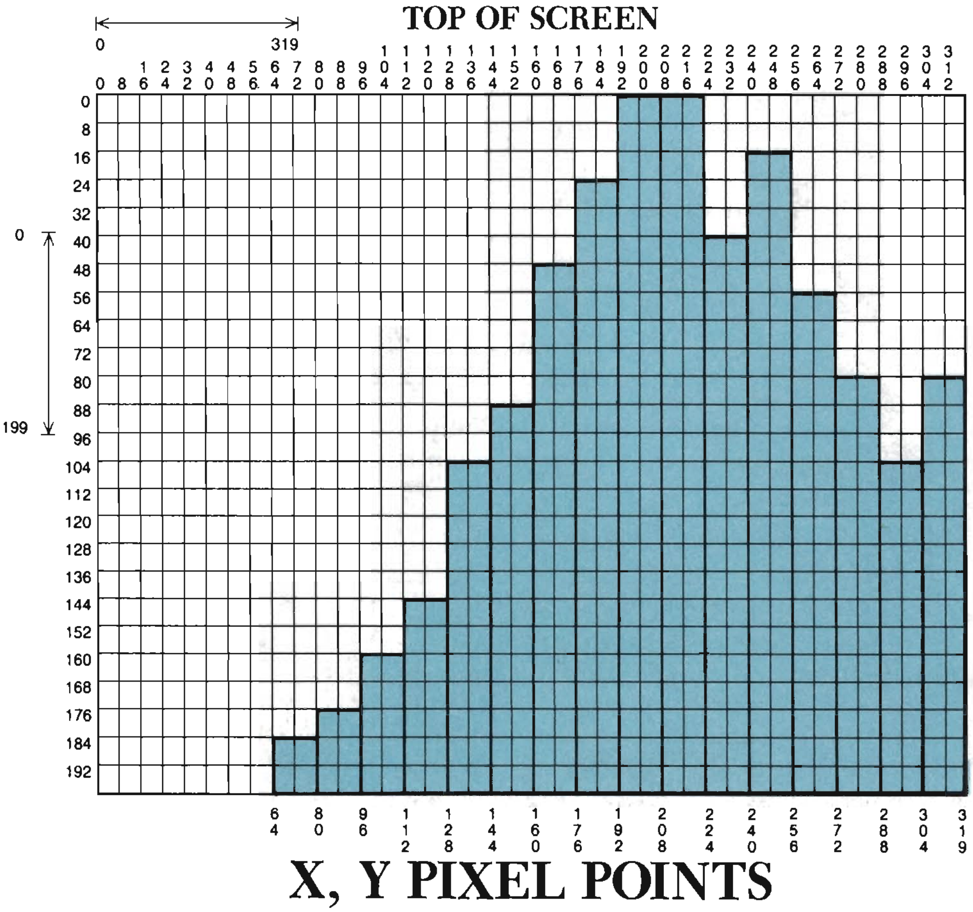
2	T(2,0)W(0,0)
	+T(2,1)W(1,0)
	+T(2,2)W(2,0)
		T(2,0)W(0,1)
		+T(2,1)W(1,1)
		+T(2,2)W(2,1)
			T(2,0)W(0,2)
			+T(2,1)W(1,2)
			+T(2,2)W(2,2)

Notice that the only change in the T array subscripts occurs in the columns of the new matrix; and the only change in the W array subscripts occurs in the rows of the new matrix. This shows that the multiplication can be easily implemented by placing the subscripts in a loop structure.

Painting While Translating

Learning to paint in the midst of transforming a shape is really quite easy. The trick is to quit transforming long enough to paint the most recent form of the shape. In the case of the quilt pattern, for example, the program needs to paint the design immediately following each occasion it is drawn.

A "color bar" on the grid below has been translated at regular intervals across the screen. Painting each translated version of the bar using several different colors can produce a beautiful finished drawing, as you will see in a moment.



Make sure that this chapter's program, as developed so far, is in memory. Next, change line 172 so that C=1018; then RUN the ZAP routine. Finally, re-insert lines 6000, 6010, and 6020 that call Tool 20 when the ← key is pressed.

The DATA statements that describe the color bar are shown below. Notice where the bar is originally located. Its top left endpoint is described as point 1,1. Looking back at the grid for this design, you will find that the first plotted color bar will not appear anywhere near point 1,1. This is because the original bar, as described in the DATA statements below, is never plotted.

```
1Ø18 DATA "COLOR BAR",4,3
1Ø2Ø DATA  1, 1, 15, 1, 15,2Ø1, 1,2Ø1
1Ø22 DATA  2, 2
1Ø24 DATA  Ø, 1, 1, 2, 2, 3, 3, Ø
```

Line 1022 describes the single paint point that is necessary to paint this shape. The paint point (at location 2,2) had to be selected with some care. The shape is going to be translated so that parts of it fall below the visible screen. The clipping tool will then go into action and "clip" all the points outside of the screen. We had to be careful to choose a paint point that, for this particular design, was not on going to be clipped after the shape was translated. We chose one in the shape's top left corner, which we will always keep on the screen.

The next section of program lines does several things. Take a moment to try and deciper it. Then, add them to your program.

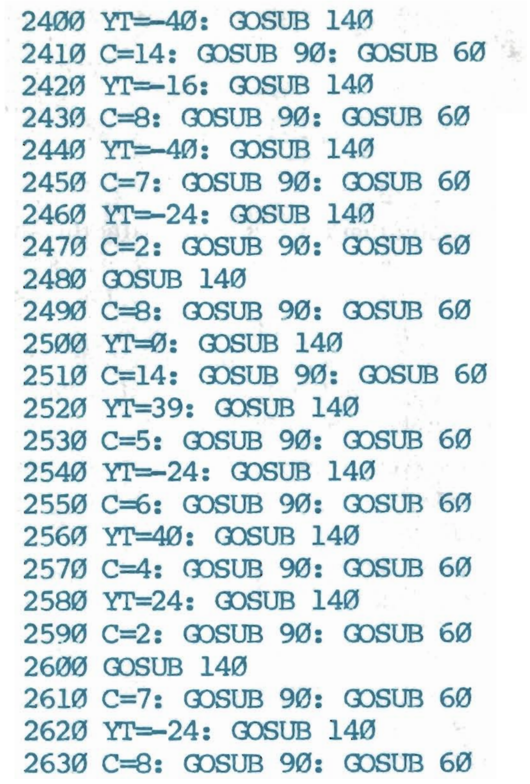
```
2ØØØ GOSUB 1Ø: C=11: GOSUB 3Ø
21ØØ REM::::::RETRIEVE BAR SHAPE
211Ø SE$="COLOR BAR": GOSUB 8ØØ
22ØØ REM::::::CLEAR C MAT
221Ø GOSUB 11Ø
23ØØ REM::::::TRANS,DRAW,PAINT 1ST BAR
231Ø XT=63: YT=183: GOSUB 14Ø
232Ø C=7: GOSUB 9Ø: PP=4: GOSUB 6Ø
```

Briefly, lines 2100-2210 retrieve the shape and set the system up for drawing (by clearing the C matrix). Line 2310 translates the bar 63 pixel locations along the X axis, and down the Y axis 183 pixel locations. This is the starting position of the series of bars to plot. After calling the translate tool (140), the shape is drawn (Tool 90). Before doing anything else to the bar, it is painted.

The bar is painted by setting the painting variable (PP) to 4. This pinpoints the fifth set of coordinates in the shape's Point data list. This paint point, because it is defined in the DATA statements, has been transformed along with the rest of the shape. You don't need to know or care where it is now. All you need to do is identify its location on the Point list. Calling the painting tool (60) results in painting the first translated bar.

The final section of program lines (below) will translate the color bar 15 more times, each time stopping to draw and paint it with a new color. Add these lines to the program also:

```
233Ø REM::::::SET XT AND FINISH BARS
234Ø XT=16: YT= -8: GOSUB 14Ø
235Ø C=6: GOSUB 9Ø: GOSUB 6Ø
236Ø YT=-16: GOSUB 14Ø
237Ø C=5: GOSUB 9Ø: GOSUB 6Ø
238Ø GOSUB 14Ø
239Ø C=2: GOSUB 9Ø: GOSUB 6Ø
```



```
2400 YT=-40: GOSUB 140
2410 C=14: GOSUB 90: GOSUB 60
2420 YT=-16: GOSUB 140
2430 C=8: GOSUB 90: GOSUB 60
2440 YT=-40: GOSUB 140
2450 C=7: GOSUB 90: GOSUB 60
2460 YT=-24: GOSUB 140
2470 C=2: GOSUB 90: GOSUB 60
2480 GOSUB 140
2490 C=8: GOSUB 90: GOSUB 60
2500 YT=0: GOSUB 140
2510 C=14: GOSUB 90: GOSUB 60
2520 YT=39: GOSUB 140
2530 C=5: GOSUB 90: GOSUB 60
2540 YT=-24: GOSUB 140
2550 C=6: GOSUB 90: GOSUB 60
2560 YT=40: GOSUB 140
2570 C=4: GOSUB 90: GOSUB 60
2580 YT=24: GOSUB 140
2590 C=2: GOSUB 90: GOSUB 60
2600 GOSUB 140
2610 C=7: GOSUB 90: GOSUB 60
2620 YT=-24: GOSUB 140
2630 C=8: GOSUB 90: GOSUB 60
```

In line 2340, XT is set to 16 and is never changed again. This enables each color bar to be translated 16 pixel locations to the right of the last one drawn. Because the C matrix is not cleared during all of this, the number of columns to translate the shape from its origin keeps increasing by 16 each time Tool 140 is called.

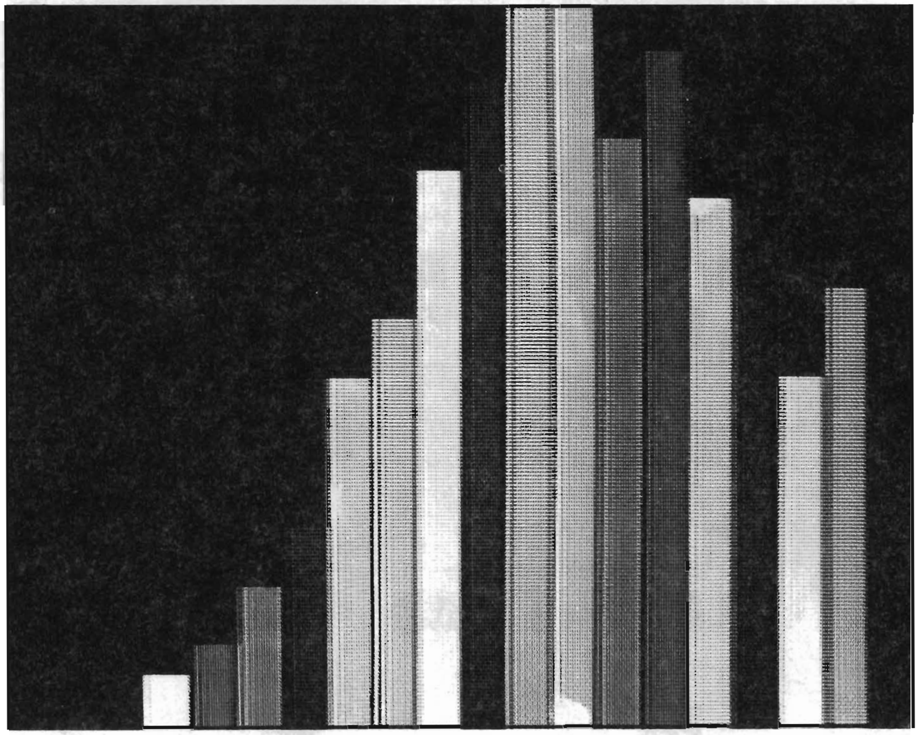
YT, on the other hand, changes for almost every copy of the shape. This situates each copy at a different horizontal level than those immediately around it. Again, because the C matrix stores all of the transformations, each YT will move the shape relative to the last copy of the shape that was drawn.

RUN the program to see the results.

Depending on your monitor, and how well it is adjusted, the screen should display a cluster of color bars, each one displayed as a brilliant, sharp color.

From a design point of view, the bars clustered together appear as a single, whole object rather than individual, separate shapes. The composition is asymmetrically balanced—the bars are more on the right side—yet it *appears* to be in balance. This is because there is an equal amount of background area counterbalancing the "weight" of the shapes.

The "weight" of objects, which we will speak of further, is visually estimated, primarily based on a shape's size in relation to everything else in the design. When the foreground shapes have the same weight (take up an equal amount of space) as the background area, you achieve a balance.

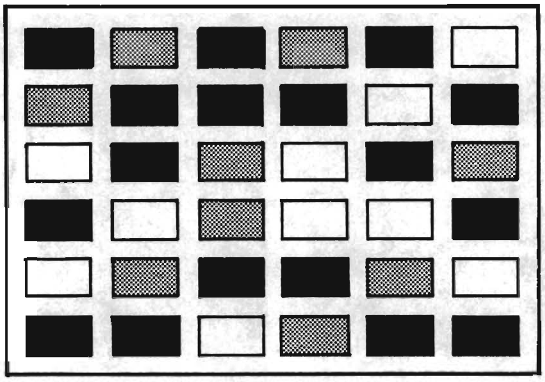


The bright, sharp colors of the bars make them appear to be on top of the background. This is because of the contrast existing between the variety in colors used for the bars and the solid colored background. As the opportunities arise, you should experiment with the use of contrast to make certain objects stand out among others.

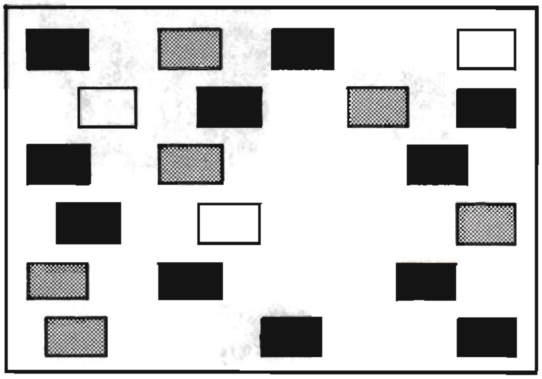
Return to text mode and again examine the program. Translating objects is easy if you know how. However, it is easy to forget how XT and YT compound when the C matrix is not cleared. It's just as easy to forget to set the C matrix in the first place. You should spend a little more time practicing with this tool than you did with tools in the past.

Start out small. Take a simple shape, like as a square, and sketch a design. Then map out on paper the most efficient way to get the design on the screen. Finally, enter the necessary program lines and RUN the program. If it doesn't work as planned, take the time to figure out *why*.

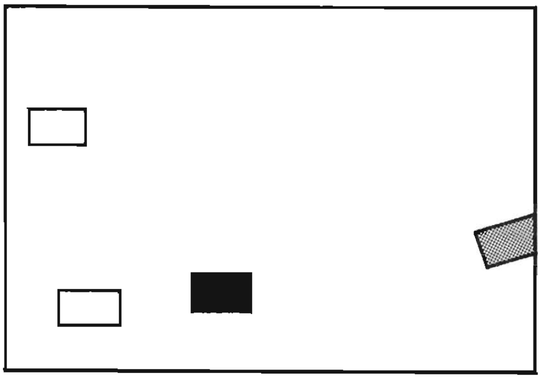
One idea to start you off is shown below. This sketch, which can be plotted with relative ease, expresses a sense of activity. This activity is suggested by the variety of tones used throughout the picture. Its composition is a scattering of light, medium, and dark tones. Although the tones are varied, the spacing is regular. The rectangular shapes are evenly spaced apart for balance at uninterrupted steps.



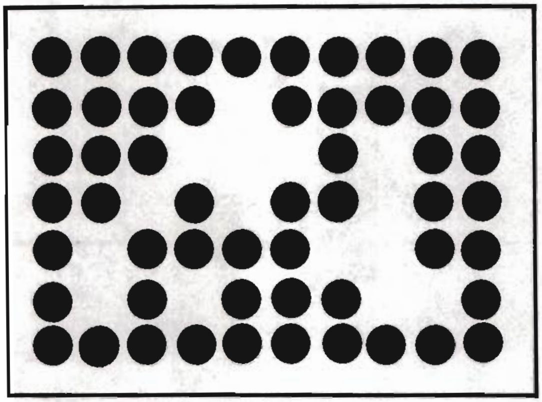
We suggest a planned disorganization of shapes in the next design. Even though the space is irregular, there still appears to be a sense of order and unity. The order results from the equal distribution of tones, and the horizontal arrangement (ordering) of the same geometric shape.



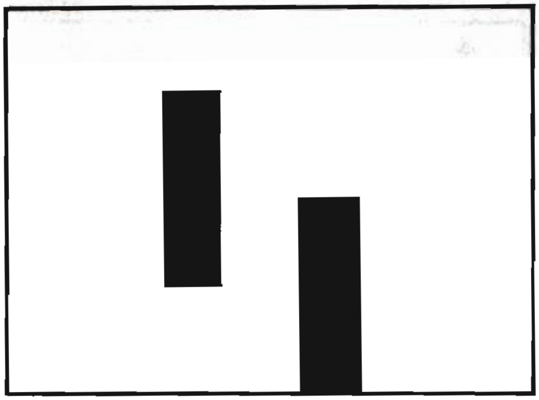
The opposite of a balanced, orderly arrangement is a disorderly and unexpected arrangement. Disorder, irregular spacing, and random placement of shapes can be illustrated as follows:



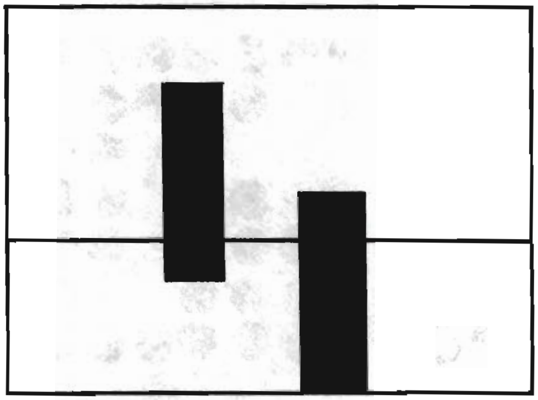
The spacing in our next sketch is irregular and uneven. Although there is much variety by the spacing, order is maintained by the repeating shapes. The space surrounding the shapes creates its own kind of shape—a *negative shape*. Even though the space is not a "real" shape, it appears as one.



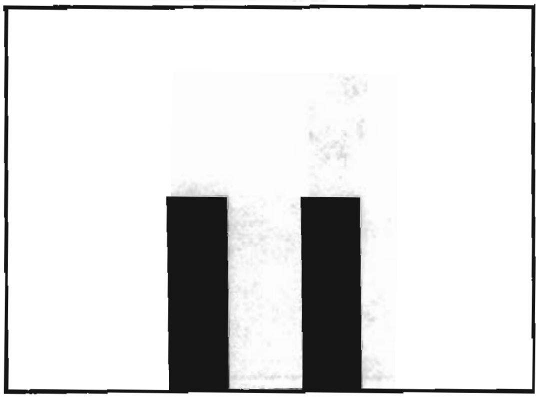
Take a look at the two shapes sketched below. It is not clear from the design which shape is closer than the other. Each appears to float in space.



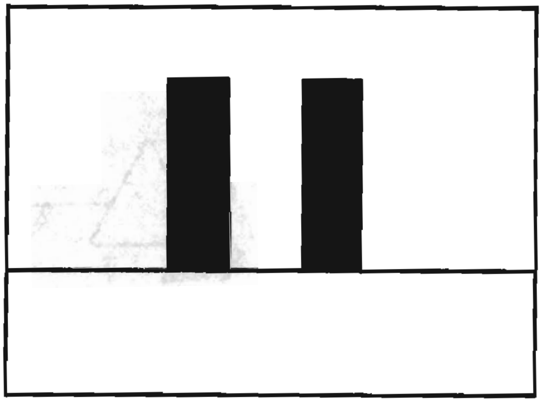
By adding a horizon line, we achieve a sense of depth. Which of the two shapes below appears closer to you now? The distance of a shape can change depending on where it is placed in relation to the horizon line. The shape drawn higher in the picture seems farther away.



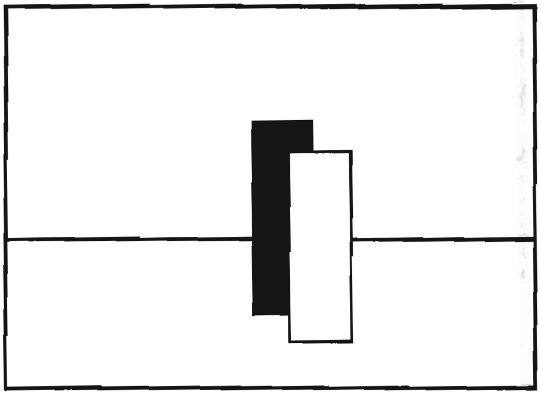
Now which shape is closer? Shapes which are of similar size and are placed at similar vertical positions have similiar distances in space. Since the depth is not indicated by the design, these shapes reside in what is called "flat" space.



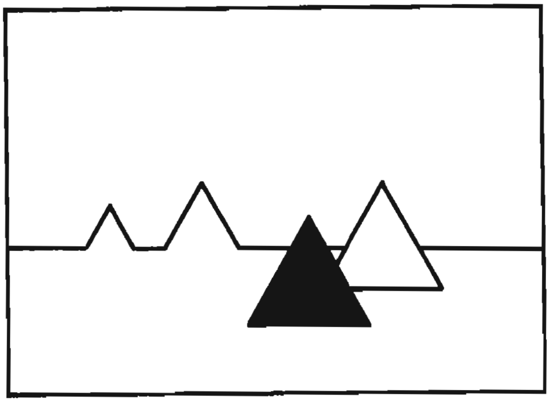
Now we have placed the same two shapes on top of a horizon line (see below). They look as if they are sitting on top of a table. The horizon line, as used here, does not effectively indicate depth. You have no idea how far or how near these shapes are suppose to be.



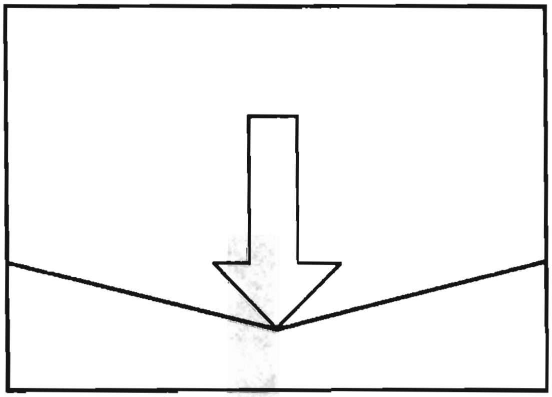
Look how effectively the horizon line in our next sketch creates a feeling of distance. This distance is even more distinct because the shapes are overlapping. Overlapping shapes and the use of a horizon line are just two ways to represent depth in a picture.



The horizon line can be drawn in many ways, as shown below. Here, the horizon repeats the shape of the foreground triangles. This repetition unifies the shapes that make up the picture.



In our final sketch, the horizon line intensifies the direction of the arrow. Tension is created at the point where the arrow and horizon line meet.



Summary

Congratulations! You deserve it. In case you haven't taken stock lately, let us quickly review the graphics capabilities you have now mastered:

—turning high resolution "on and off"
—clearing the high resolution screen
—plotting points
—plotting lines
—defining shapes
—retrieving shapes
—drawing shapes
—clipping shapes
—painting shapes
—translating shapes

That's a lot of new information. Before practicing or moving on to the next chapter, take the time to save your program under the filename "CHAPTER 4".

With this chapter's TRANSLATE A SHAPE tool and its collection of "supporting" tools, you are on your way to creating complex graphics displays with minimal effort. By using it as a "test plotter," you can test the appearance of your shape at various screen locations before deciding on its final placement. With the ability to duplicate shapes, you can repeat a pattern at regular or irregular intervals simply by setting offset values, calling Tool 140, and calling Tool 90.

The steps to moving and copying a shape are outlined below. You should have no problems using your new tool if you follow them carefully.

(1) At the beginning of the program, clear the C matrix (GOSUB 110).

(2) Retrieve the shape to translate.

(3) If necessary, set the color variable (C) to a drawing color.

(4) If the shape is to be offset from its origin, insert a GOSUB 110 statement to clear the C matrix. (This is not necessary if no transformations have taken place since step one above).

(5) Set the variable XT to the distance right (+XT) or left (-XT) you want to move the shape. Set the variable YT tot he distance up (—YT) or down (+YT) you want the shape moved.

(6) Call the TRANSLATE A SHAPE tool (GOSUB 140).

(7) Draw the newly located shape (GOSUB 90).

(8) If you want to paint the shape, do it immediately by setting the painting variable (PP) and calling the PAINT A SHAPE tool (GOSUB 60).

What follows is a checklist of those "easy to forget but terribly important things to do" concering your new tools:

—Has the C matrix been cleared at the beginning of your program? (It should be.)

—Has the C matrix been cleared of any past, unneeded transformations in preparation for the one you are working on? (If you want to transform the *original* shape, it should be.)

—Have you cleared the C matrix when it should, in fact, be accumulating the list of transformations you have entered? (If you want to transform a *transformed* shape, it *shouldn't* be.)

—Is your shape set to be drawn in a color other than the one used for your background? (It should be.)

—Have you modified your point or line list, and, if so, have you changed the DATA statement that gives the count of points and lines in your shape? (You should.)

—When using loops to perform multiple transformations, does GOSUB 140 fall within the loop? (It should.)

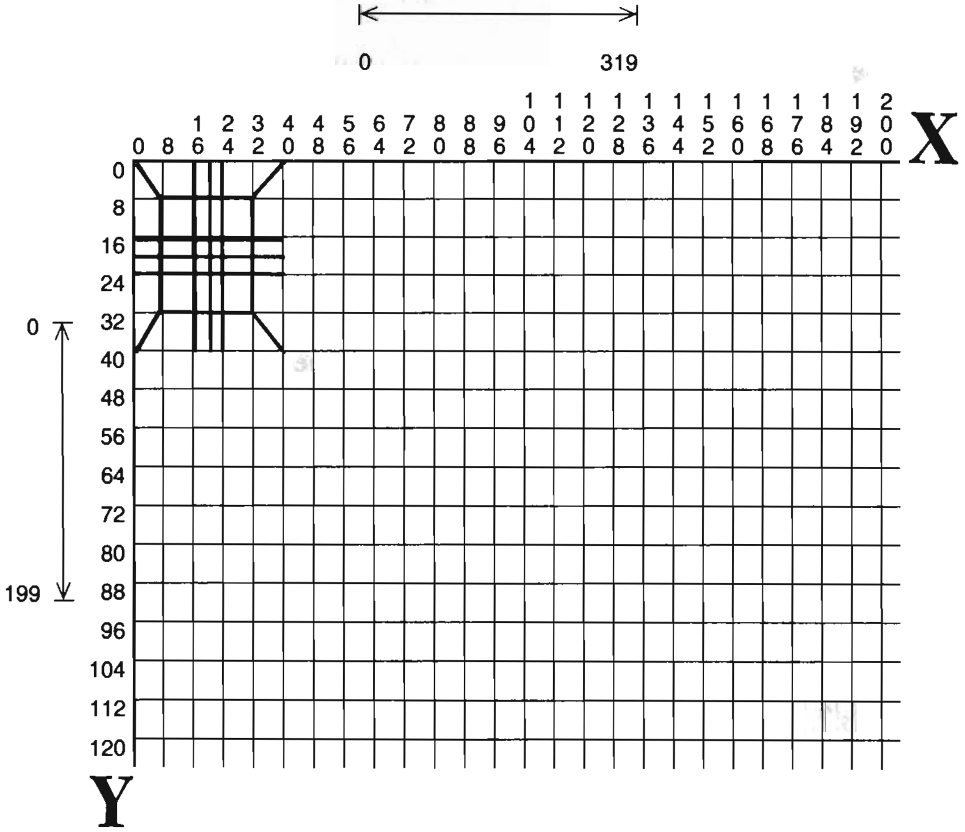
—Are your shape's paint points going to be "clipped" because of the location to which the shape will be transformed? (You probably don't want them to be.)

This checklist does not consider every possible area of omission, but it's a good place to start when things go awry. If you are having problems, don't assume that you have done everything correctly and that the computer is acting up. If your shape does not get translated as expected, look to make sure that you have a GOSUB 110 or a GOSUB 140 statement where needed in the program. Even things as basic as setting the color variable to a drawing/painting color can and will be forgotten. Take the time to check over each area of your program. Follow through each line, mapping out in your mind how the computer will respond to each program line it comes to.

There are two exercises for this chapter, given below. You are ready for tougher problems and less discussion of the answers at this point. For this reason, these exercises outline what you need to do, and are then followed by one solution program (although there could be several). We offer no discussion on the solution program.

Exercise 1

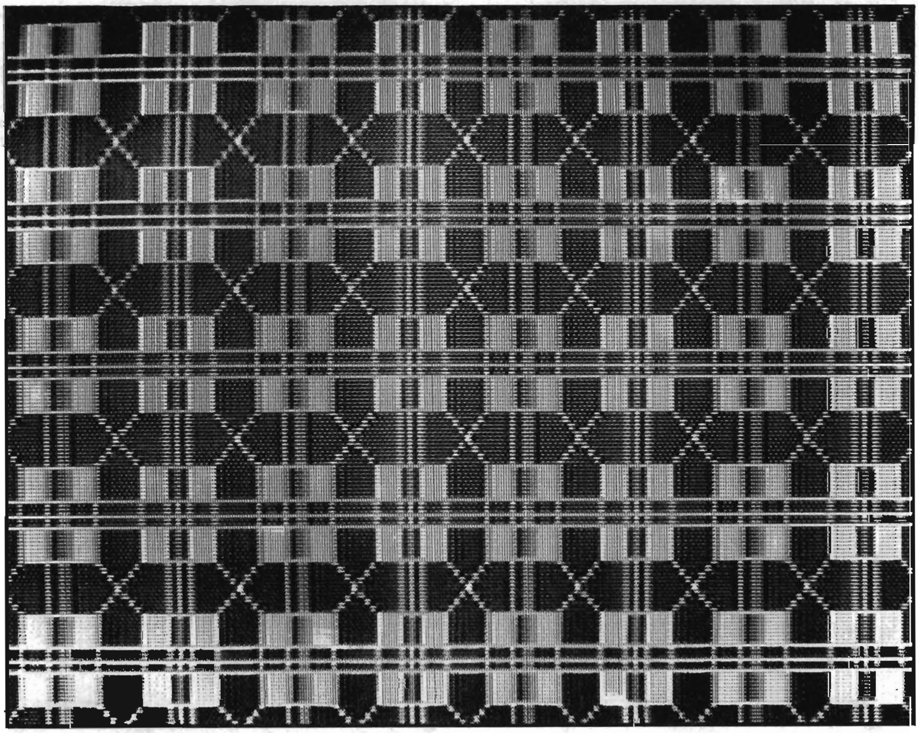
Write down a Point data list and Line data list for the "shape" drawn on the grid below. The Point list should have twenty endpoints, numbered 0 through 19. In addition, add four paint points, numbered 20 through 23, to paint the four outer



squares created by the design (paint point 10,10 is located in one of the four squares to which we refer). The Line data list for this shape should describe fourteen lines, numbered 0 through 13.

Using Gray 2 as the background color, and yellow for the foreground color, repeat this shape five times across your screen in eight different rows. The repeated shapes should meet at every corner, but never overlap. As you draw each shape, paint the four squares within it (paint in yellow). You may name the shape any name you like. In the solution, we use the name CROSS-PATCH.

Solution 1

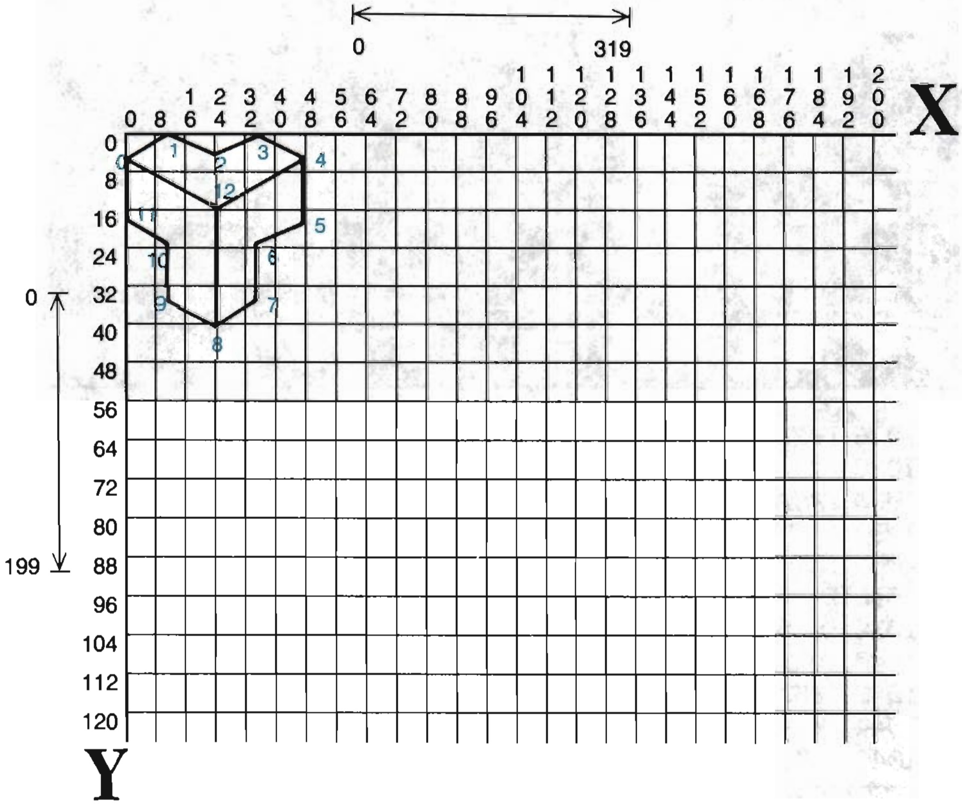


```
1Ø26 DATA "CROSS-PATCH",23,13
1Ø28 DATA 16, Ø,2Ø, Ø,23, Ø,39,16
1Ø3Ø DATA 39,2Ø,39,23,23,39,2Ø,39
1Ø32 DATA 16,39, Ø,23, Ø,2Ø, Ø,16
1Ø34 DATA  7, 7,32, 7,32,32, 7,32
1Ø36 DATA  Ø, Ø,39, Ø,39,39, Ø,39
1Ø38 DATA 1Ø,1Ø,28,1Ø,28,28,1Ø,28
1Ø4Ø DATA  Ø, 8, 1, 7, 2, 6, 3,11
1Ø42 DATA  4,1Ø, 5, 9,12,13,13,14
1Ø44 DATA 14,15,15,12,12,16,13,17
1Ø46 DATA 14,18,15,19
2ØØØ GOSUB 1Ø: C=11: GOSUB 3Ø
```

```
2010 SE$="CROSS-PATCH": GOSUB 800
2020 GOSUB 110: C=7
2030 FOR L=1 TO 5
2040 FOR M= 1 TO 8
2050 GOSUB 90
2060 PP=20: GOSUB 60
2070 PP=21: GOSUB 60
2080 PP=22: GOSUB 60
2090 PP=23: GOSUB 60
3000 XT=40: GOSUB 140
3010 NEXT M
3020 XT=-320: YT=40: GOSUB 140
3030 YT=0: NEXT L
```

Exercise 2

The next design has *perspective*; that is, it creates the illusion of three dimensions. In this exercise, you need to fill your screen with this shape. Use Gray 2 as the background color, and Blue (color code 6) as the foreground color. The shapes should overlap one column and one row (thus, just touching each other). Offset the very first shape, setting XT=-18 and YT=-10. As you repeat this shape, paint the top section in Blue.



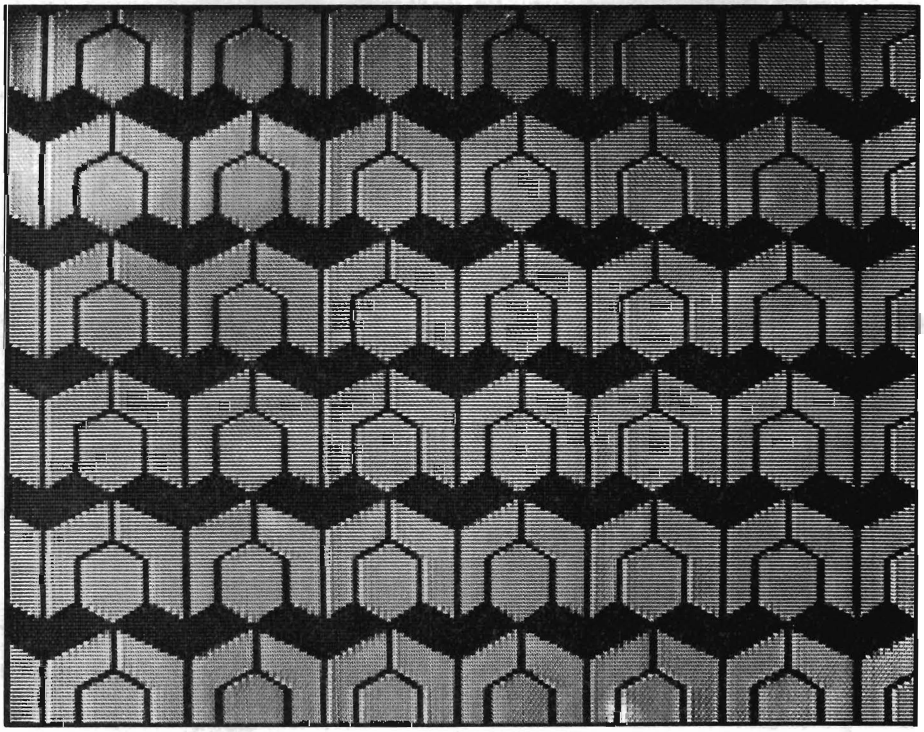
Because the endpoint coordinates of this design cannot easily be determined by looking at the grid, we provide the Point data list below. Although no paint point is shown on the grid above, one has been identified in this Point list.

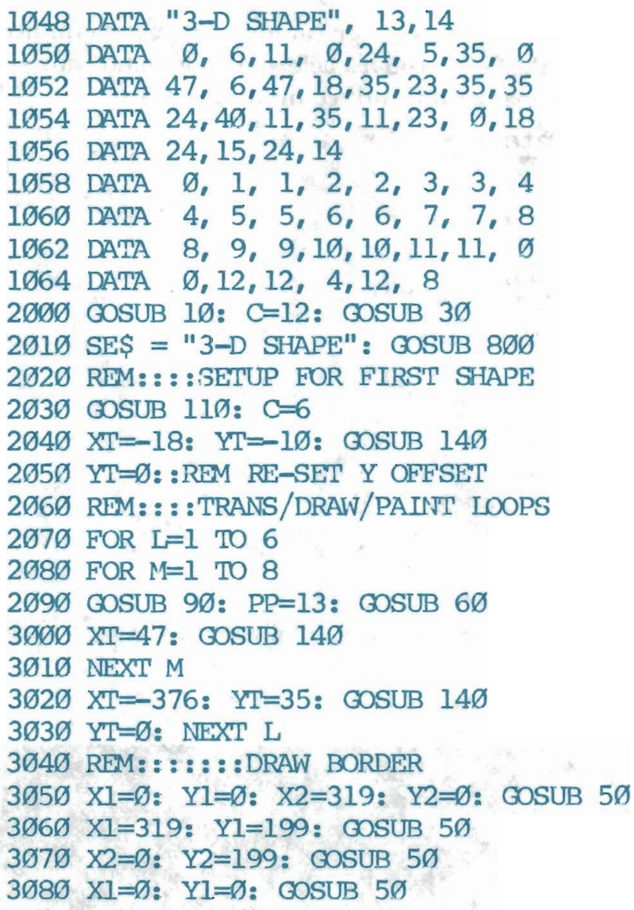
Point Data List

Endpt. #Ø	Ø, 6
Endpt. #1	11, Ø
Endpt. #2	24, 5
Endpt. #3	35, Ø
Endpt. #4	47, 6
Endpt. #5	47,18
Endpt. #6	35,23
Endpt. #7	35,35
Endpt. #8	24,4Ø
Endpt. #9	11,35
Endpt. #1Ø	11,23
Endpt. #11	Ø,18
Endpt. #12	24,15
Paint Point #13	24,14

When the design is complete, add program lines that will draw a border around the outer edges of the high resolution screen (plotting area).

Solution 2





```
1Ø48 DATA "3-D SHAPE", 13,14
1Ø5Ø DATA  Ø, 6,11, Ø,24, 5,35, Ø
1Ø52 DATA 47, 6,47,18,35,23,35,35
1Ø54 DATA 24,4Ø,11,35,11,23, Ø,18
1Ø56 DATA 24,15,24,14
1Ø58 DATA  Ø, 1, 1, 2, 2, 3, 3, 4
1Ø6Ø DATA  4, 5, 5, 6, 6, 7, 7, 8
1Ø62 DATA  8, 9, 9,1Ø,1Ø,11,11, Ø
1Ø64 DATA  Ø,12,12, 4,12, 8
2ØØØ GOSUB 1Ø: C=12: GOSUB 3Ø
2Ø1Ø SE$ = "3-D SHAPE": GOSUB 8ØØ
2Ø2Ø REM::::SETUP FOR FIRST SHAPE
2Ø3Ø GOSUB 11Ø: C=6
2Ø4Ø XT=-18: YT=-1Ø: GOSUB 14Ø
2Ø5Ø YT=Ø::REM RE-SET Y OFFSET
2Ø6Ø REM::::TRANS/DRAW/PAINT LOOPS
2Ø7Ø FOR L=1 TO 6
2Ø8Ø FOR M=1 TO 8
2Ø9Ø GOSUB 9Ø: PP=13: GOSUB 6Ø
3ØØØ XT=47: GOSUB 14Ø
3Ø1Ø NEXT M
3Ø2Ø XT=-376: YT=35: GOSUB 14Ø
3Ø3Ø YT=Ø: NEXT L
3Ø4Ø REM::::::DRAW BORDER
3Ø5Ø X1=Ø: Y1=Ø: X2=319: Y2=Ø: GOSUB 5Ø
3Ø6Ø X1=319: Y1=199: GOSUB 5Ø
3Ø7Ø X2=Ø: Y2=199: GOSUB 5Ø
3Ø8Ø X1=Ø: Y1=Ø: GOSUB 5Ø
```

Chapter Five

SCALING SHAPES

Scaling is one of the harder graphic concepts to learn. This is said not in an attempt to discourage you, but rather to prepare you for a lesson that requires more patience and perseverance than those before it. The hardest part of scaling is knowing *in advance* where your shape is going to end up on the screen. With careful attention and planning, though, you can become very adept with this chapter's SCALE A SHAPE tool.

Scaling is a type of transformation, as it will physically change the appearance of your shape. The main use of scaling is to be able to "stretch" and "shrink" your shapes, horizontally and/or vertically. This is done through the use of *scale factors.* A scale factor is a value that specifies by how much you want to scale a shape. A scale factor of 2 indicates that you want the shape scaled to twice its size. A scale factor of .5 means you want the shape scaled to half its size.

The question then arises as to whether you want the shape scaled in height or in width. You control this by using variables. Assigning a scale factor to XS (X-Scale) means you want the shape scaled along its X axis. This will scale the shape's width. Assigning a scale factor to YS (Y-Scale) means you want the shape scaled along its Y axis. This will scale the shape's height. Thus, if YS=5 and XS=2, your shape will be scaled to five times its current height, and two times its current width.

There are two immediate uses for such a tool. The first is that scaling shapes creates new shapes. This can add considerable variety to a design. Draw one man, and you can scale him to form tall men, short men, wide men, lean men, tall and lean men, tall and wide men, short and lean men, and so forth. In fact, scaling one shape can create so many different shapes, that a design can be formed in its entirety. One common example is that of a square scaled to an increasingly smaller size. The result is an interesting tunnel of "infinite" squares.

A second application for scaling is that shapes used in one design can be scaled up or down for use in other designs. Whenever you start work on a new picture you can "sort" through your library of shapes, retrieve a few of the more interesting or applicable ones, and then scale them to the necessary size.

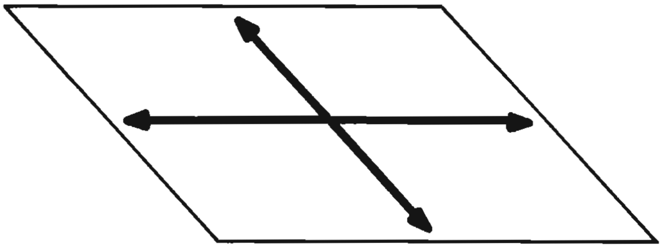
There is no trick to knowing what happens to a shape that is scaled. Scaling in the X direction (XS) alters the width of the shape. Scaling in the Y direction (YS) alters the height of the shape. A scale factor that is greater than 1 (>1) will increase the shape's height/width. A scale factor that is less than 1 (<1) will decrease the shape's height/width. If the C matrix is cleared out, scaling will affect the original form of the shape. If the C matrix is storing a set of transforms, scaling will alter the current form of the shape.

The trick to scaling, as mentioned, is knowing where the scaled shape will end up on the screen. To understand this, you will need to learn a little more about the X,Y coordinate system you have been using—the Cartesian coordinate system.

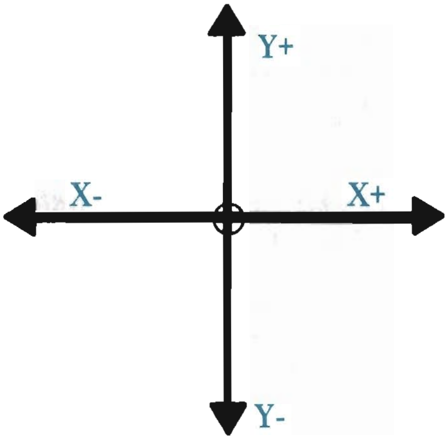
Using the Cartesian Coordinate System

The Cartesian coordinate system is universally used for displaying data graphically. Sometime before the completion of high school most students will be confronted with this system of measurement. Although it can be set up to represent three-dimensional space (height, width and depth), we are only concerned here with this system as based on two-dimensional space (height and width).

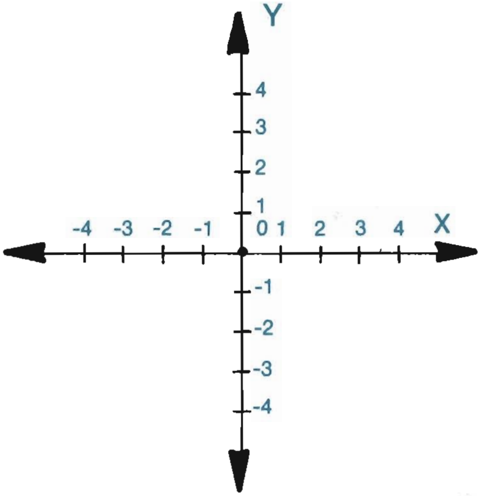
In this system, there are two axes that run perpendicular to each other: the X axis and the Y axis. Together, they form a "plane," or flat surface, that cuts through space. Planes, such as this, theoretically run forever in all four directions.



The point at which the X axis and Y axis intersect can be thought of as the anchor point of the plane. When units of measurement are applied to the plane, this anchor point has a measure of 0. Units running along the X axis increase (+) as they move to the right of the anchor, and decrease (-) as they move to its left. Units running along the Y axis increase as they move above the anchor point, and decrease as they move below the anchor:

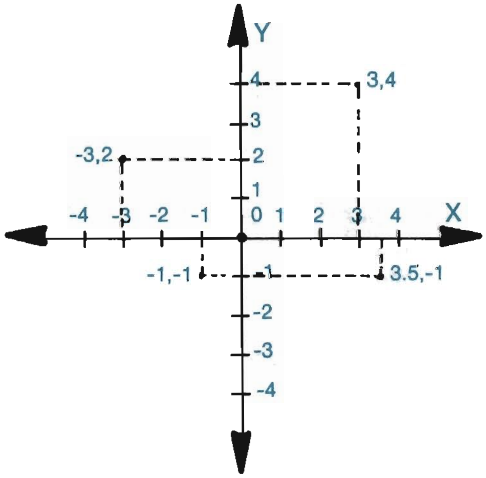


Assuming that each unit has a measure of 1, this could be demonstrated as:



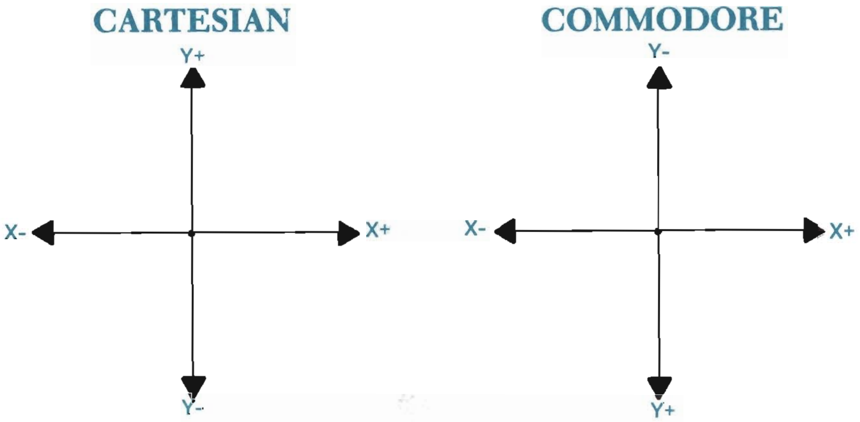
The values indicated along an axis (e.g., 0, 1, 2, 3 ...) mark off even increments, and are called "tick marks." This term stems from the ticking of a clock, which marks off even increments of time.

Any point that lies on the plane can be identified by its X,Y coordinates. The X coordinate is the point's position along the X axis, and the Y coordinate is the point's position along the Y axis. Together, they form the X,Y coordinates of the point. You are by now very familiar with the use of X,Y notations to specify points:



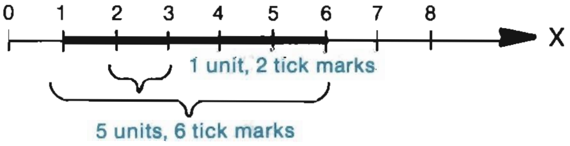
In fact, by now you should be comfortable with this coordinate system. You have been using a modified version of it all along to describe the points and lines that make up your two-dimensional shapes.

There are two main differences between the Cartesian coordinate system and the Commodore coordinate system. First, the direction in which the Y axis increases/decreases is reversed from one system to the other:

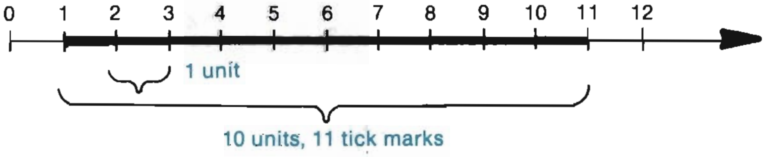


Second, while a true plane is said to be indefinite, with an infinite number of points making up each section of it, the Commodore has a limited number of points along each axis. There is no point at location 3,8.2; neither is there one at 5.9,7. There are only points at each integer location. Because of this limitation, the Commodore will round fractional coordinates down to the their nearest integer representation.

To scale a line on the Cartesian system, its length is increased/decreased by the scale factor. The length is determined by the number of *units* the line encompasses. A line drawn from point 1,0 to 6,0, for example, is 5 units in length:

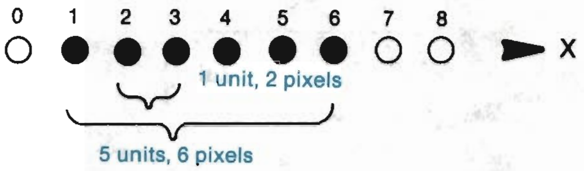


Notice that each "tick mark" (1, 2, 3, ... 6) is not a unit. If it were, then the line would be 6 units long. Instead, the distance *between* each marker is a unit. Scaling this line to two times its length produces the following line:

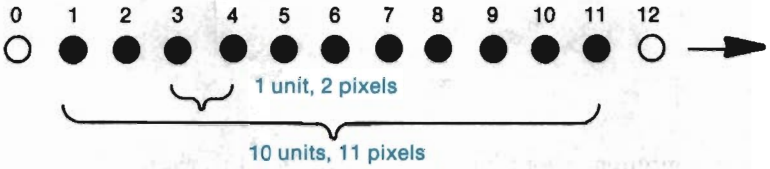


The line now has twice the length it previously had.

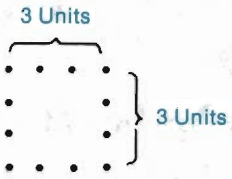
The same ideas are true on the Commodore. A line's length is determined by the number of units it encompasses. A unit is not the same thing as a pixel. A unit is the distance between two pixels. Drawing the same line (from 1,0 to 6,0) on the Commodore produces the following:



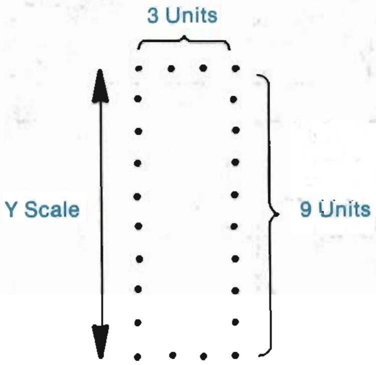
This line, although made up of 6 pixels, has a length of 5. Scaling it to twice its length gives you a line that is 10 units in length:



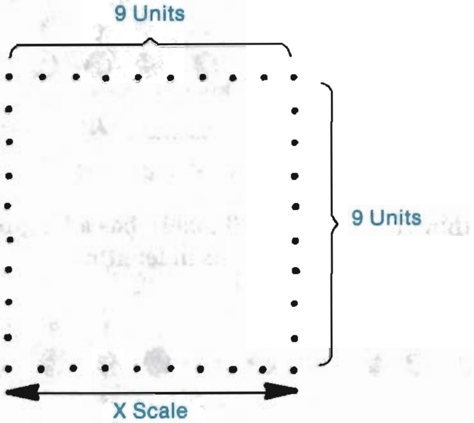
Since a line is scaled according to its length in units, it follows that a shape is scaled in the same manner. Take, for example, the square shape below. It is a 4 pixel by 4 pixel shape. However, since scaling deals in units and not pixels, we must look at its measurement in terms of units. This shape is 3 units by 3 units.



Scaling the height of this shape increases the number of units along its Y axis. If this shape were scaled to three times its height, the new shape will have three times as many units along the Y axis:



In the same way, scaling the shape's width increases the number of units along its X axis. The shape above, when scaled three times in width, becomes:

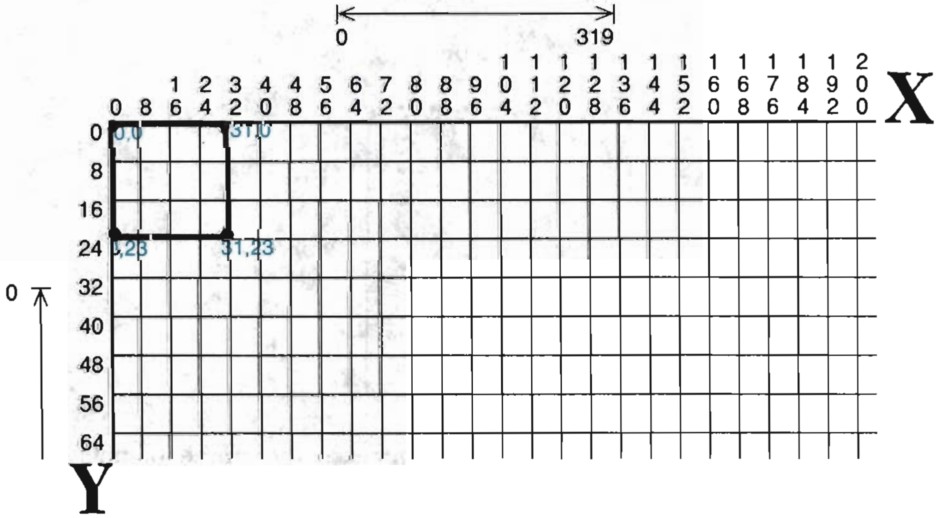


Think for a moment about how scaling changed this shape. We started with a square that was 4 pixels by 4 pixels. After scaling its height and width by three, we arrived at a square that was 10 pixels by 10 pixels. If scaling dealt in pixels and not units, this scaled shape would have been 12 pixels by 12 pixels in size. Since scaling does *not* work in this fashion, you cannot multiply the scale factor by the number of pixels to obtain a scaled height/width.

With this basic understanding of scaling, you are ready to begin working with some simple exercises. As you do, the mathematics behind scaling will be explained.

The Mathematics of Scaling

The math involved in scaling is simple in and of itself. The reason the math works, however, is not always obvious. To help you through this necessary topic, we will use the square diagrammed below.



Take the time now to LOAD your "CHAPTER 4" file. Next, change line 172 so that C equals 1008, and run the ZAP routine. Finally, enter the SCALE A SHAPE tool as follows:

```
150 REM::::::SCALE A SHAPE
151 GOSUB 12Ø
152 XS=XS-(XS=Ø): YS=YS-(YS=Ø)
153 T(Ø,Ø)=XS: T(1,1)=YS
154 GOTO 13Ø
```

Compare this new tool with your TRANSLATE A SHAPE tool at line 140. They are almost identical. The only difference is that the values for XS and YS will be stored at different locations in the T matrix than the values for XT and YT.

The following program lines define, retrieve, and draw the square. Type them into your program.

```
1ØØ8 DATA "SQUARE",3,3
1Ø1Ø DATA  Ø, Ø,31, Ø,31,23, Ø,23
1Ø12 DATA  Ø, 1, 1, 2, 2, 3, 3, Ø
2ØØØ GOSUB 1Ø: C=7: GOSUB 3Ø
2Ø1Ø SE$="SQUARE": GOSUB 8ØØ
2Ø2Ø GOSUB 11Ø
2Ø3Ø C=2: GOSUB 9Ø
6ØØØ GET A$
6Ø1Ø IF A$="←" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
```

RUN the program to view the original form of this shape. It should appear in the top left corner of your screen. Press ←to return to text mode. By adding two simple program lines, you will create a new shape. Program lines 2040 and 2050 (below) will draw a square that is twice as high and twice as wide as the original square:

```
2Ø4Ø XS=2: YS=2: GOSUB 15Ø
2Ø5Ø GOSUB 9Ø
```

RUN the program again to see both versions of this shape. Notice that the scaled square is situated in the same corner area as the unscaled square. Obviously, because it has been scaled to a larger size, it projects farther right and down than the original shape. Return to text mode.

Let's try scaling down to a smaller size. Add program lines 2060 and 2070, as shown below, to scale the shape by 2/10ths its height and width. Notice that we said *by* 2/10ths and not *to* 2/10ths. There's a difference. RUN the program to see what we mean.

```
2Ø6Ø XS=.8: YS=.8: GOSUB 15Ø
2Ø7Ø GOSUB 9Ø
```

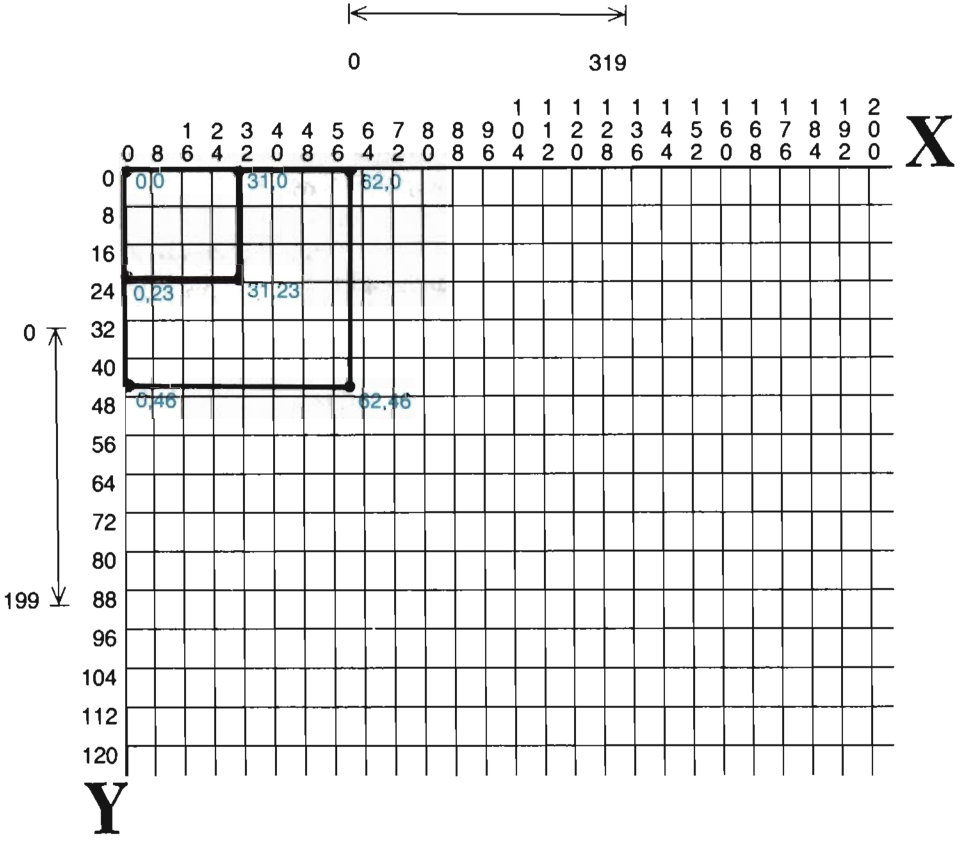
Was the shape scaled as you expected? Scaling a shape by 2/10ths (i.e., .8) produces a smaller but almost equal shape. If you look at your screen, you will find that the newly added square is *larger* than the original square. Take a moment to think why this happened.

As you may have guessed, the C matrix needed to be cleared with a GOSUB 110 in order to scale the original square. Since this matrix held a transformed shape, the shape that was scaled was the previously transformed shape. Look again at your screen, and you will find that the newest shape is approximately 2/10ths smaller in height and width than the largest square.

Add a line number 2055 containing GOSUB 110, and RUN the program. This time, your final square is 2/10ths smaller than the original square. Here's how all of this works.

Scaling is accomplished by altering each endpoint in the shape to scale. Once the endpoints have been moved to their scaled positions, the Line data in L% can connect them to form the scaled shape.

On the grid below we have outlined the square as defined in your DATA statements, and the same square scaled to twice the height and width. It is immediately apparent that the scaled square does not have twice the number of pixels in height and width as the smaller one. It does, however, have twice the number of units in height and width.



The number of units in a plotted line is equal to *# OF PIXELS -1.* This is true for any line in any shape. To increase a shape's number of units (scale up), the shape's endpoints must be moved farther apart. To decrease a shape's number of units (scale down), the shape's endpoints must be moved closer together. When scaling a shape's width, the endpoints must be moved farther apart or closer together horizontally (along the X axis). When scaling a shape's height, the endpoints must be moved farther apart or closer together vertically (along the Y axis).

Look at the last grid again. The top line in the original square runs from 0,0 to 31,0. This covers 32 pixels, or 31 units. On scaling the square "up," this line runs from 0,0 to 62,0. This covers 63 pixels, or 62 units. Thus, the number of units doubles—from 31 to 62—exactly as it should.

The formula used to scale the endpoints of a shape is:

Old X • XS = New X
Old Y • YS = New Y

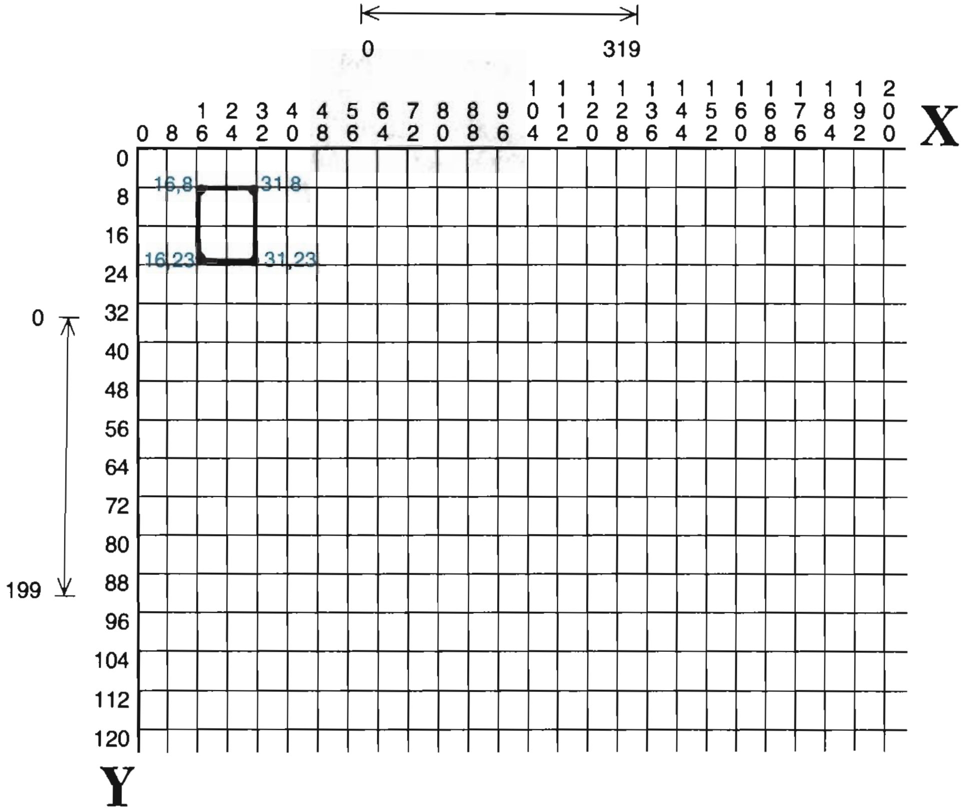
That's it. No other math is necessary. Look once more at the last grid. Notice that if you multiply each X coordinate and each Y coordinate in the smaller square by two, you arrive at the endpoints for the larger, scaled square. This can be shown as follows:

Endpoint #0:	Old X (0) • XS (2)	= New X (0)
	Old Y (0) • YS (2)	= New Y (0)
	Scaled Endpoint #0	= 0,0
Endpoint #1:	Old X (31) • XS (2)	= New X (62)
	Old Y (0) • YS (2)	= New Y (0)
	Scaled Endpoint #1	= 62,0
Endpoint #2:	Old X (31) • XS (2)	= New X (62)
	Old Y (23) • YS (2)	= New Y (46)
	Scaled Endpoint #2	= 62,46
Endpoint #3:	Old X (0) • XS (2)	= New X (0)
	Old Y (23) • YS (2)	= New X (46)
	Scaled Endpoint #3	= 0,46

There is only one exception to this method of scaling. If either variable (XS or YS) is set to 0, the computer will *not* multiply the associated coordinates by 0. Instead, it will set that variable (XS or YS) to 1, and then perform the necessary multiplication.

This safeguarding allows you to forget to set XS or YS with no significant impact on your shape. As you probably know, all variables are set to zero by the computer at the beginning of each program execution. Suppose you wanted to scale a shape to two times its width, while leaving its height unchanged. If you set XS to two, and then called the SCALE A SHAPE tool, YS would still be set at zero. This means that every Y coordinate in the shape would be multiplied by zero, which results in each Y coordinate becoming zero. This would transform your shape into a ***straight line*** along the X axis. By having the subroutine automatically change XS or YS to one if they are found to be set to zero, this mishap is bypassed.

Let's change your shape's DATA statements to define a different square. See if you can make the necessary adjustments to line 1010 to describe the shape below:



The DATA statement that will define the new shape is:

```
1010 DATA 16, 8,31, 8,31,23,16,23
```

Make sure your program line 1010 is the same as that above. Delete lines 2055, 2060 and 2070, and RUN the program. The new square will be plotted; then it will be scaled to twice its height and width, and plotted again.

Watch this time as the scaled square is plotted to the right and below the original square. The scaled shape is not, as before, situated in the same general location as the unscaled shape. The reason lies in the mathematics of scaling. Press ← to return to text mode. Examine the DATA statements for this shape.

Let's apply the scaling formula to the four endpoints in this shape:

Endpoint #0: Old X (16) • XS (2) = New X (32)
Old Y (8) • YS (2) = New Y (16)
NEW ENDPOINT #0: 32,16

Endpoint #1: Old X (31) • XS (2) = New X (62)
Old Y (8) • YS (2) = New Y (16)
NEW ENDPOINT #1: 62,16

Endpoint #2: Old X (31) * XS (2) = New X (62)
Old Y (23) * YS (2) = New Y (46)
NEW ENDPOINT #2: 62,46

Endpoint #3: Old X (16) * XS (2) = New X (32)
Old Y (23) * YS (2) = New Y (46)
NEW ENDPOINT #3: 32,46

The important thing to note is that none of these endpoints started out on an axis. When an endpoint rests on one or both axes, it is *anchored*. This means that it will remain on that axis, even after the scaling multiplication is applied to it. The reason is simple. By resting on the X axis, it has an X coordinate of 0. Multiplying this by any X scale factor still produces an X coordinate of 0, leaving it on the X axis. The same is true if it rested on the Y axis. This new square we are working with has no endpoints anchored on an axis. When the scale factors are applied to the endpoints, all of them are increased accordingly. Changing a coordinate (either by increasing its value or decreasing it) will move the coordinate.

Change the scale factors to the following:

```
2Ø4Ø XS=8: YS=8: GOSUB 15Ø
```

RUN the program. You will find that the gap between your original square and your scaled square has widened. Return to text mode, and change the scale factors once more:

```
2Ø4Ø XS=.5: YS=.5: GOSUB 15Ø
```

See if you can anticipate where the scaled shape will appear this time. After you've formulated a guess, RUN the program to see if you are correct.

Scale factors of less than one will reduce the value of your coordinates. In the current example, all endpoints were multiplied by .5, which is the same thing as dividing them by two. This decreases their value, bringing them closer to point 0,0.

Let's add a FOR/NEXT loop to scale this shape several times. Add the following lines:

```
2Ø35 FOR F = 1 TO 4
2Ø6Ø NEXT F
```

Before running the program, change line 2040 to set the scale factors to two, as follows:

```
2Ø4Ø XS=2: YS=2: GOSUB 15Ø
```

RUN the program. The square will be plotted five different times. Each new square will be twice as high and twice as wide as the preceding one. The final square will be so large that it is only partially on the screen.

How would you change the program to prevent scaling at such a rapid rate? That is, how could you scale the original shape several times, each time just a little larger than the time before? One necessary addition is a program line clearing out the C matrix. This will allow you to work with the original shape during each loop through the scaling process. Add program line 2055 as shown below:

```
2Ø55 GOSUB 11Ø
```

This change alone, however, is not enough. RUN the program to see why. Notice that after the shape is drawn, scaled, and re-drawn, nothing else seems to happen. There *is* something happening, though. The shape is being scaled to the same size and drawn over four times. Return to text mode.

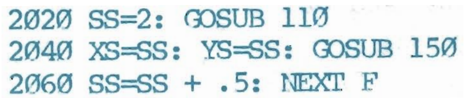
If you mentally follow the program through, you will find that three major steps take place within the loop:

(1) The original shape is scaled by XS=2: YS=2.

(2) The scaled shape is drawn.

(3) The C matrix clears, and you return to step one.

What you need is a way to gradually increase the scale factors in order to draw new shapes. This can be done using another variable. We will use the variable SS (Scaling-Step) to increase our XS and YS variables each time the loop is processed. Change the following program lines:



```
2020 SS=2: GOSUB 110
2040 XS=SS: YS=SS: GOSUB 150
2060 SS=SS + .5: NEXT F
```

RUN the program. This time, scaling occurs at a much slower rate. Each time through the loop, the scale factors are increased by .5. This scales the original shape by 2, 2.5, 3, and 3.5. When five squares have been plotted on the screen, press ← to return to text mode.

Spend some time trying out your own scale factors. Change the DATA statements to describe a different shape and scale it. Try translating the shape as you scale it. Use loops to scale shapes over and over again. Be careful of using loops that don't clear the C matrix, though. Each time through the loop, the translate and scale variables will increase. If the loop processes enough times, the values for these variables can become so large that you generate an ILLEGAL VALUE or ILLEGAL QUANTITY error.

Another area to be careful of lies in the use of variables. At this point, several of the more than 2,000 available variable names are being used specifically by the subroutine tools. It is important that you do not use these variables accidently for any other purpose (for example, in FOR/NEXT loops). The current list of variables used by your tools is:

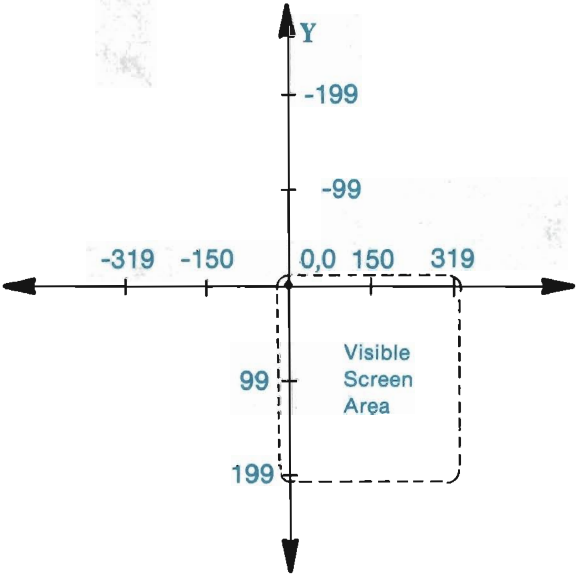
C	E1%	E2%	I	J	MU	ND	NL	PP	S
S1	S2	X	X1	X2	XH	XL	XS	XT	Y
Y1	Y2	YH	YL	YS	YT				

Finally, be sure that you set all necessary variables before calling a tool. Even if you simply want a variable set to zero, be sure to set it in the program. Don't count on it automatically being set to zero, for you may be forgetting a place in the program where you set the variable to some other value.

Anchoring Shapes for Orientation

We need to discuss how to anchor your shapes before scaling them. When a shape is defined on or around point 0,0, it is anchored. This means it will remain in that same general location after it has been scaled. To see how this is possible, examine the drawing below.

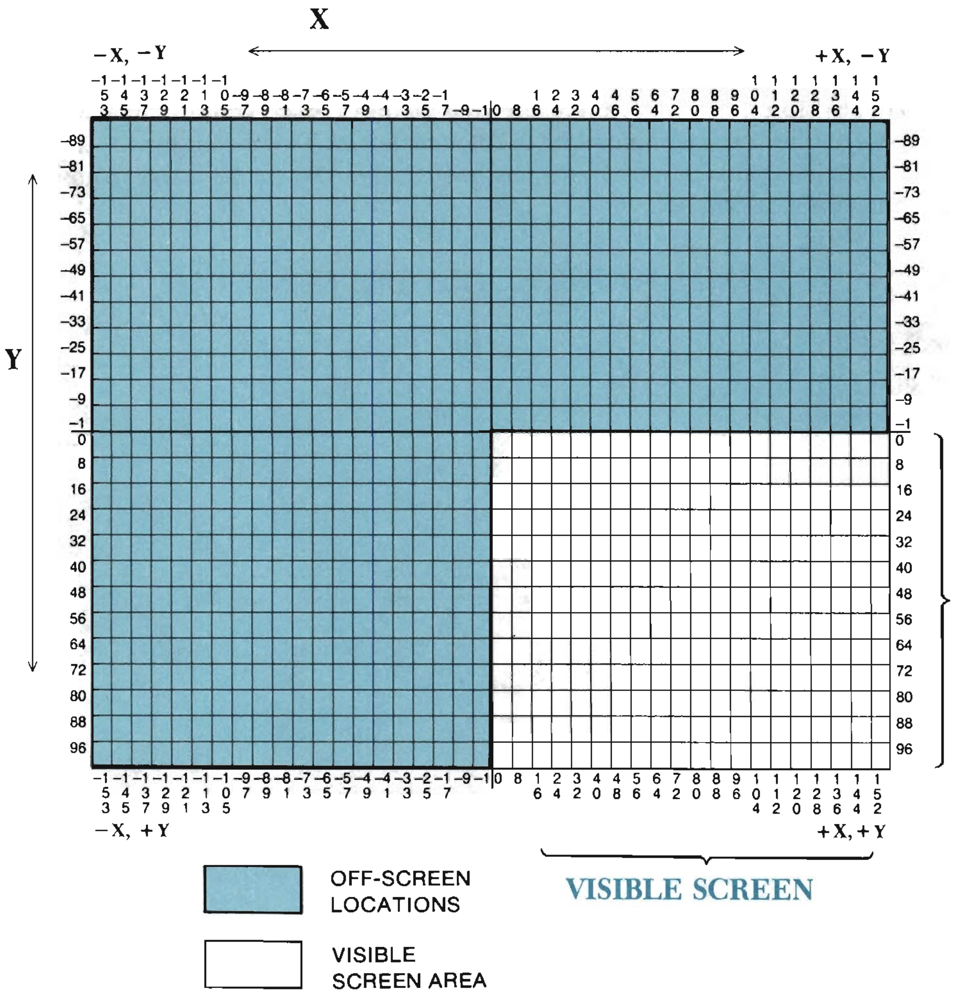
COMMODORE COORDINATE SYSTEM



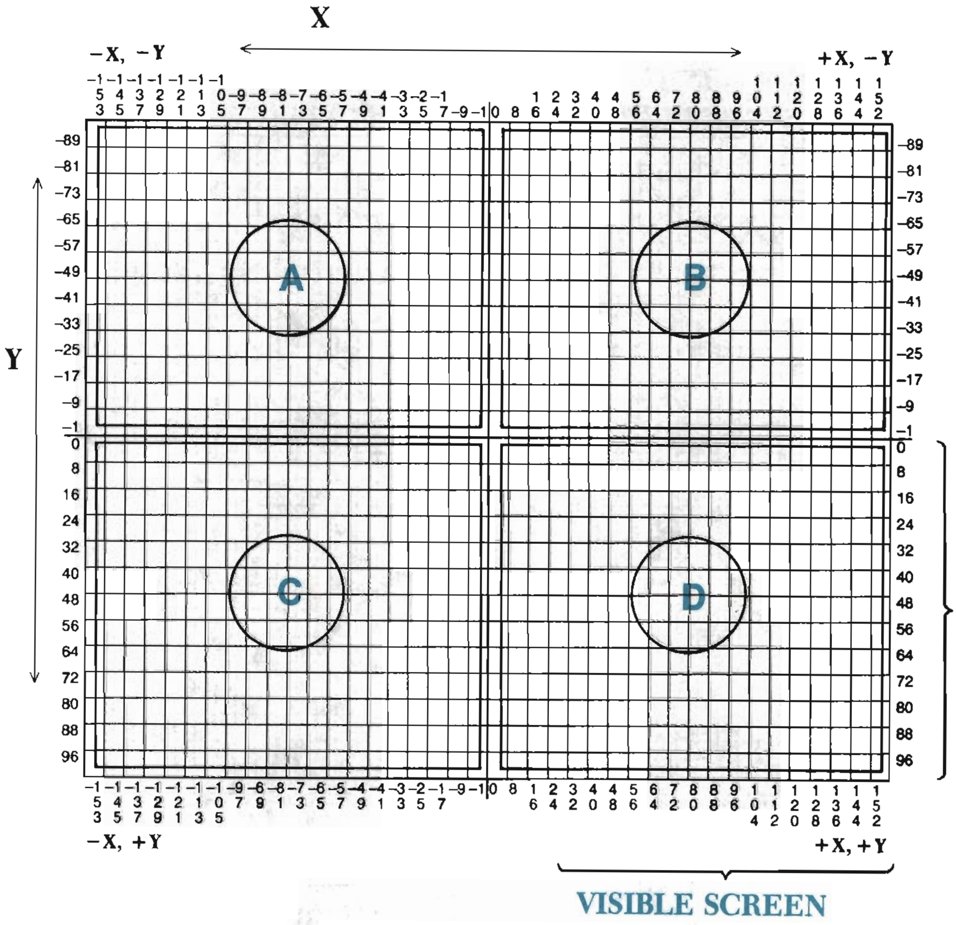
You see that the X axis falls along the top row of screen pixels. The Y axis falls down the left column of screen pixels. The anchor point (0,0) is then situated in the top left corner of your screen. The rest of the coordinates, including those that fall outside of the visible screen area, fall into their respective locations.

You may question whether the Commodore really has points at -150,0 or 0,-199. Actually, it doesn't. However, in order to keep a shape within certain boundaries as that shape is scaled, it will be necessary to use the "off-screen" pixel locations anyway. As long as you never attempt to plot off the screen (which the CLIP A SHAPE tool prevents you from doing), points outside of the visible screen can be "simulated" and used whenever necessary.

To clarify this, we use the grid below. Take a good look at it, for this grid will become something of a constant fixture for the next few chapters. The shaded areas represent pixel points that fall outside the visible screen. The unshaded area represents the top left corner of your screen (*not* the entire screen area.) Across both the top and bottom of the grid you will find the X location of every 8th pixel column. Down the left and right edges, you are given the Y location of every 8th pixel row.

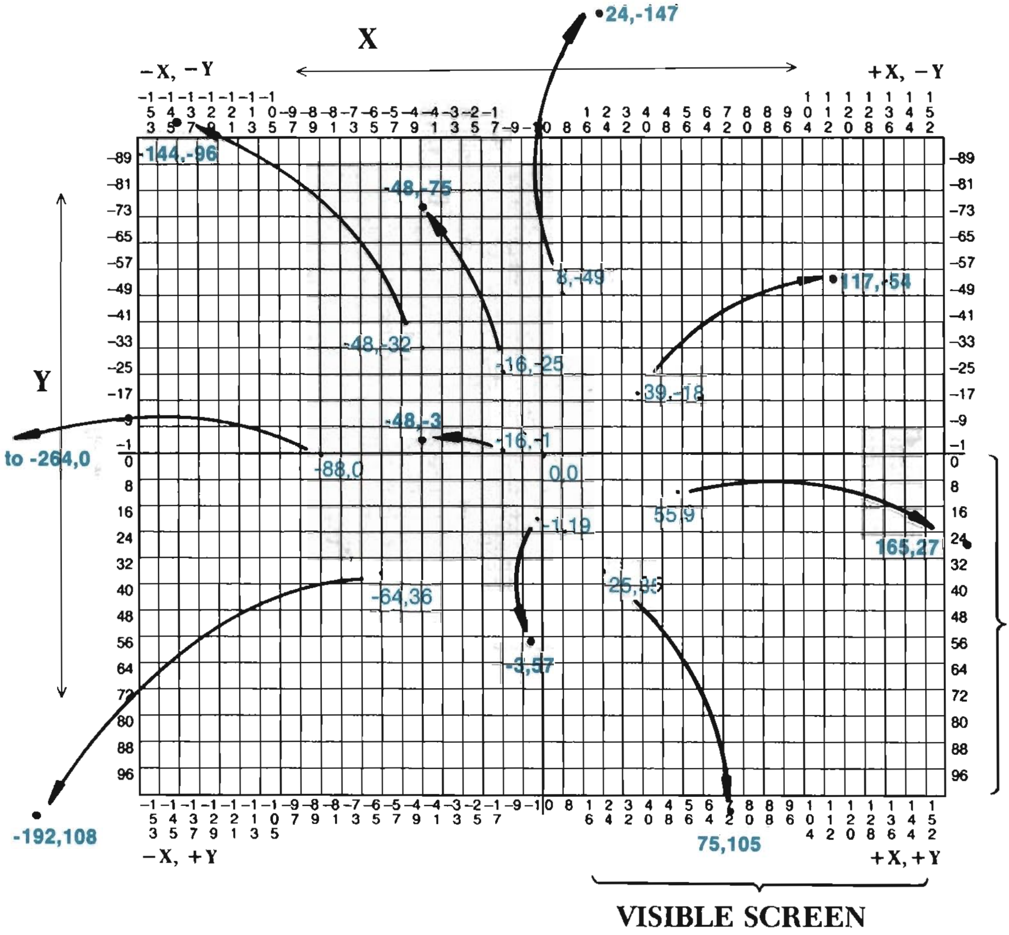


Why is this expanded coordinate system necessary? The answer can be found in the properties of positive and negative numbers. One such property is that a number remains the same sign (+/-) if you multiply it by a positive number. This means that as long as you use positive scale factors, a negative coordinate will remain negative, and a positive coordinate will remain positive. Look at the grid below. On the grid we have outlined the four distinct X,Y sections: (A) -X,-Y; (B) +X,-Y; (C) -X,+Y; and (D) +X,+Y.

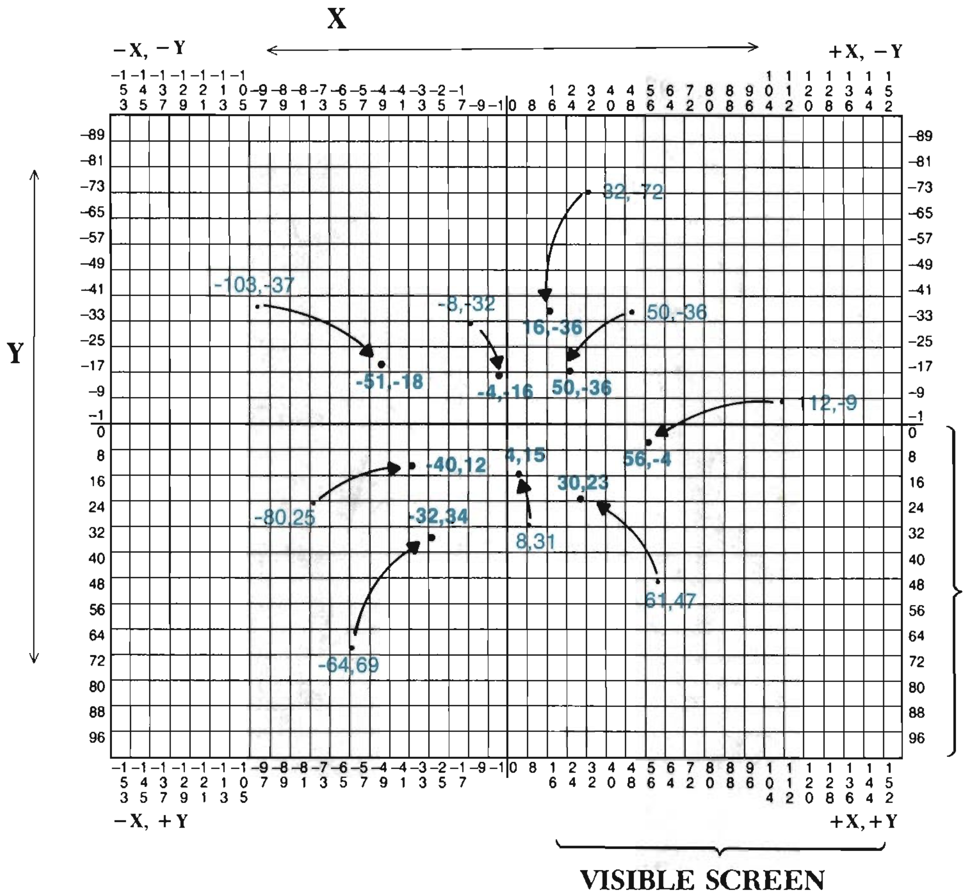


An endpoint that lies in section (A) above will *always* remain in that section if scaled by a positive factor. An endpoint found in section (B) will remain in section (B) if scaled by a positive scale factor. This idea is also true of sections (C) and (D). Endpoints *cannot* be moved from one section to another with a positive scale factor.

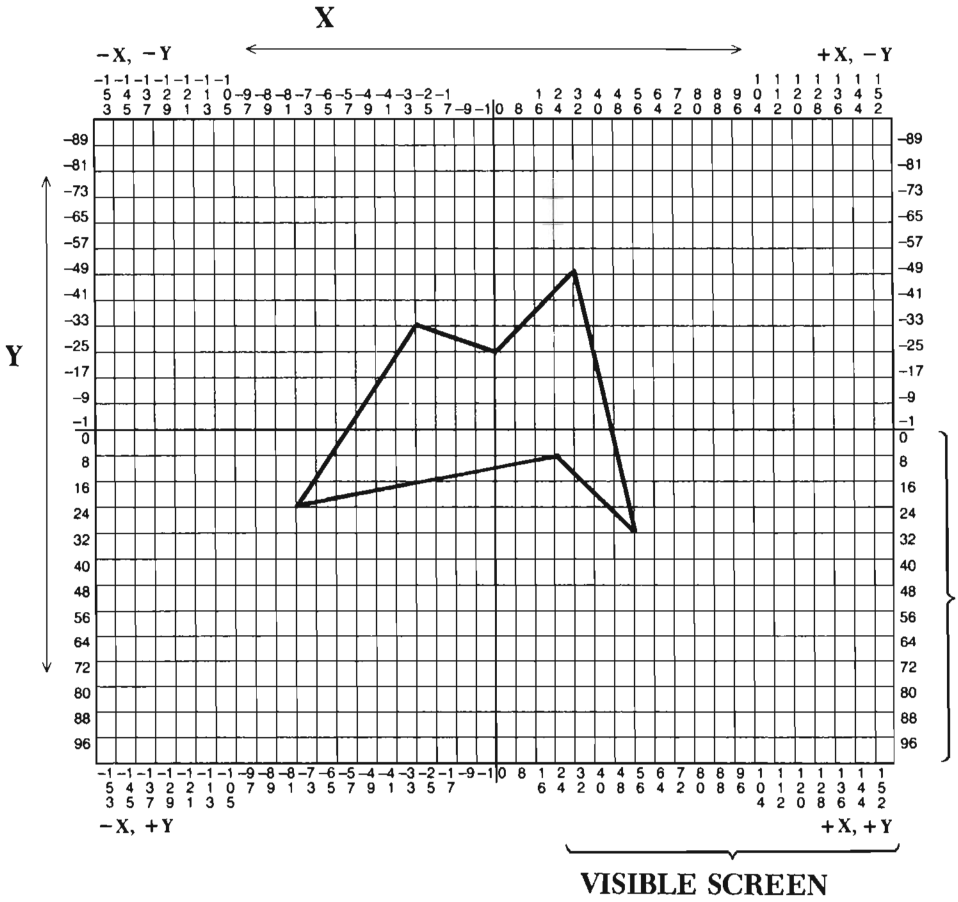
Now look at the next grid. On it, several endpoints have been plotted in each of the four available sections. The arrows from each endpoint locate where the endpoint would move were it to be scaled with scale factors of XS=3 and YS=3. Notice that each endpoint moves away from the anchor point at 0,0, but never moves from one section to the next.


24,-147
X
−X, −Y
+X, −Y
Y
-144,-96
-48,-75
8,-49
117,-54
-48,-32
-16,-25
39,-18
-48,-3
-16,-1
to -264,0
-88,0
0,0
55,9
-1,19
165,27
25,35
-64,36
-3,57
-192,108
75,105
−X, +Y
+X, +Y
VISIBLE SCREEN
-89
-81
-73
-65
-57
-49
-41
-33
-25
-17
-9
-1
0
8
16
24
32
40
48
56
64
72
80
88
96

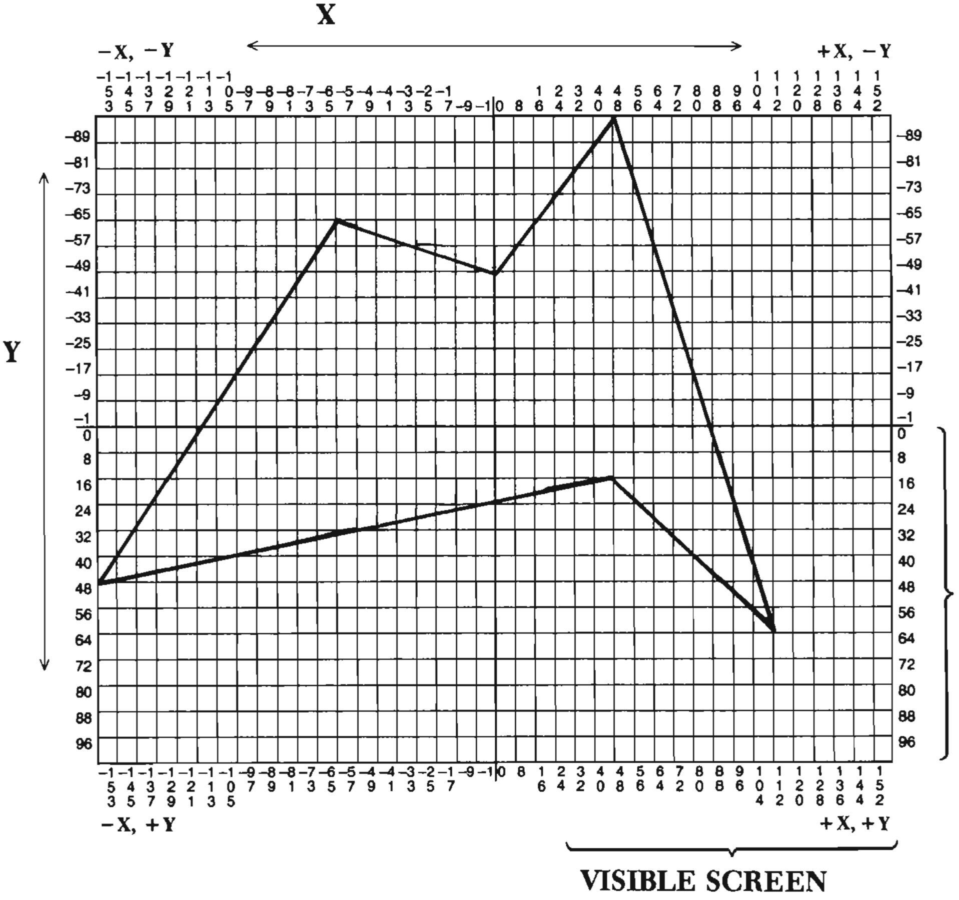
Scaling with fractions (for example, XS=.5: YS=.5) produces a similiar result:



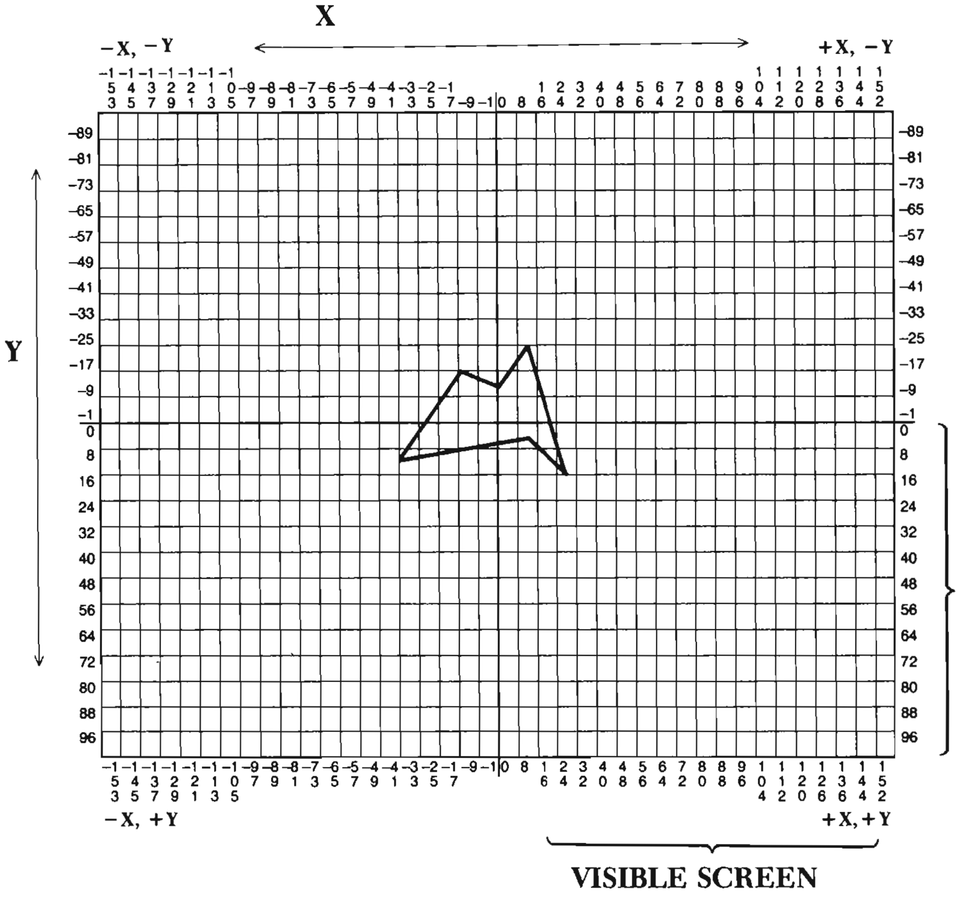
All of the "motion" centers around point 0,0. The endpoints remain within their sections, moving closer to or farther away from this anchor point. Since endpoints (or any points) remain within their respective sections, a shape defined around 0,0 will remain within its general location after scaling. To see how this works, look at the shape defined around 0,0 below:



Scaling this to twice its height and width will move the endpoints away from point 0,0:

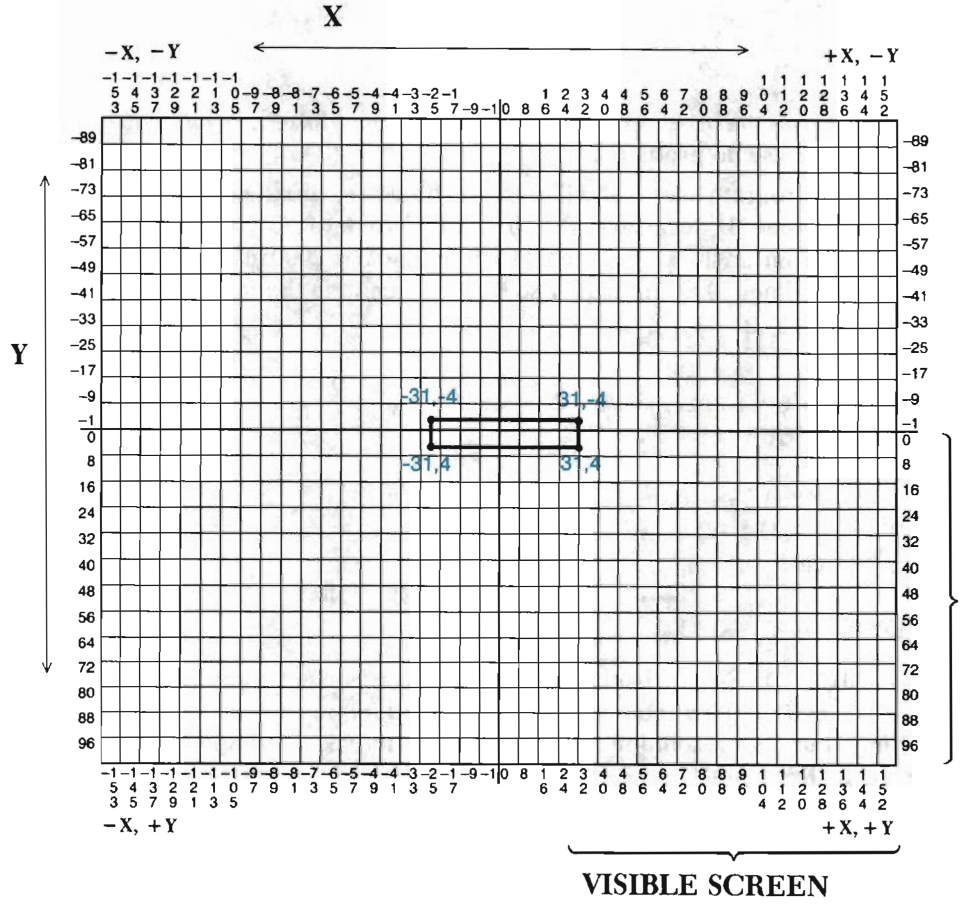


If, instead, the shape were scaled to half the height and width, the endpoints would be drawn closer to point 0,0:



Any shape falling into all four sections of the grid will be anchored. Scaling it up or down in either direction will not move it away from this location. The new shape may be taller, shorter, thinner, and/or wider, but it will still be situated around 0,0. The best possible anchor position is to locate a shape's *center-most point* at 0,0. This allows scaling to occur equally in all directions. If a shape's center-most point is at 0,0 before scaling, the scaled shape's center-most point will also be at 0,0.

Once the shape has been scaled to the desired size, it can be moved into view with your TRANSLATE A SHAPE tool. Let's try this out on the following shape:



This shape is sixty-three pixels wide, by nine pixels high. Because is has an odd number of pixels in both height and width, it has a single center point that can be (and is) positioned at point 0,0.

Notice that this shape does not fall into color blocks exactly the same from section to section. For instance, the point at -31,4 is closer to the center of its color block than the point at 31,-4. Shouldn't they fall into opposite but equal positions?

Opposite values fall differently into color blocks because the X and Y axes occupy space in the color blocks. Thus, the grouping of negative coordinates in a color block (e.g., -9 through -1) will always be one higher in absolute value than the grouping of positive coordinates in a color block (e.g., 0 through 8).

Now, back to our shape. Defining a shape that uses off-screen coordinates is done just as you might expect. The coordinates are entered into the program as DATA statements. If you have added any shapes to your program between line numbers 1014 and 1020, delete them now. Then, add the following:

```
1014 DATA "SCALE SHAPE",3,3
1016 DATA -31,-4,31,-4,31, 4
1018 DATA -31, 4
1020 DATA   0, 1, 1, 2, 2, 3, 3, 0
```

Again, as long as the off-screen coordinates are never plotted, using them to define a shape will cause no problems.

Below are program lines that will retrieve, translate, and then draw this shape on the visible screen. If necessary, run the ZAP routine to rid your program of any extraneous program lines. (Be sure that C is set equal to 2000 in line 172 first.) Then, enter/modify lines 2000 and higher so that they contain the following:

```
2000 GOSUB 10: C=7: GOSUB 30
2010 POKE 53280,C
2020 SE$="SCALE SHAPE": GOSUB 800
2030 GOSUB 110
2040 C=6
2050 XT=159: YT=99: GOSUB 140
2060 GOSUB 90
6000 GET A$
6010 IF A$ = "←" THEN GOSUB 20: END
6020 GOTO 6000
```

Line 2010 POKEs the background color (C's current value) into a special memory location. This memory location, 53280, controls the border color surrounding your high resolution screen. Line 2050 moves the shape to the center of the screen. Knowing that the center of this shape is originally defined at 0,0, translating it 159 columns to the right and 99 rows down will approximately center it on your screen (your screen has no single center point).

RUN the program.

The program should draw this rectangle, in blue, at the center of your screen. In addition, the border color will match the background color of yellow.

> *Note:* At this stage, there are many problems possible with your program due to typing errors. The more complex the subroutines, the more you need to be wary. For this reason, we have devoted the next section in this chapter entirely to helping you use this new tool. As part of this effort, we discuss locating those hard-to-find program bugs. If you have problems scaling, now or later, turn to the section called *Scaling Tips and Problems* for help.

Press ← to return to your program. We are going to add a loop to the program that does several things. First it will translate the current shape to the center of the screen, drawing it there. Next, it will retrieve the original shape and scale it. Each time through the loop, the XS and YS variables will increase by a factor of 1.2. This loop will process ten times, producing ten versions of the shape.
times, drawing eighteen versions of the scaled shape.

Add these lines to your program:

```
2Ø45 FOR L = 1 TO 1Ø
2Ø7Ø GOSUB 11Ø
2Ø8Ø XS=XS*1.2: YS=YS*1.2: GOSUB 15Ø
2Ø9Ø NEXT L
```

Notice that each time the loop is processed, line 2080 will increase the scale factors by one-fifth of their current value. Since the subroutine automatically sets XS and YS to one instead of zero, the first time through the loop the object remains simply an unscaled shape. At the bottom of the loop, the scale factors are increased, and the loop is repeated. RUN the program.

Gradually, your picture will build into a tunnel of rectangles extending inward. Or is it two stairwells, each one stepping deeper and deeper into the heart of the computer? No, it's a pyramid-like structure drawn as if viewed from above.

Actually, this illusion-creating design can be viewed in any of these ways. There is no true hint as to how the artist wants it to be viewed. (Although its probably safe to say that it was meant to be viewed in all three ways.) Creating illusions of depth is easy with your SCALE A SHAPE tool. Ordinarily, objects appear progressively smaller as they move off into the distance. Scaling a shape over and over again, increasing the scale factors each time, suggests this change in distance.

Return to your program and list lines 2000-2090. The most important thing to learn here is the necessity of anchoring your shapes. This shape was scaled ten different times. If it had not been anchored, you would have needed a hand-calculator to determine the location of each scaled shape. You would then have had to translate each scaled shape the appropriate distance in X and in Y to place it at the center of the screen. Here, we cleared out the C matrix each time. This meant that each scaling affected the original shape—which was an anchored shape.

This brings us to one final point. If you have scaled and translated a shape, and you want to work with its current form, don't clear the C matrix to re-anchor it. Instead, *translate* the shape back over point 0,0. This usually can be done with XT=-XT: YT=-YT: GOSUB 140. This merely reverses the translation you last performed—usually the translation that moved the shape away from 0,0 in the first place.

From now on, any shape you create should be defined around point 0,0. Even if you don't intend to scale it. There is always the possibility that you will rotate it, or even scale it for use in another design.

Before continuing, take the time to save this chapter's program. Save it under the filename "CHAPTER 5".

At this point, you have several options. You should begin by practicing with the program you have. Change line 2050 so that the shape is translated to different locations. Also, change line 2080, increasing (or decreasing) the scale factors at different rates. Now is the time to make mistakes and learn from them.

The next section deals with the tricks and problems of scaling, as well as common program bugs. If you are having problems with your program, or merely have an interest in the subject matter, read through this section. A section on design ideas specifically for scaling follows. If you have an itch to put your new tool to work, skip directly to the designs. When you're ready to be put to the test, flip to the Summary. In it you will find an exercise that will challenge your knowledge of the SCALE A SHAPE tool.

TOOL 150 :::::: SCALE A SHAPE

```
150 REM::::::SCALE A SHAPE
151 GOSUB 12Ø
152 XS=XS-(XS=Ø): YS=YS-(YS=Ø)
153 T(Ø,Ø)=XS: T(1,1)=YS
154 GOTO 13Ø
```

What It Does: This tool can "expand" or "shrink" your shapes along the X and/or Y axis.

Example Use: Two variables must be set before calling this tool with a GOSUB 150 statement. The variable XS (X-Scale) should be set to the desired X scale factor. The variable YS (Y-Scale) should be set to the desired Y scale factor.

Scale factors increase/decrease the number of units in a shape's height or width. (The distance from one pixel point to the next is equivalent to a *unit.*) A scale factor greater than one (>1) will increase the number of units in a shape's height/width. A scale factor of less than one (<1) will decrease the number of units in a shape's height/width.

This tool will scale the current form of your shape if the C matrix is not cleared first with a GOSUB 110 statement.

An example program line for scaling a shape's height by two and width by one-half is:

```
2Ø4Ø XS=.5: YS=2: GOSUB 15Ø
```

Technical Description: SCALE A SHAPE is similar to TRANSLATE A SHAPE, except that the T matrix takes the following form:

T Matrix

	0	1	2
0	XS	0	0
1	0	YS	0
2	0	0	1

Scaling Tips and Problems

This section covers two separate but indirectly related topics:

(1) Common Program Problems and Possible Answers; and
(2) Using Negative Scale Factors

The thread that ties these topics together is their non-crucial but nonetheless helpful nature. The absence of either one of these topics does not prevent you from furthering your expertise in scaling or in graphics. However, each has a value of its own that was deemed worthy of some discussion.

Topic 1 is self-explanatory: program bugs. Because this topic is so extensive, we will address only the more common mistakes. We assume that you've entered the subroutine tools correctly, and that the problem lies in the main routine. Make sure our assumption is correct before searching for an answer here.

Topic 2 covers the interesting possibilities presented by *negative* scale factors. This discussion is not comprehensive, but serves only to point you in directions not developed earlier in the chapter.

Common Program Problems and Possible Answers

There are three things you need to do whenever your program fails. First, you need to get back to text mode *without* erasing the text screen. Don't attempt this with the RUN/STOP and RESTORE keys (RESTORE will erase the text screen). Instead, press ←.

If the ← key does not return you to text mode, very carefully type: GOSUB 20 and press **RETURN**. You won't be able to see the letters you type, so type carefully. This should take you back to your program listing (try this two or three times before giving up and using RUN/STOP and RESTORE).

> REMEMBER: If your program set the screen's background color to light blue, the text screen may be a solid, light blue color. It will appear as if you are still in high resolution graphics, when you are actually viewing your program. To remedy this problem, change the cursor color to white and then type: POKE 53281,6 **RETURN**.

The second step is to convert your program listing to lower-case. You can easily do this by pressing C= and **SHIFT** at the same time. Don't be too skeptical of the benefits of this move. Many typing errors that are hard to find in upper-case are easily spotted in a lower-case program listing.

The third step is to look for an error message. Error messages are extremely valuable because they give the line number where the computer experienced problems. This is not to say that the error will be in that line number. It could be that you are given an OUT OF DATA ERROR IN 804, when the problem is in a DATA statement in line 1020. The line number is a clue to the problem, though, and is the first place you should check.

As a quick checklist, make sure the C matrix has been cleared *prior* to retrieving a new shape in the program. Make sure that all necessary variables are set before each GOSUB statement. Often, variables will be holding values carried over from a previous shape. Thus, it is important to set all variables that a subroutine uses before calling it. Finally, remember that a minimum of five steps are necessary to draw and paint any shape:

(1) The shape must be defined in data statements.
(2) The shape must be retrieved (SE$="?": GOSUB 800).
(3) C must be set to a drawing color (C=?).
(4) The DRAW A SHAPE tool must be called (GOSUB 90).
(5) The PAINT A SHAPE tool must be called (GOSUB 60).

If the shape is defined around point 0,0, it must also be translated to a more visible area.

Several error messages are listed below. Possible reasons for the message are given beneath each one. This list is only intended as a guide, and does not consider every possible problem or solution. Other program problems are discussed immediately following these error messages.

BAD SUBSCRIPT ERROR

.Check to see if you have a DIM statement in line 1006 to create arrays P%, R%, L%, T, C, and W.

.If you are working with a complex shape of over 100 endpoints and/or lines, you need to dimension P%, R% and/or L% to a larger size.

.Check your Shape Library. Make sure you have correctly given each shape's count of points and lines. Then, make sure those points and lines have all been entered.

DO YOU KNOW WHAT YOU ARE DOING?

.Somewhere in your program you have a GOSUB 170 statement or a GOTO 170 statement. This sends the computer to the ZAP routine. If you continue, you will erase your main routine. Check each GOSUB and GOTO statement.

FILE NOT FOUND ERROR IN 0

.Your machine language was not in memory, so program line 0 tried to load file "M/L". The "M/L" file was not found on your disk/tape, producing this message. Copy your "M/L" file onto this disk/tape, or copy this program onto a disk/tape that already contains the "M/L" file.

ILLEGAL VALUE or ILLEGAL QUANTITY ERROR

.Somewhere you have used or are trying to store a value larger than the computer can hold. First, check the variables in your program. Type a question mark (?), followed by the variable you want to check (?X), and press **RETURN**. The computer will display the current value it is storing for that variable. Check every variable. For floating point variables (e.g, X, Y, X1, PP, etc.), the computer can store very large numbers. Loops, however, can increase the value of a variable at a surprising rate. If the computer displays a variable's value in scientific notation (e.g., 3.7698E+23), you are probably looking at the source of the problem.

.Integer variables (e.g., R%, L%, P%) can only hold whole numbers between -32768 and +32767. Your endpoints could be scaled or translated to such an extent that they surpass these limits. To check the contents of an array, like R%, type: ?R%(*,0),R%(*,1). Replace the asterisks (*) with the endpoint number you want to check. To check endpoint number 5, type: ?R%(5,0),R%(5,1). The computer will display two numbers. The first number will be the X coordinate of the endpoint. The second one will be the Y coordinate. If you are checking L%, the first number will be the "from" endpoint number, and the second will be the "to" endpoint number.

OUT OF DATA ERROR

.Check to see if you've made a typing error in the name SE$ is set to.

.Count the number of points and lines in the shape you are retrieving, as well as those before it. Make sure the number of points and lines in a shape matches the count of points and lines given in the shape's very first DATA statement.

SYNTAX ERROR

.This is a generic error code that identifies any error for which a "customized" error code is not available. Often, the error will be in the line number given in the error message. With this error code, you can only start looking.

Error messages can indicate a typing error, or an error in logic. Check for typing errors first:

.Have you typed the variables correctly? It is very easy to type X2 instead of XS, Y1 instead of Y2, XT instead of YT, etc. Look closely. Are they all correct?

.Have lower-case L's (l) been typed in place of ones (1) or capital o's (O) typed in place of zeroes (0). This is a very common typing mistake.

As a final checklist, glance through the titles of symptoms below. Hopefully, you'll find your answers there.

SHAPE NEVER GETS PLOTTED ON THE SCREEN

.Did you retrieve the shape?

.Did you set the color variable (C) to a drawing color?

.Did you include a GOSUB 90?

.Is the shape somewhere off of the screen, needing to be translated to a visible area? (Look at the contents of R% to find out.)

.Was the shape unanchored and then scaled off of the screen? (Again, look at the contents of R%.)

.Is one tiny pixel plotted, but nothing else? (You probably forgot a GOSUB 110 at the beginning of the program.)

SHAPE IS NOT BEING PAINTED PROPERLY

.Has the color variable (C) been set to a color other than that used for the background?

.If you are *not* using a predefined paint point, did you insert GOSUB 62 as needed (not GOSUB 60)? Did you set X and Y to a point that is definitely inside the shape?

.Are you using a predefined paint point? If so, did you insert GOSUB 60 in the main routine? Have you set PP to the correct point number from your data list?

.Does the paint point fall *inside* your shape, or on it? (Make sure it falls inside the shape.)

.Is the shape being painted in polka dots? (The color variable is probably set to some number less than 0 or greater than 16. Change it to a correct color code.)

.Are you painting in multi-color? (See next topic.)

MULTI-COLOR IS NOT WORKING PROPERLY

.Are small portions of the screen showing incorrect colors? (Your program is probably not in an endless loop, which is necessary in multi-color.)

.Does the computer begin painting to the left of your shape instead of inside it, as planned? (In multi-color, a paint point can become part of the outline shape if it is too close to the left side of the shape. This is because every two pixel columns are treated as one. If your paint point is adjacent to the shape's left edge, try moving it one column to the right, and one row down.)

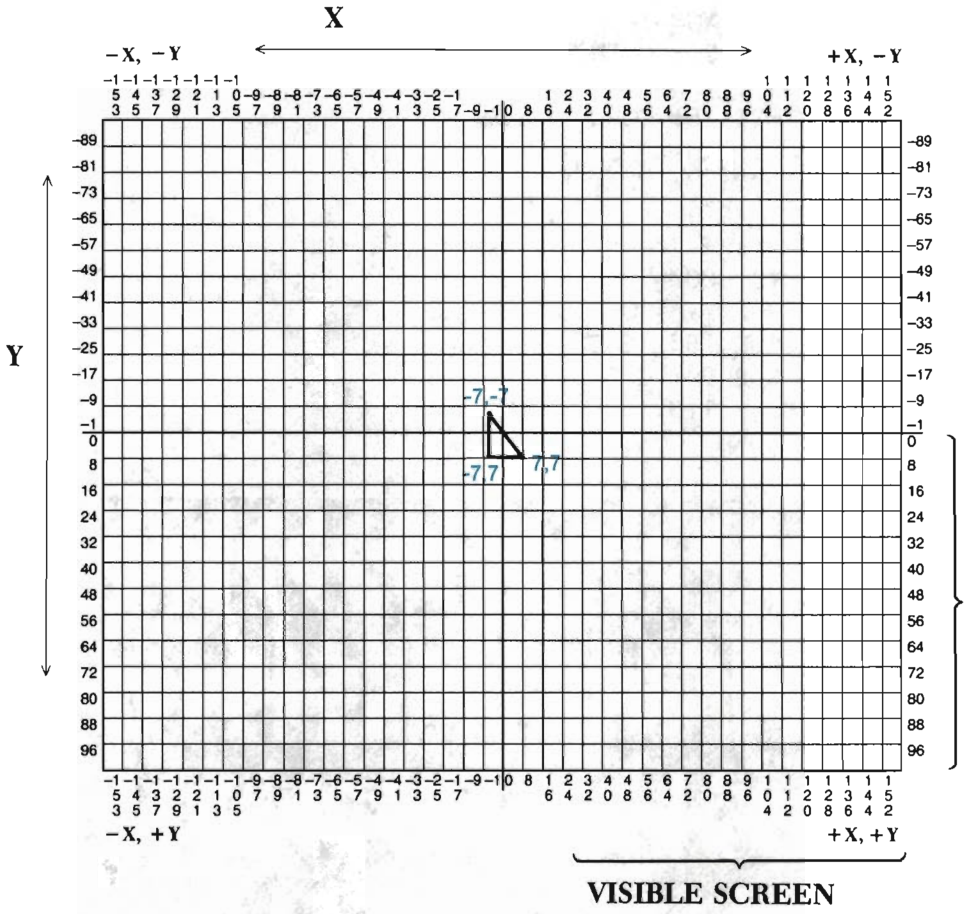
Using Negative Scale Factors

Using negative scale factors creates mirror images of your shapes. This is a valuable tool for many designs, and is not something normal scaling or rotation can do. A negative YS value will reverse the shape from top to bottom (plot it upside-down). A negative XS value will reverse the shape from right to left. Other than this, the shape will be scaled up or down to the same size the positive scale factor would have scaled it. If XS=-2 and YS=-.5, the shape will be flipped horizontally and vertically. At the same time, it will be scaled to twice the width and half the height.

To understand how this works, recall that if you multiply a number by a negative value, the product will have the opposite sign (+/-) of the number with which you started. From this you can see that an XS scale factor of -1 simply reverses the signs of every X coordinate in the shape. A YS scale factor of -1 reverses the signs of every Y coordinate in the shape. By reversing the signs, an equal but opposite shape is drawn on the other side of the axis.

Using values other than -1 (e.g., -2, -3, -6) will reverse the coordinate signs *and* increase/decrease their values. This will scale them appropriately.

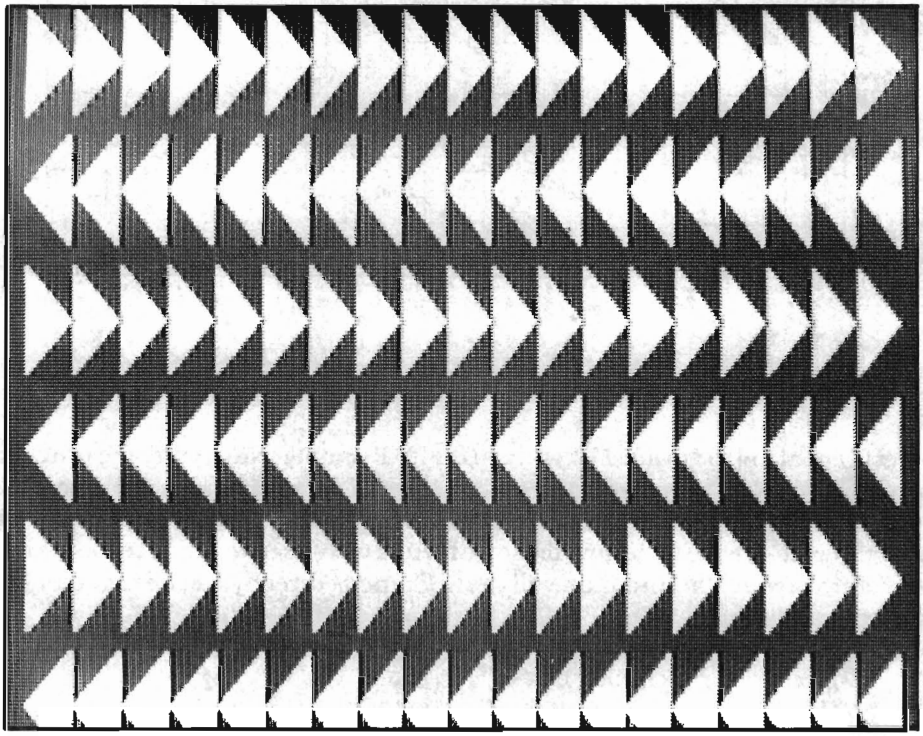
Through experimentation with negative scale factors, you should be able to implement them easily enough. To start you out, we are providing a simple practice program that works with the following shape:



Set C equal to 1008 in line 172, and run the ZAP routine. Next, enter the program below and RUN it. As the program runs, pay careful attention to how many versions of the triangle are plotted (there will be four). When you return to the program listing, glance over it so see how the scale factors were used to produce the design. (A picture of what this program will draw if typed correctly can be found below the program listing.)

```
1Ø16 DATA "MIRROR IMAGE",3,2
1Ø18 DATA -7,-7, 7, 7,-7, 7,-2, 2
1Ø2Ø DATA  Ø, 1, 1, 2, 2, Ø
2ØØØ GOSUB 1Ø: C=6: GOSUB 3Ø
2Ø1Ø SE$="MIRROR IMAGE": GOSUB 8ØØ
2Ø2Ø C=3: PP=3: GOSUB 11Ø
2Ø3Ø YT=1Ø: FOR L=1 TO 3
2Ø4Ø GOSUB 7ØØØ
2Ø5Ø YS=-1: XS=1
2Ø6Ø YT=YT+15: GOSUB 7ØØØ
```

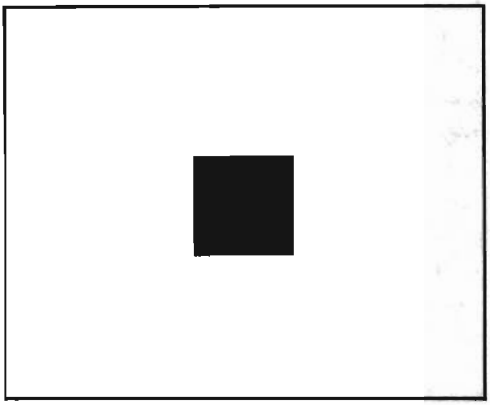
```
2Ø7Ø YS=1: XS=-1
2Ø8Ø YT=YT+2Ø: GOSUB 7ØØØ
2Ø9Ø YS=-1: XS=-1
3ØØØ YT=YT+15: GOSUB 7ØØØ
3Ø1Ø XS=1: YS=1
3Ø2Ø YT=YT+2Ø: NEXT L
6ØØØ GET A$
6Ø1Ø IF A$ = "<--" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
7ØØØ FOR XT=16 TO 319 STEP 16
7Ø1Ø GOSUB 15Ø: GOSUB 14Ø
7Ø2Ø GOSUB 9Ø: GOSUB 6Ø
7Ø3Ø GOSUB 11Ø: NEXT XT
7Ø4Ø RETURN
```



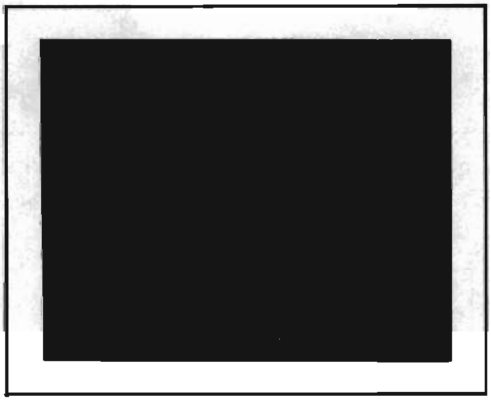
Design Ideas

The purpose of scaling is to alter a shape's height/width to create a better, different, or more realistic design. The sketches for this chapter are meant to help you realize the potential of scaling.

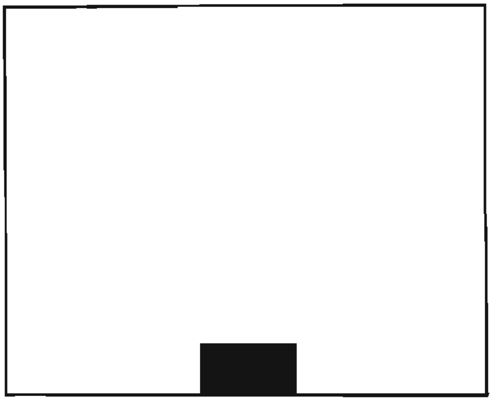
The shape in the first sketch can be considered "small" because of its size in proportion to its surroundings. It is surrounded by lots of space.



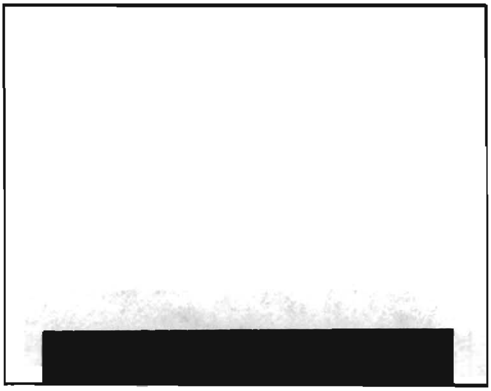
The square in the next sketch is regarded as "large" because it exists in a small amount of surrounding space. The relationship of an object to its surroundings will change as the scale (proportion) of that object changes.



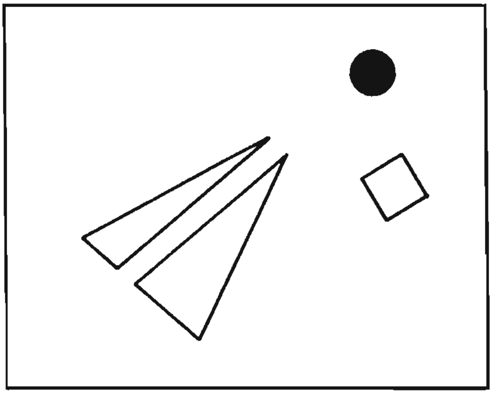
The small square has been translated below so that it resides partially out of the picture. The surrounding space completely dominates this picture. There is more background space than the space taken up by foreground objects.



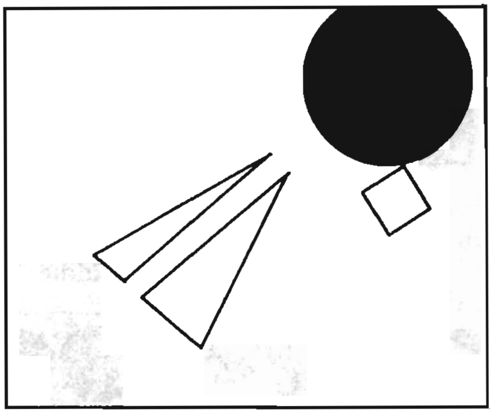
This same translated shape has been scaled to a larger size in the next sketch. The large square does not dominate the picture because it has been placed mostly outside of the picture. A large object can appear less dominant, depending on where is it positioned in the picture. All objects in a picture are compared visually to each other, as well as to the surrounding space and the border.



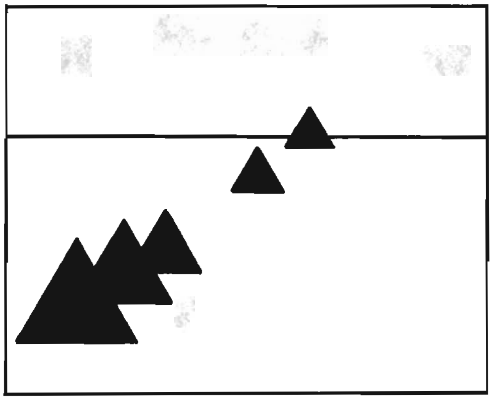
Scale can be thought of as an object's "weight," or its proportion in relation to the rest of the picture. The objects in the next picture suggest light weight and small size when compared to the surrounding field of space.



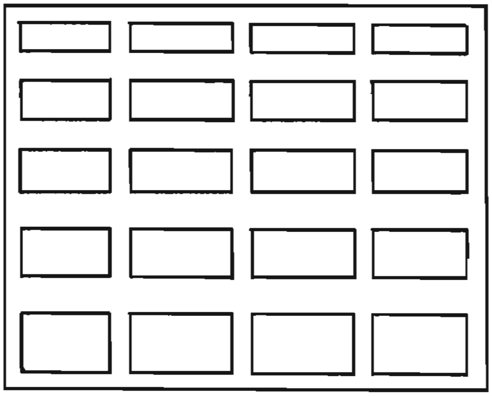
When the proportions of objects change, their "weights" also change. Look at the sketch below. It uses the same objects as in the previous sketch, but not the same proportions. Notice that each object appears to have a visual weight as compared to the other objects. For example, the black ball looks heavier than the small square. In the previous sketch, the black ball seemed to be small and light. In this sketch, the ball dominates the picture.



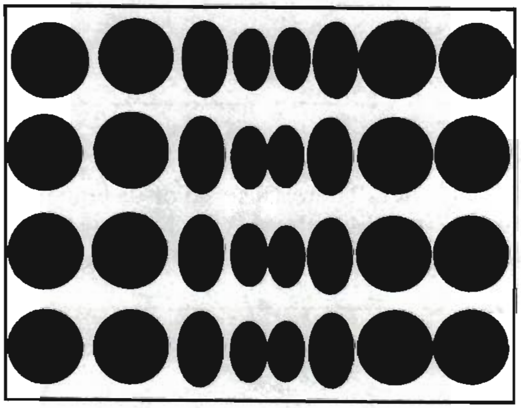
Our next sketch contains several scaled versions of the same triangle. Each object appears to be placed in space. There's a sense of distance and depth. A sense of depth can be achieved by reducing the size of objects that are to recede back into space. The horizon line intensifies the feeling of distance.



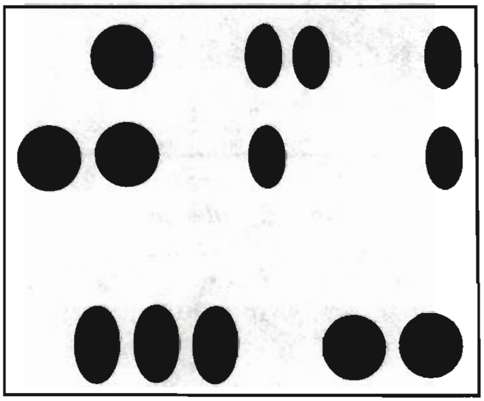
A rectangular shape has been scaled *and* translated in the design below. The scaling provides some variety in this repetitive composition. The rectangle is continually scaled vertically along its Y axis, while the X scale remains the same. The spacing between the translated shape also remains the same.



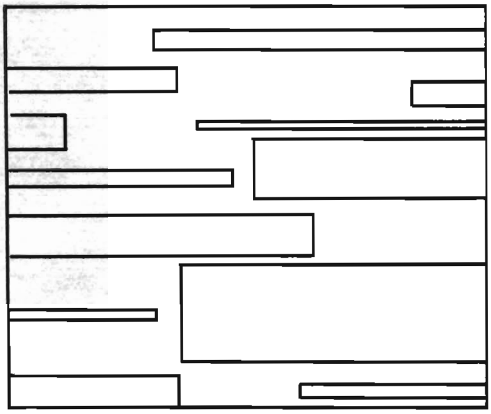
Although the size varies in the next sketch, spacing is consistent and regular. The arrangement of shapes is orderly and balanced. The shapes seem to recede back into space, gradually reducing in size toward the middle.



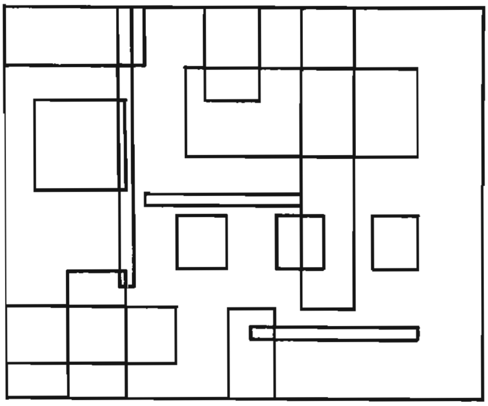
The next sketch, in contrast, is irregular in order and spacing. There's variety in it's use of space and the placement of each shape. The shapes appear to "float" in a flat space. A sense of depth is not evident in this picture.



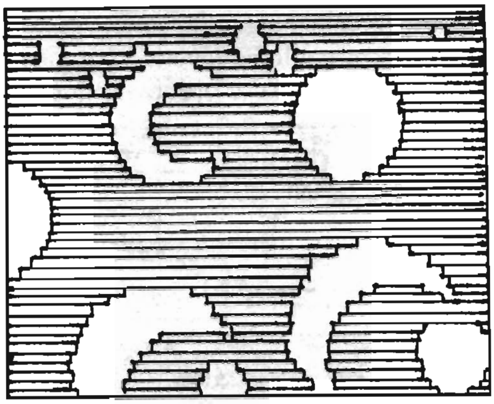
In the following sketch, there is once again variety in the size and placement of shapes. Order is maintained through the repetition of the same shape throughout the picture. A pattern develops from this orderly arrangement of shapes.



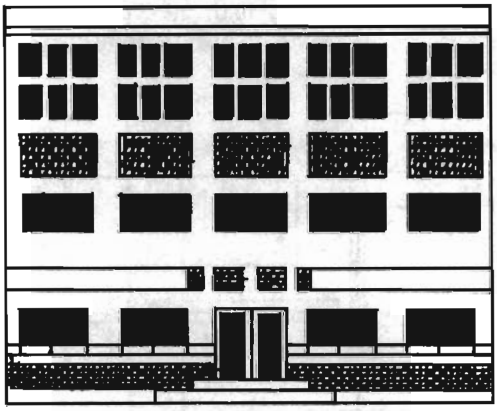
The shape sizes are varied a great deal in the next picture. Some shapes are scaled up vertically, whereas others are scaled up horizontally. This complex arrangement emphasizes an unexpected, irregular order.



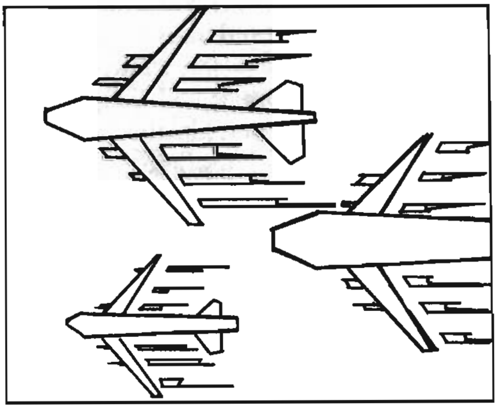
A rectangle has been carefully scaled and translated over and over again to create the interesting composition below. The clear, white spaces interspersed between the rectangles are perceived as the foreground shapes. Shapes such as these are called "negative shapes." Negative shapes, like the moon shapes in this design, are not physically defined with points and lines (like the rectangles).



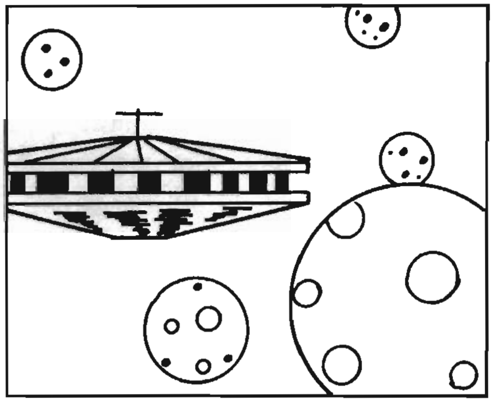
Below you see results of the calculated scaling and translation of a rectangle to create a building design. Variation appears both in the size of the shapes and their tones. The outline shapes contrast with the solid colored shapes. A pattern or logical order can be seen in the arrangement of the shapes.



The jets in the next sketch illustrate the scaling and translation of one object. The smaller jet at the bottom appears further away in space. The basic jet shape is a collection of many shapes put together.



The final sketch shows how new shapes can be made by combining other shapes. The planets and spacecraft were composed from smaller shapes (like circles and squares) that were carefully scaled and translated. Notice how the large foreground planet has been placed partially off of the screen. A sense of depth is created by the varying sizes of each planet.



Summary

You are at the close of yet another chapter, and have added a SCALE A SHAPE tool to your graphics tool kit. At the beginning of this chapter we stated, "Scaling is one of the harder graphics concepts to learn." Looking back on your experience thus far with scaling, this opening statement may sound a bit exaggerated to you. Or, perhaps you feel it wasn't a firm enough warning. Either way, you have come through it and are ready to move on. Before you do, however, you will of course want to prove your skill by working through this chapter's exercise. As you do, keep in mind the following:

—Scaling in the X direction (XS) alters the width of a shape.
—Scaling in the Y direction (YS) alters the height of a shape.

—A shape's height = # Pixels in Height -1.
—A shape's width = # Pixels in Width -1.

—A scale factor greater than one (>1) increases the height/width of the shape.
—A scale factor less than one (<1) decreases the height/width of the shape.

—A negative XS scale factor flips the shape horizontally.
—A negative YS scale factor flips the shape vertically.

—To scale the original shape, the C matrix must be cleared.
—To scale the *current* form of the shape, the C matrix must be left alone.

—To anchor a shape, it must be defined around point 0,0.

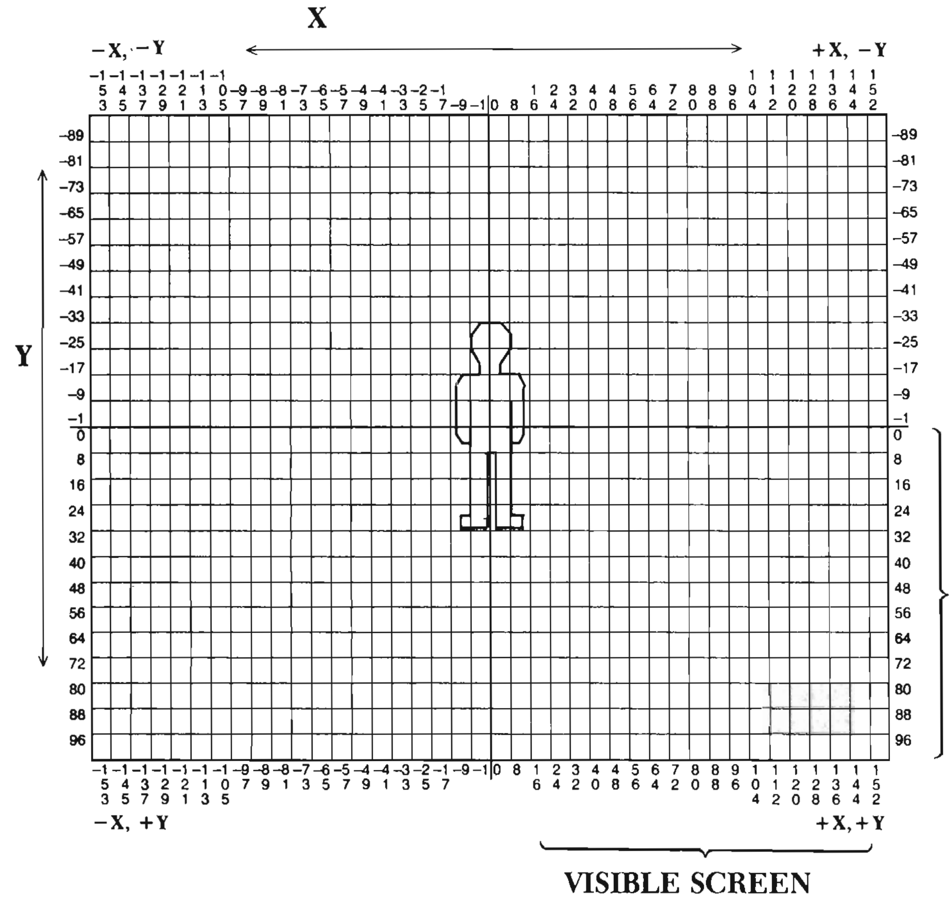
—To re-anchor a translated shape, either the C matrix must be cleared out, or the shape should be translated back over 0,0.

—The mathematics of scaling will transform each endpoint as follows:

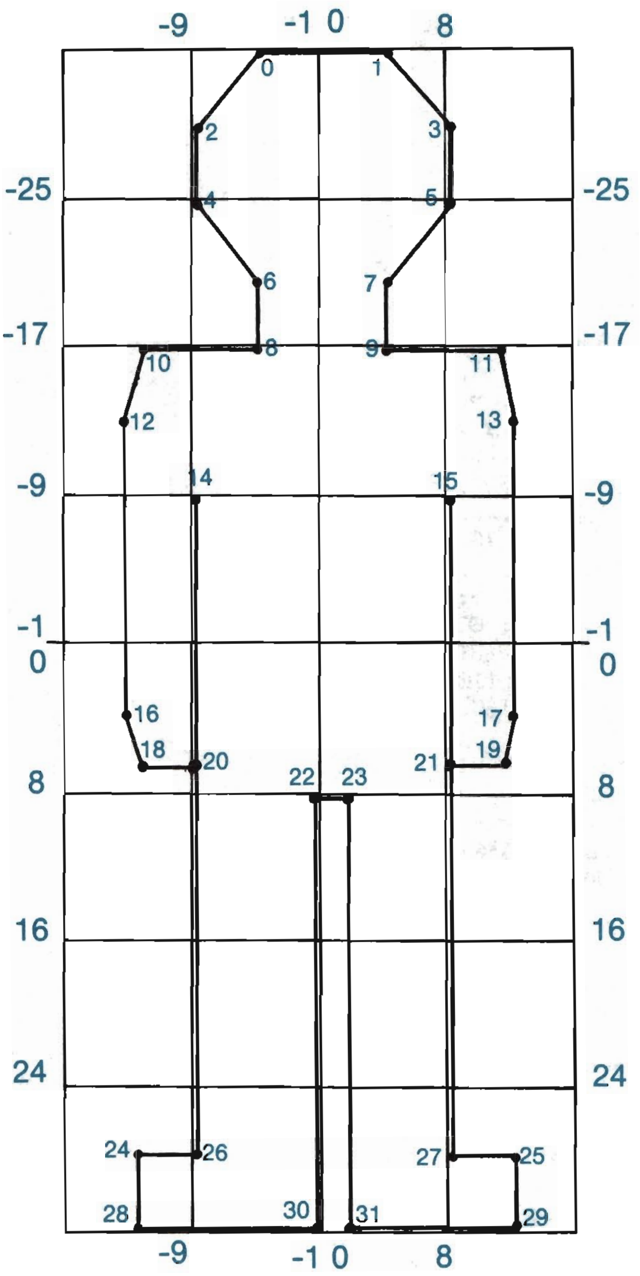
Old X * XS = New X
Old Y * YS = New Y
NEW ENDPOINT = New X,New Y

Exercise

Enter DATA statements that define the shape sketch on our grid below.



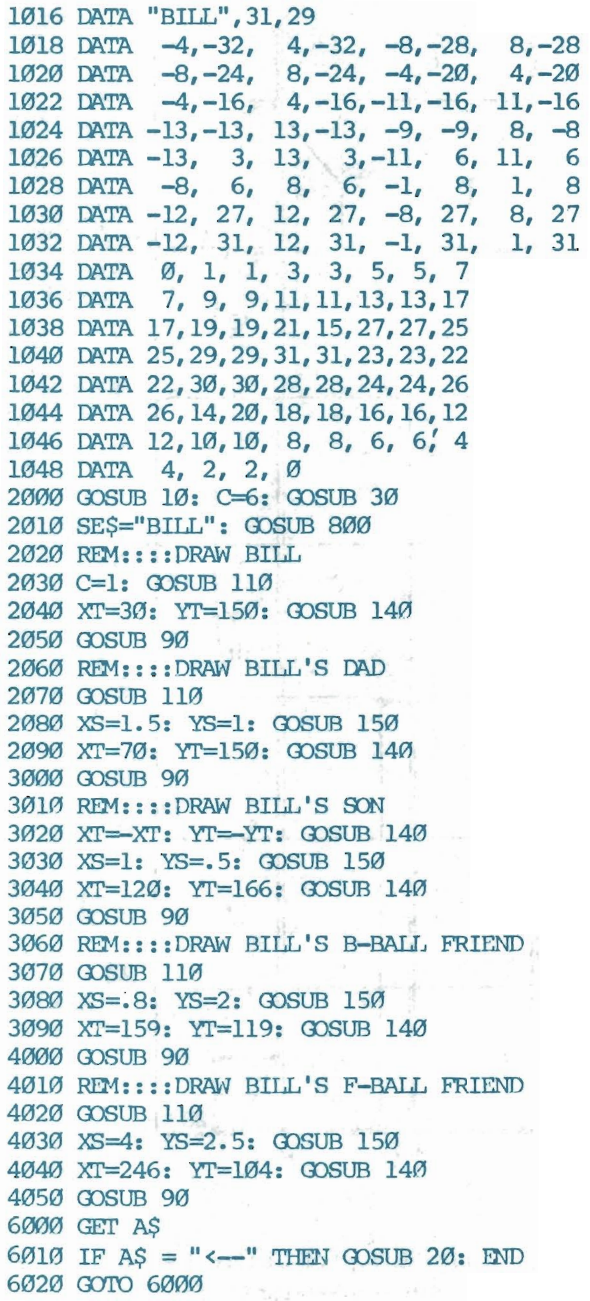
To help you in your endeavor, we have enlarged this figure and numbered the endpoints (see below). Notice that the endpoint numbers jump from one side of the Y axis to the other. This makes determining the coordinates easier. Endpoints 0 and 1 have opposite X locations (-4 and 4), and the same Y locations (-32). Endpoints 2 and 3 have opposite X locations (-8 and 8), and the same Y locations (-28). This is true for each set of even and odd endpoints.



After you've defined this shape, scale it to several different widths and heights, drawing it as you go. An example program is given for the solution.

(Note: Translation and Scaling take time. It will take several seconds before each shape is drawn. Please be patient.)

Solution



```
1Ø16 DATA "BILL",31,29
1Ø18 DATA  -4,-32,  4,-32, -8,-28,  8,-28
1Ø2Ø DATA  -8,-24,  8,-24, -4,-2Ø,  4,-2Ø
1Ø22 DATA  -4,-16,  4,-16,-11,-16, 11,-16
1Ø24 DATA -13,-13, 13,-13, -9, -9,  8, -8
1Ø26 DATA -13,  3, 13,  3,-11,  6, 11,  6
1Ø28 DATA  -8,  6,  8,  6, -1,  8,  1,  8
1Ø3Ø DATA -12, 27, 12, 27, -8, 27,  8, 27
1Ø32 DATA -12, 31, 12, 31, -1, 31,  1, 31
1Ø34 DATA  Ø, 1, 1, 3, 3, 5, 5, 7
1Ø36 DATA  7, 9, 9,11,11,13,13,17
1Ø38 DATA 17,19,19,21,15,27,27,25
1Ø4Ø DATA 25,29,29,31,31,23,23,22
1Ø42 DATA 22,3Ø,3Ø,28,28,24,24,26
1Ø44 DATA 26,14,2Ø,18,18,16,16,12
1Ø46 DATA 12,1Ø,1Ø, 8, 8, 6, 6, 4
1Ø48 DATA  4, 2, 2, Ø
2ØØØ GOSUB 1Ø: C=6: GOSUB 3Ø
2Ø1Ø SE$="BILL": GOSUB 8ØØ
2Ø2Ø REM::::DRAW BILL
2Ø3Ø C=1: GOSUB 11Ø
2Ø4Ø XT=3Ø: YT=15Ø: GOSUB 14Ø
2Ø5Ø GOSUB 9Ø
2Ø6Ø REM::::DRAW BILL'S DAD
2Ø7Ø GOSUB 11Ø
2Ø8Ø XS=1.5: YS=1: GOSUB 15Ø
2Ø9Ø XT=7Ø: YT=15Ø: GOSUB 14Ø
3ØØØ GOSUB 9Ø
3Ø1Ø REM::::DRAW BILL'S SON
3Ø2Ø XT=-XT: YT=-YT: GOSUB 14Ø
3Ø3Ø XS=1: YS=.5: GOSUB 15Ø
3Ø4Ø XT=12Ø: YT=166: GOSUB 14Ø
3Ø5Ø GOSUB 9Ø
3Ø6Ø REM::::DRAW BILL'S B-BALL FRIEND
3Ø7Ø GOSUB 11Ø
3Ø8Ø XS=.8: YS=2: GOSUB 15Ø
3Ø9Ø XT=159: YT=119: GOSUB 14Ø
4ØØØ GOSUB 9Ø
4Ø1Ø REM::::DRAW BILL'S F-BALL FRIEND
4Ø2Ø GOSUB 11Ø
4Ø3Ø XS=4: YS=2.5: GOSUB 15Ø
4Ø4Ø XT=246: YT=1Ø4: GOSUB 14Ø
4Ø5Ø GOSUB 9Ø
6ØØØ GET A$
6Ø1Ø IF A$ = "<--" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
```

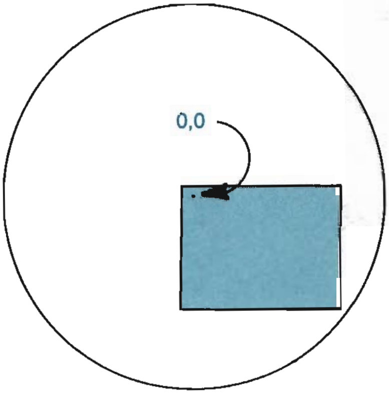
Chapter Six

ROTATING SHAPES

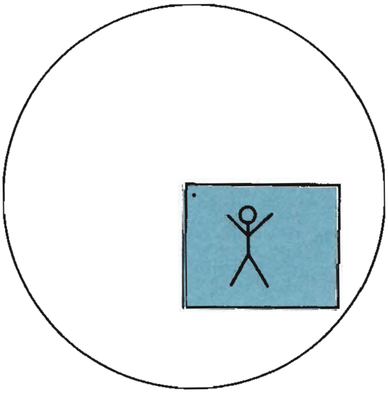
Rotation is the turning of an object on or around a central location. If you've ever watched a windmill turning in the breeze, you've witnessed rotation. If you've ever spun a child's top, you've put rotation into motion. If you've ever twirled on a swivel chair or ridden on a merry-go-round, then you have also rotated.

The examples above are examples of three-dimensional shapes rotating. You, however, are working with two-dimensional shapes. Two-dimensional rotation is difficult to illustrate with real life examples because there are no two-dimensional shapes to be found in real life. Since this is the case, we'll ask you to use your imagination as we work though an example of two-dimensional rotation.

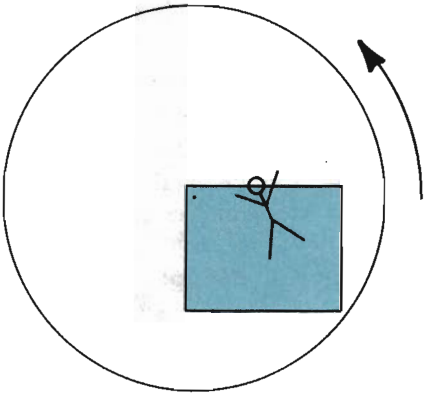
Imagine, if you will, that your screen has a large "drawing wheel" attached it to, like the one diagrammed below. Notice that this drawing wheel is approximately four times larger than the screen, and is attached by a small pin running through its center point. This pin sticks into point 0,0 on the screen.



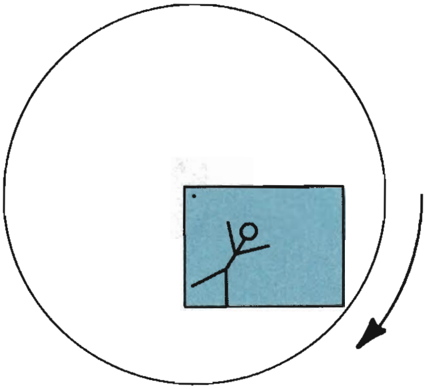
Let's assume that you draw the stick figure below on your drawing wheel. The stick figure is located near the center of the screen area, and appears to be standing on level ground.



This is where the fun starts. Watch what happens when you rotate the drawing wheel slightly to the left. The picture below shows the stick figure tilted at an angle, and he appears to be walking uphill.

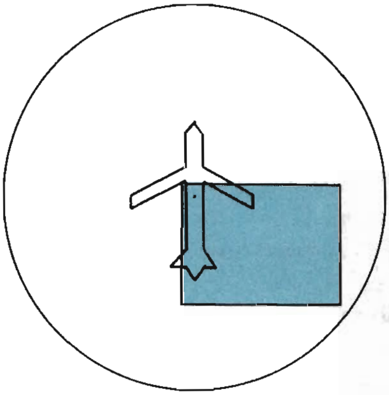


Just the opposite impression occurs when you rotate the wheel to the right. The figure then appears to be walking downhill:

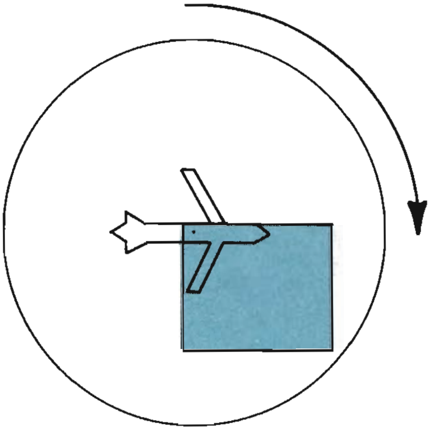


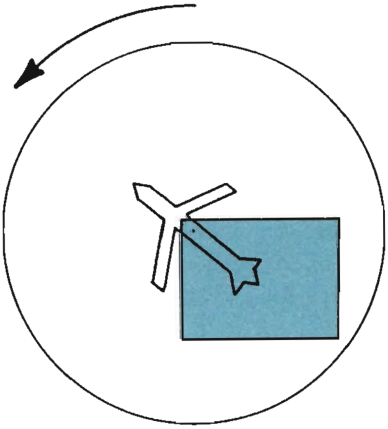
Notice that the shape itself never changes form, as might happen in three-dimensional rotation. Instead, it simply moves along a circular path to a new screen position. If you were to turn the wheel completely around, the stick figure would return to its original position.

Now let's assume that you decide to draw another shape. You start with a "clean" drawing wheel, again with its center pinned to point 0,0 on the screen. This time, you draw an airplane that is centered over 0,0, like the one shown here:



You find that rotating the drawing wheel this time merely turns the airplane in its place:

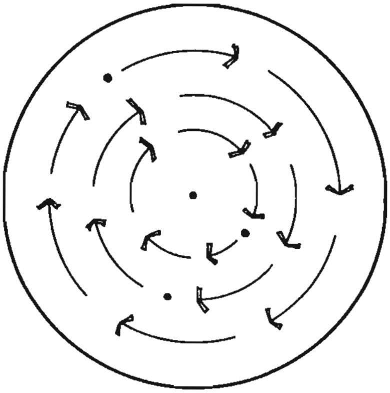




Although you won't be drawing your shapes on a drawing wheel, as here, the same general principles apply when you rotate shapes on your computer screen. Your shapes will rotate around point 0,0 in the same manner that we've seen in these "drawing wheel" examples.

Keep in mind that the computer retrieves a shape and then transforms its *endpoints*. When we discussed translation in Chapter 4, you saw that the shape's endpoints were translated to a new location, and then connected by the lines described in L%. Chapter 5 showed how a shape's endpoints were scaled to a new location, and then connected by the lines described in L%. As we study rotation, you will see that each endpoint is individually rotated around point 0,0, and then connected by the lines described in L%.

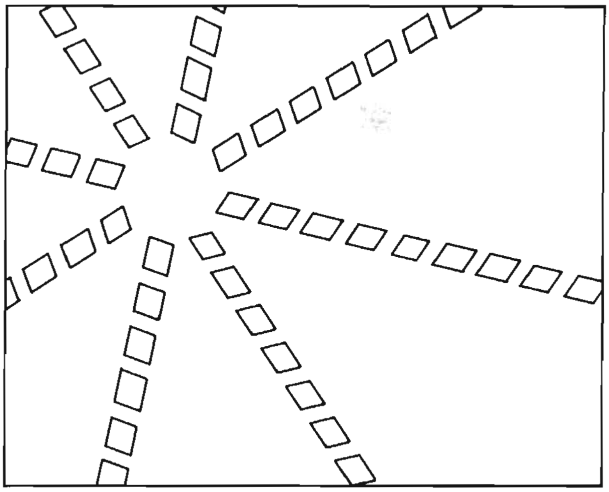
Let's look at that drawing wheel one more time. In the sketch below, the wheel is shown with its center at 0,0. Three other points have been drawn at different locations on the wheel. The screen was left out of this sketch so that the paths of each point could be seen more easily.



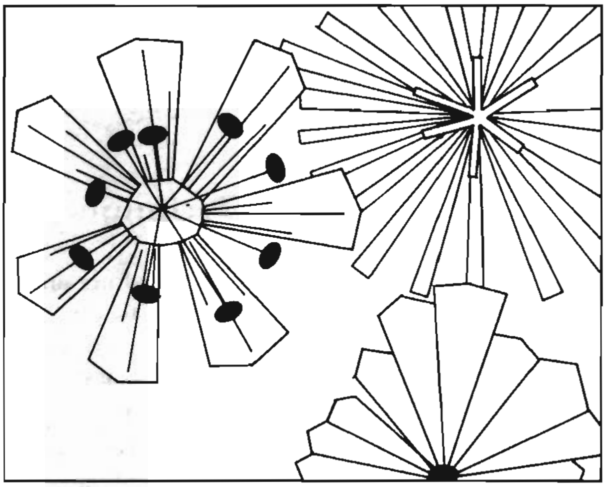
The rotation path of each point is shown by arrows. Notice that each point follows a perfect circular path around 0,0 when the wheel is turned. This path will be the same whether the wheel is turned to the left or to the right. The basic idea behind rotation is that a point always stays the same distance from 0,0 as it moves around this anchor point. The only way this is possible is for the point to move along an imaginary circule centered at 0,0.

We will refer to a shape's "rotation path" frequently during the course of this chapter. This path is the imaginary circle the shape will follow as its "drawing wheel" turns. If you have problems determining the circular path a shape will take, think about the drawing wheel. Also, keep in mind that only the shape's endpoints are being rotated. This will be of particular help when you are dealing with a shape that is drawn over 0,0—like the airplane shown earlier. Shapes drawn around 0,0 have no specific path that you can readily imagine. Recalling that each endpoint has its own circular path may be helpful.

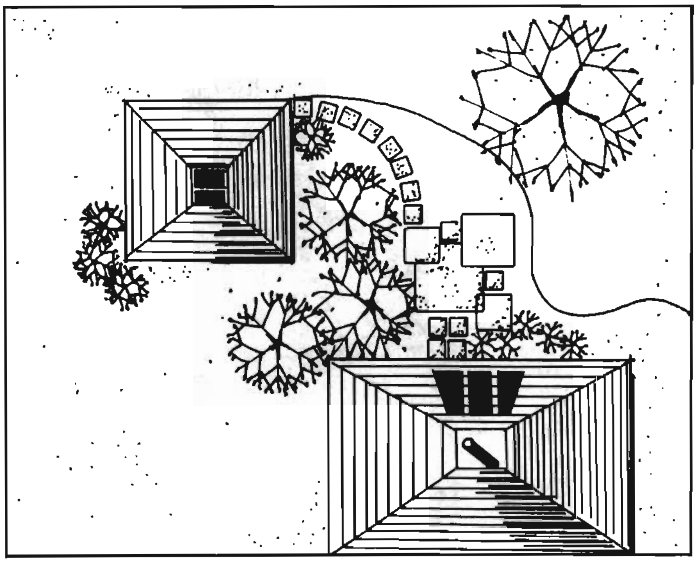
A few designs incorporating rotation will best demonstrate the usefulness of such a tool. The sketch below, for example, was created by rotating and translating a small square. Observe how the squares seem to radiate from the center of the design. The squares are lined up like the spokes to a wheel. The center—the hub—has been carefully located to the left of the screen's center for the purpose of variety.



A flower arrangement can be produced from a few scaled, rotated and translated shapes, as shown here:



Similarly, the shrubbery in our landscape design was also created by utilizing scaling and rotation. Each shrub is a complete rotation of one crooked line. The crooked line is scaled up and down for various versions of the shrub.



Any design or object that can be made by repeating a shape along a circular path is perfect for the ROTATE A SHAPE tool. Stars, snowflakes, and sun rays are just a few examples. Take a moment or two to come up with your own "rotation-oriented" shapes and designs. Make a note of any that you might like to try.

LOAD the "CHAPTER 5" program into memory before continuing. Then list line 172 and make sure it has C set to 1008 (C=1008). RUN the ZAP routine.

Simple Rotation Techniques

You now understand that rotation moves a shape along an imaginary circular path. This circular path has its center at 0,0 and runs through the shape. You can change the circular path of rotation by translating the shape to a different "starting position." This is the same principle used when a compass rotates a pencil to draw a circle.

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 the circle you desire, and rotate the compass around the central point.

The ROTATE A SHAPE tool will rotate your shape in the same fashion. The further away from 0,0 the shape is located, the larger the circular path on which it will be rotated. The closer the shape is to 0,0, the smaller the circular path it will take. The tool itself handles all of the mathematics necessary to peform the rotation.

The ROTATE A SHAPE tool is shown below. Note that in line 162, the symbol for pi ("π") is used. This symbol can be typed by holding down a **SHIFT** key while pressing ↑ (located to the left of **RESTORE**.) Add this tool to your program now:

```
160 REM::::::ROTATE A SHAPE
161 GOSUB 12Ø
162 RD=RO/36Ø*2*π
163 T(Ø,Ø)=COS(RD): T(Ø,1)=SIN(RD)/1.23
164 T(1,Ø)=-SIN(RD)*1.23: T(1,1)=COS(RD)
165 GOTO 13Ø
```

You need to specify four things in the main routine when you want to rotate a shape. Those four things are:

(1) The shape you want to rotate;
(2) The "starting position" of the shape;
(3) How far you want the shape rotated; and
(4) In which direction you want it rotated.

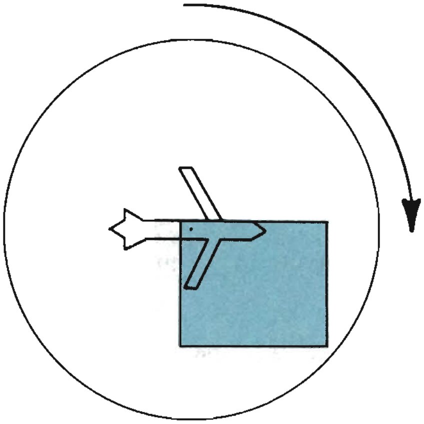
Step 1 is satisfied by retrieving the desired shape with the RETRIEVE A SHAPE tool. Step 2 is accomplished with the TRANSLATE A SHAPE tool. Again, the starting position of the shape will determine how large the circle of rotatation is. If the shape is defined around 0,0, you may need to translate it to a new starting position. Steps (3) and (4) are both handled by assigning a value to the variable RO ("ROtate").

The value you assign to RO indicates how many *degrees* you would like the shape rotated. The term "degree" has it roots in the early studies of the earth's rotation around the sun. It was found that the earth took approximately 360 days (one year) to travel the complete orbit. From these early discoveries in astronomy, it was established that every circle has *360 degrees*.

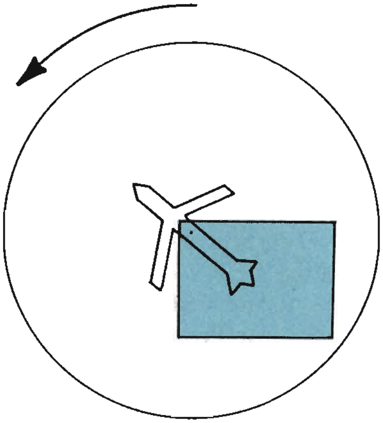
Degrees, then, do not describe physical distance in the same manner as miles, inches, kilometers or centimeters do. Thirty degrees along one circle might be two inches in length, while thirty degrees along another circle might be six inches in length. Nevertheless, degrees can be used as an accurate measurement of the direction and distance you want a shape rotated.

Since a full circle has 360 degrees, setting RO equal to 360 will rotate your shape entirely around its rotation path. This would bring it back to its starting position. Setting RO equal to 180 will rotate the shape halfway around its rotation path. The shape will rotate a quarter way around its rotation path if RO is set to 90, and an eighth of the way around its rotation path if RO is set to 45.

Now you know how to control the distance a shape is rotated (Step 3). The final step is to specify in which direction the shape is to be rotated. A shape can be rotated to the right or to the left, as the imaginary "drawing wheel" has shown. Positive degrees of rotation (RO=+?) will rotate the shape to the right:

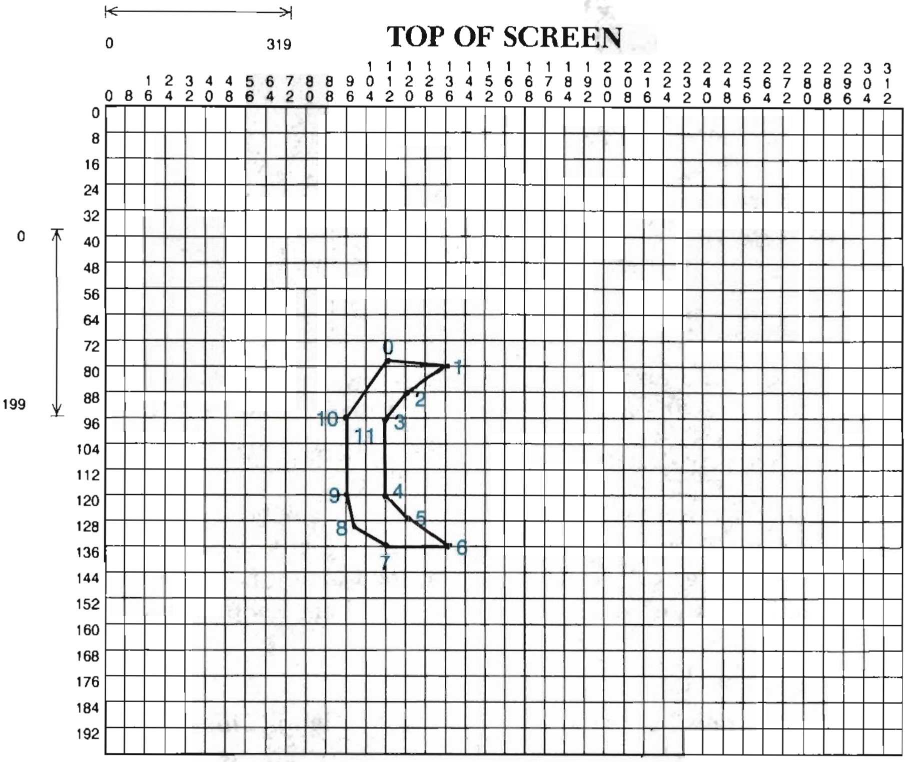


Negative degrees of rotation (RO=-?) will rotate the shape to the left:



You can thus complete both steps 3 and 4 by correctly setting the RO variable.

We will demonstrate how the ROTATE A SHAPE tool works by using the moon shape sketched below.



X, Y PIXEL POINTS

Point #11 is a paint point that will be used to fill the moon in with color. We have not anchored the moon around 0,0, and so it will rotate in a large circular path around the anchor point. Add the defining DATA statements as follows:

```
1ØØ8 DATA "MOON",11,1Ø
1Ø1Ø DATA 112, 79,136, 8Ø,119, 88
1Ø12 DATA 111, 96,111,119,12Ø,127
1Ø14 DATA 137,136,111,136, 99,13Ø
1Ø16 DATA  96,119, 96, 96,1Ø4,1Ø4
1Ø18 DATA  Ø, 1, 1, 2, 2, 3, 3, 4
1Ø2Ø DATA  4, 5, 5, 6, 6, 7, 7, 8
1Ø22 DATA  8, 9, 9,1Ø,1Ø, Ø
```

We will begin by drawing the shape and painting it. Next, we will rotate the moon -2 degrees (turning the drawing wheel to the left), and draw it again. This, along with our usual set of graphics tasks, is accomplished by adding the following main routine lines to your program:

```
2ØØØ GOSUB 1Ø: C=11: GOSUB 3Ø
2Ø1Ø POKE 5328Ø,C
2Ø2Ø SE$="MOON": GOSUB 8ØØ
2Ø3Ø C=7: GOSUB 11Ø
2Ø4Ø GOSUB 9Ø: PP=11: GOSUB 6Ø
2Ø5Ø RO = -2
2Ø7Ø GOSUB 16Ø: GOSUB 9Ø
6ØØØ GET A$
6Ø1Ø IF A$ = "<—" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
```

RUN the program.

Your screen should turn a dark grey, including the border that surrounds the high resolution drawing area. The moon will then be plotted and painted in a bright yellow (you may need to adjust your monitor to achieve this color). The rotated moon will be drawn a few moments later.

What has happened is that the original endpoints have been rotated two degrees in a circular path to the left. This moves them up a little bit on the screen. The distance that two degrees covers is based on the rotation path of the shape. The rotation circle is divided into 360 small arcs, and the endpoints are moved the distance of two arcs.

Return to text mode.

The next program lines you will enter create a loop. This loop will process twenty-seven times, each time rotating the moon shape -2 degrees and then drawing it. Type these lines:

```
2Ø6Ø FOR L = 1 TO 27
2Ø8Ø NEXT L
```

RUN the program again. The moon will rotate counterclockwise, and be redrawn for every two degrees of rotation. Soon, you will begin to see an actual circle being formed by the top left edges of all the moons. You are drawing the rotation path of this shape with the shape itself. Press ←, and the computer will return to text mode when the twenty-eighth moon has been drawn.

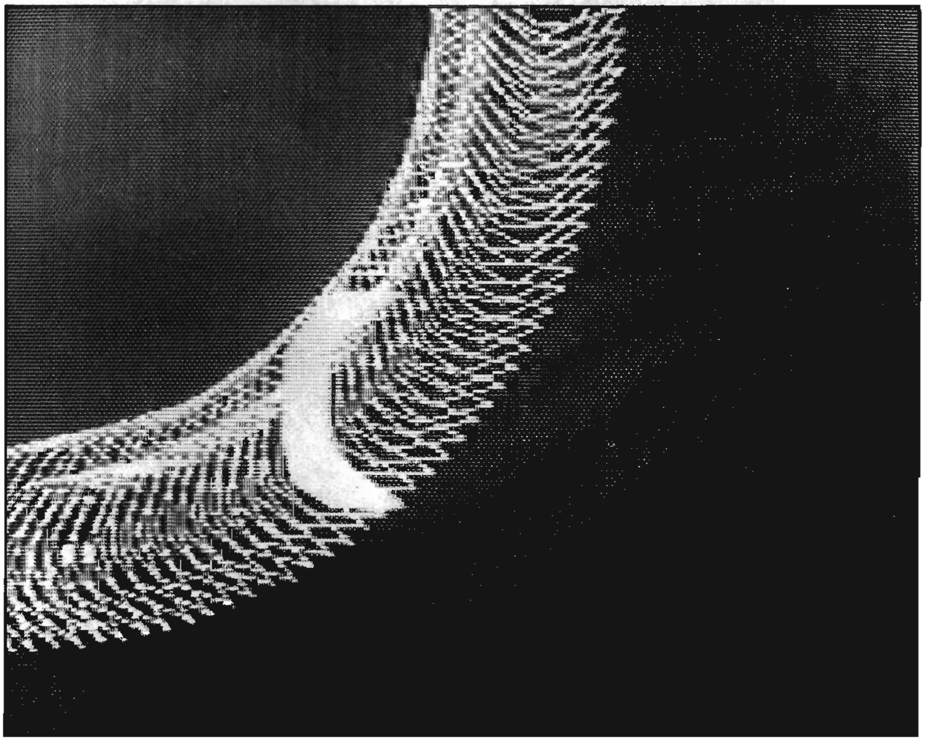
You can easily reverse the direction of rotation by setting RO equal to a positive value. Add program lines 2090 through 3030 to your program:

```
2Ø9Ø GOSUB 11Ø
3ØØØ RO = 2
3Ø1Ø FOR L = 1 TO 2Ø
3Ø2Ø GOSUB 16Ø: GOSUB 9Ø
3Ø3Ø NEXT L
```

You may already know what these new program lines will do. Program line 2090 clears the C matrix so that we can again work with the original (unrotated) shape. Line 3000 sets RO to a positive value, moving the rotation in a clockwise direction instead of counterclockwise. Finally, lines 3010 through 3030 rotate and draw the moon twenty different times in this new direction.

RUN the program.

You will need to wait as the first portion of the program executes as before. The new section of program lines will begin executing after the twenty-eighth moon has been drawn. When this happens, the circular path will be picked up at the original moon and taken in the opposite direction. The design pictured below should be plotted on your screen.



Leave the finished design on your screen for a moment as we perform a few mental calculations. We started by rotating the moon twenty-seven different times, each time moving it two degrees counterclockwise:

27 * 2 degrees = 54 degrees counterclockwise

Next, we took the original shape and rotated it twenty times clockwise, still moving it two degrees each time:

20 * 2 degrees = 40 degrees clockwise

This means that the shape was rotated and drawn a total of 94 degrees:

54 counterclockwise + 40 clockwise = 94 total degrees

Since a full circle is a span of 360 degrees, our finished design should be just a little larger than a quarter of a circle (94 / 360 = .27). You can see that it is by looking at your screen.

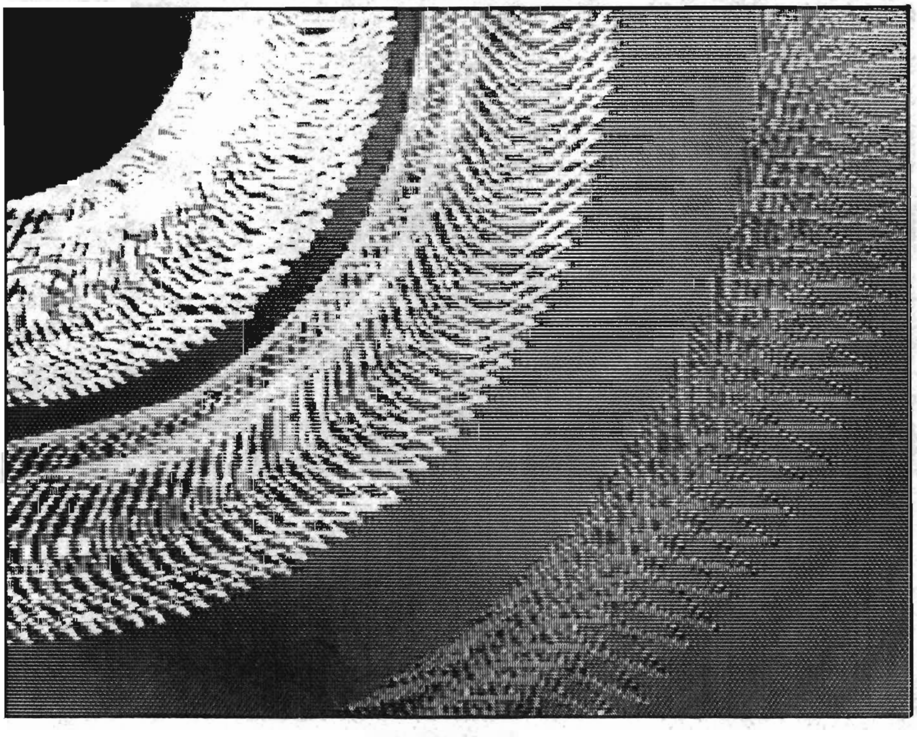
Modify lines 2040 and 2090 as follows:

```
2Ø4Ø GOSUB 9Ø
2Ø9Ø GOSUB 11Ø: GOSUB 14Ø
```

Add program line 2035, as well as lines 3040 through 3070:

```
2Ø35 FOR V = 1 TO 3
3Ø4Ø GOSUB 11Ø
3Ø5Ø IF V=1 THEN XT=15Ø: YT=-1Ø: GOSUB 14Ø: C=2
3Ø6Ø IF V=2 THEN XT=-5Ø: YT=-5Ø: GOSUB 14Ø: C=1
3Ø7Ø NEXT V
```

RUN the program a final time. This program will draw three versions of the moon wheel, each in a different color. The center, yellow wheel will be drawn first. Next, a larger red wheel will be drawn to the right side of the screen. Finally, a white inner wheel will be drawn close to the screen origin. The result should look like the following:



Note how the smallest wheel (the wheel closest to 0,0) appears to be packed more solidly with shapes. The larger, outside wheel seems to be more spread out, less condensed. This is because we have plotted the same number of moons on two very different sized circles. The larger the circle, the more spread out the shapes become. The smaller the circle, the more compressed the design will be. We would need to plot moons more often (say, every .50 degrees) on the larger wheel to obtain the same density as the smallest wheel.

Return to text mode and list lines 2035 through 3070 on your screen. The first item of note is in line 2090. This line calls the translation tool, but without first giving an XT or YT value. Since you don't set XT and YT, they are automatically set to 0 by the computer. The shape is then translated 0 columns in X, and 0 rows in Y (keeping it in the same place).

The V loop will be processed three times, once for each wheel to be drawn. The first time through the loop we don't want the shape translated. We do, however, want it translated to different locations the second and third time through the loop. Line 3050 translates the shape the required distance for the first loop. Notice, though, that after drawing twenty-seven translated moons in the first L loop (lines 2060-2080), the C matrix is cleared. This means we must translate the shape again for the second L loop. XT and YT are already set to the necessary offset values, so Tool 140 is called. This same process is used for the third and final V loop.

Spend some time practicing with your new tool. You should try changing the degrees of rotation, working with both positive and negative degrees. Translate the shape to new "starting positions" before each rotation loop. (Make sure that one wheel does not interfere with another wheel's color blocks if different colored wheels are to be drawn.)

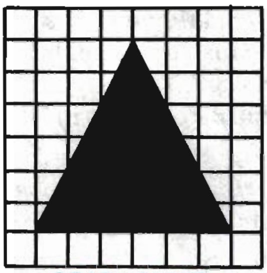
Refer back to Chapter 5 with any program problems you encounter. You might not be able to see a rotated shape being drawn if you've translated/rotated it off the screen. Other than that, no new problems should crop up.

Overcoming Aspect Ratio Problems

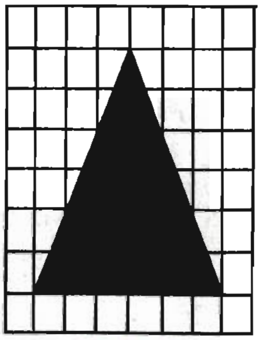
We pointed out very early in the book that we would be using graph paper that had elongated boxes (boxes that are higher than they are wide). We created this special graph paper to correspond with the elongated pixels on your screen. The ratio of a pixel's height to its width is called the screen's *aspect ratio.* This ratio will vary from screen to screen, but it is always approximately 1.23 to 1 (1.23:1).

This unbalanced aspect ratio causes a problem for anyone trying to use regular graph paper to work out designs. A shape drawn on regular graph paper will always be slightly elongated when plotted on the screen.

Take the two triangles below, for example. The triangle on the left is drawn on regular graph paper. The triangle on the right is what the first triangle will look like when plotted on the screen.

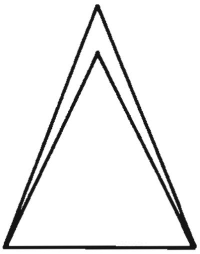


GRAPH PAPER



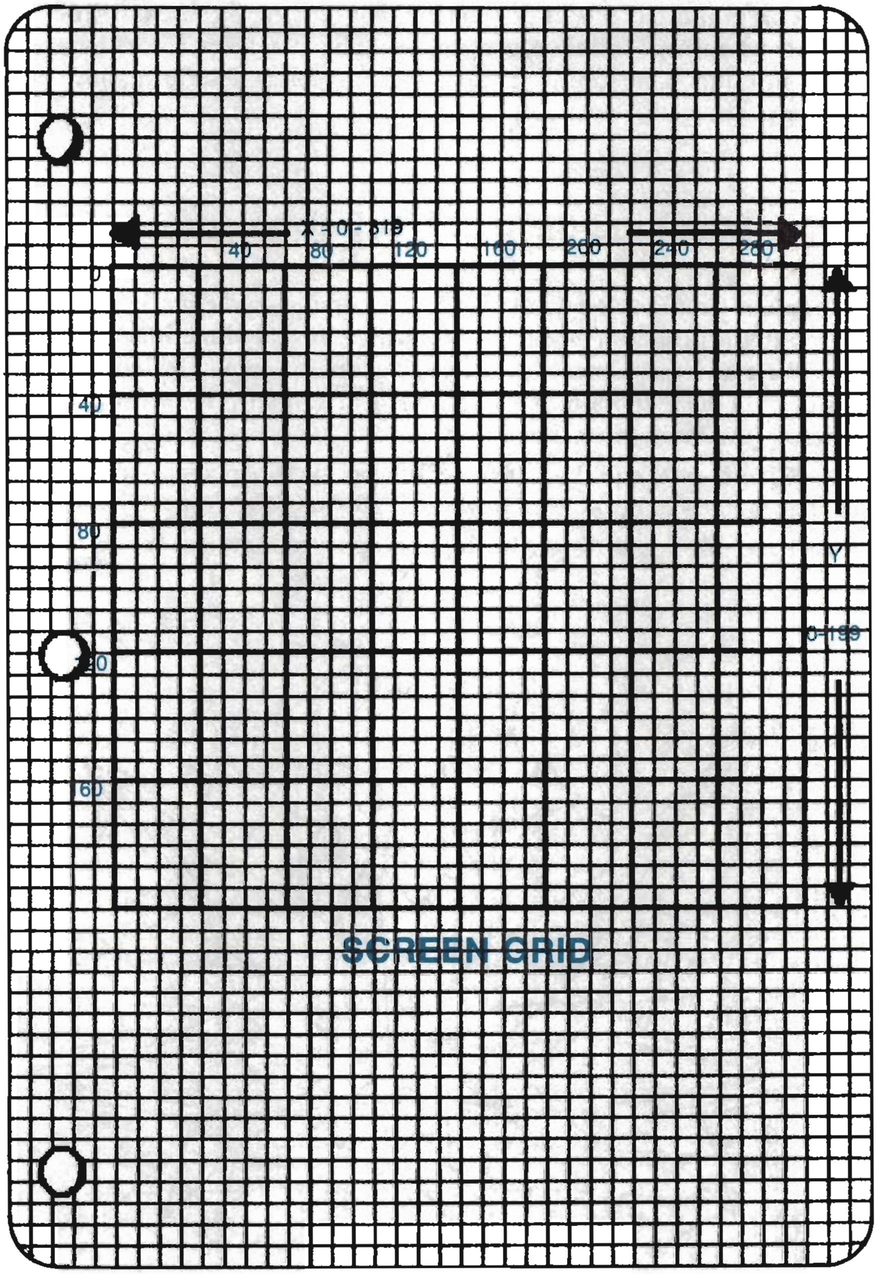
SCREEN

A tracing of each triangle, placed on top of each other, reveals the significant difference the screen's aspect ratio can make:



We recommend that you use copies of our grid when you draw your own designs. We are aware that this will not always be possible. Fortunately, there is a way to use graph paper effectively when planning your designs. There will be a slight loss of accuracy with this method—about 1/200th of a percent—but this loss is obviously not going to make a significant difference on the screen.

The sketch below shows a piece of graph paper that has a screen grid drawn on it. This screen grid is 1.25 times higher than it is wide. This is accomplished by using five blocks in height to every four blocks in width. The result is a screen grid whose aspect ratio is 5 to 4, which is the same as 1.25 to 1 (5/4 = 1.25).



The large blocks on the grid can represent any block of pixels you desire. If you want them to represent a 20 x 20 block of pixels, that's fine. The important thing is that they represent the same number of pixels in height as in width.

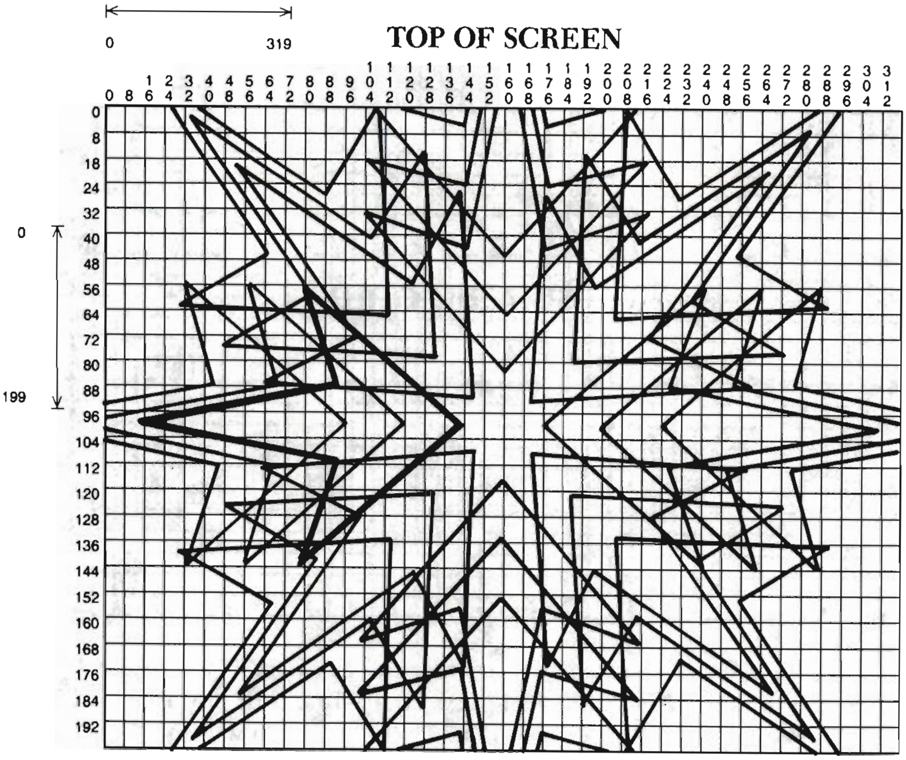
The color blocks are no longer defined when you modify graph paper in this way. Color blocks are *always* eight pixels by eight pixels. Rearranging your drawing area on graph paper will not change that fact. Again, use our specially designed paper whenever possible. Our special graph paper keeps color blocks and aspect ratios in mind. You should modify graph paper as described above only when a copier is not readily available.

Planning the Results of Rotation

Rotation can be a difficult thing to plan for in a design, and equally difficult to control through the program. The idea of rotating shapes in imaginary locations off the screen is not an easy concept to grasp. The rotation tool will always remain somewhat of a mystery, though, if you do not learn to effectively use offscreen pixels to define and rotate your shapes.

List line 172 and set C equal to 2000 (we'll keep the moon shape so that your library of shapes starts to build from now on). RUN the ZAP routine.

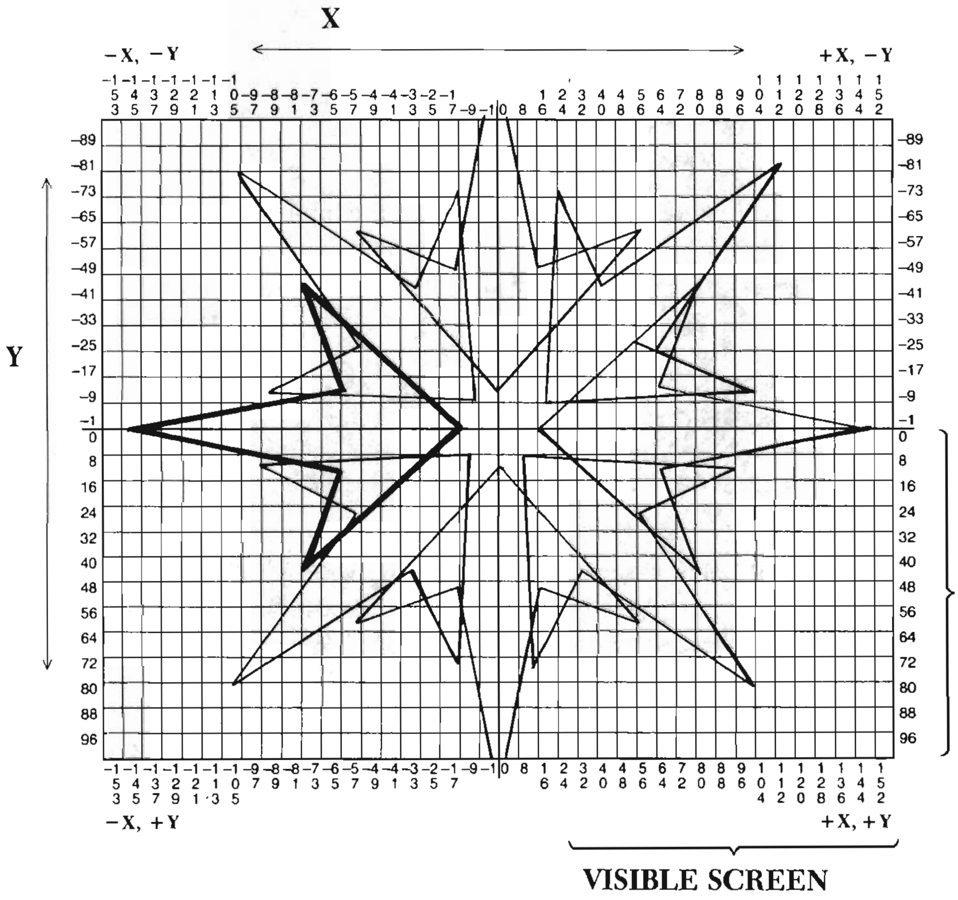
We are going to take you through the steps necessary to draw this design on your screen:



X, Y PIXEL POINTS

If you were to dissect this design, you would find that it was created by rotating and translating a single arrow many times. There are three "rings" of arrows, each ring made up of eight arrows rotated 45 degrees apart from each other.

We will start by rotating and drawing the inner ring of arrows. The original arrow will be defined so that it falls just left of 0,0 (see the drawing below). This arrow will be translated 159 columns right, and 99 columns down. This situates the arrow just left of the center screen, where it will be drawn. That arrow will then be translated back next to 0,0, where it will be rotated 45 degrees. The rotated arrow will then be moved to the center of the screen and drawn. This translate, rotate, translate and draw process will continue until all eight arrows have been drawn.



The DATA statements below will define the arrow at the proper starting position. Type them into your program.

```
1Ø24 DATA "ARROW",5,5
1Ø26 DATA -151,  Ø,-65,-12,-8Ø,-44
1Ø28 DATA  -16,  Ø,-8Ø, 44,-65, 12
1Ø3Ø DATA   Ø, 1, 1, 2, 2, 3, 3, 4
1Ø32 DATA   4, 5, 5, Ø
```

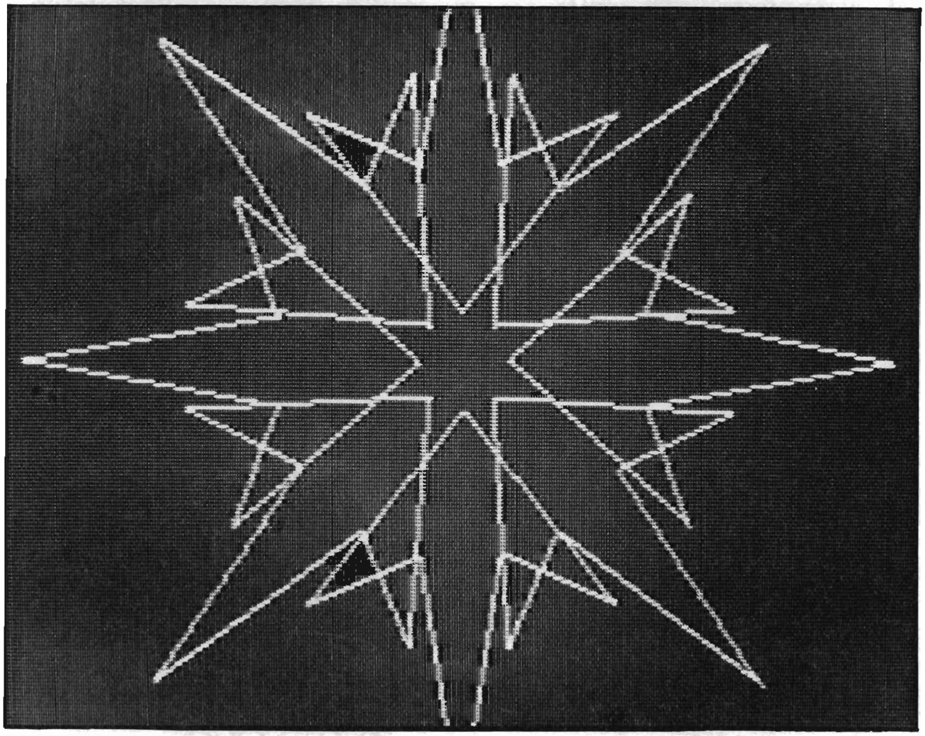
The next set of program lines set the program up for graphics and retrieve the arrow shape:

```
2ØØØ GOSUB 1Ø: C=6: GOSUB 3Ø
2Ø1Ø POKE 5328Ø,C
2Ø2Ø SE$="ARROW": GOSUB 8ØØ
2Ø3Ø C=1: GOSUB 11Ø
6ØØØ GET A$
6Ø1Ø IF A$ = "<—" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
```

Finally, the following program lines provide the information needed to draw a ring of arrows around the center area of the screen:

```
2Ø6Ø XT=159: YT=99: GOSUB 14Ø
2Ø7Ø GOSUB 9Ø
2Ø8Ø FOR V = 1 TO 7
2Ø9Ø XT=-XT: YT=-YT: GOSUB 14Ø
3ØØØ RO=45: GOSUB 16Ø
3Ø1Ø XT=-XT: YT=-YT: GOSUB 14Ø
3Ø2Ø GOSUB 9Ø
3Ø3Ø NEXT V
```

This program will be easier to understand after you've watched it in action, so take the time to RUN it now. One at a time, the arrows should be plotted. Each will appear to point at the very center of the screen. The eighth arrow will complete the circle made by the program, and your screen will display the following:



Return to text mode and list lines 2000 and higher. Lines 2090, 3000, 3010, and 3020 do most of the work. For each V loop, these lines will:

(1) Translate the arrow back to its *last* position around 0,0 (line 2090 reverses the translation that moved the arrow to the center screen);

(2) Rotate the shape 45 degrees more around 0,0 (line 3000);

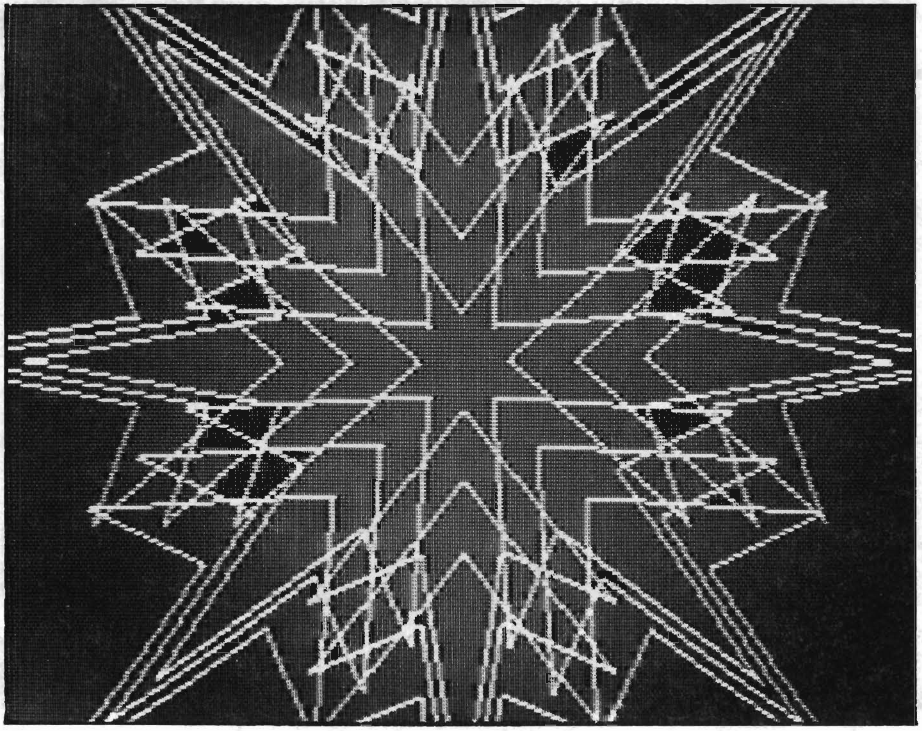
(3) Translate the arrow back to the center screen and draw it (lines 3010 and 3020).

Suppose the translations had been left out of this program. If you had merely rotated each arrow 45 degrees around 0,0, and then drawn the arrow, for a total of seven times, the resulting ring would have been drawn around 0,0 instead of around 159,99. Since the program always translated the shape 159 columns right and 99 rows down, you moved the entire ring to the center of the screen.

The next program lines you will enter create an L loop. This loop will draw three rings of eight arrows, each ring a little larger than the one before. These are the program lines to add:

```
2Ø4Ø FOR L = 1 TO 3
2Ø5Ø XT=M: YT=Ø: GOSUB 14Ø
3Ø4Ø GOSUB 11Ø
3Ø5Ø M = M-24: NEXT L
```

RUN the program. The first ring of arrows will be redrawn as before. After that, a new ring, and then another, follow. Each ring is slightly larger than the one before. The program will be finished when the following design is displayed:



Return to text mode and list lines 2040 and higher on the screen. The L loop executed three times. The first time through the loop, the shape was translated 0 along the X axis (M is set to 0) and 0 along the Y. This left it at its original starting position. The ring of eight arrows was created from that starting position.

The variable M was set to -24 when the first loop was completed. Thus, at the beginning of the second loop, the arrow was translated -24 pixels on the X axis, and 0 pixels on the Y axis. This new starting position was used to create the second ring of arrows.

The variable M was then set to -48 (M =-24-24) at the end of the second loop. This set the third loop up to translate the arrow to its third and final starting position. The third ring of arrows was drawn from that new position.

That completes this chapter's discussion and exercises concerning rotation. There is much more that can, and should, be explored with this new tool that space prohibits us from covering. We cannot emphasize enough the knowledge that can be gained through practice. Draw shapes, translate them, scale them, and rotate them. Draw shapes on the screen, off the screen, and around 0,0. Rotate shapes with negative degrees and positive degrees. Paint your shapes. Paint with and without pre-defined paint points. (The center area of this arrow design can be painted with a program line 3060 setting X=159, Y=99, and a GOSUB 62.)

The next section discusses the mathematics involved in rotation, if you are interested. The tool box for ROTATE A SHAPE is presented there, followed by the design ideas relating to rotation, and the summary section.

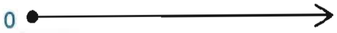
The Mathematics of Rotation

The math behind rotation is more complicated than the math behind translation or scaling. To begin with, rotation can be performed on *polar coordinates* more easily than it can be on Cartesian coordinates. This means that the Cartesian coordinates of your shape must first be converted to their polar equivalents. The rest of this section is devoted to a discussion of polar coordinates, and how those coordinates are used with your ROTATE A SHAPE tool.

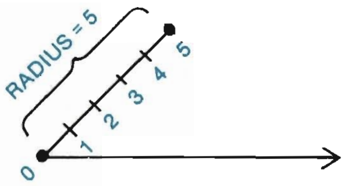
> *Note:* We will discuss the rotation of a single point as we explain polar coordinates and rotation in general. A rotated shape is produced by rotating its endpoints, and then connecting those endpoints in the manner described in the L% array.

The polar coordinate system is another method of describing and displaying data graphically. It is used to describe the same points that the Cartesian system does, but in a different manner. This system is based on two-dimensional space, and cannot be used to describe a three-dimensional object.

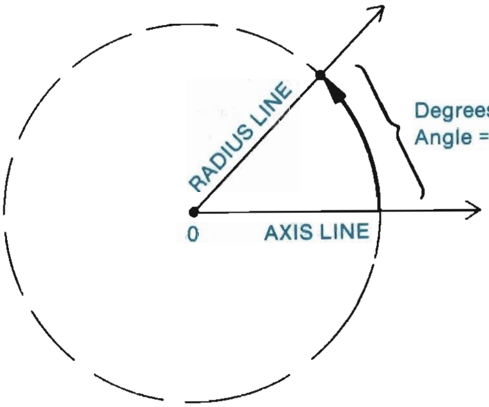
There is only one axis in the polar coordinate system. This axis lies on a plane, beginning at point 0, and runs horizontally to the right:



Any point that lies on the plane can be located and described in terms of the *radius* and *angle* it creates. The radius is the distance between the point and the origin (Point 0):



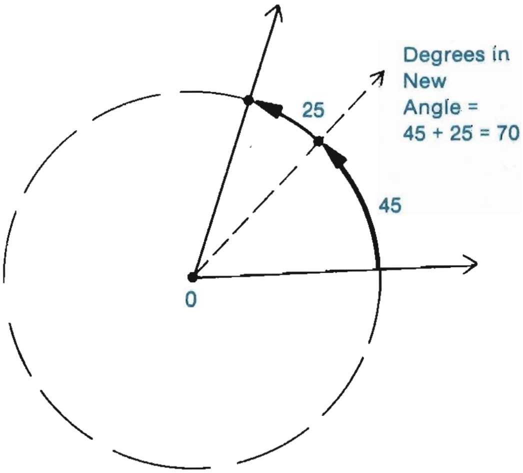
An angle is formed by the intersection of the axis and the point's radius line (the line that connects the point to the origin). The size of the angle is measured in terms of *degrees*. The size is equal to the number of degrees between the axis and radius line.



Each point has its own unique combination of radius length and angle degrees. Many points have the same radius (forming a circle around point 0), and many points have the same angle (forming a line that extends out from point 0), but no two points share the same radius *and* angle.

Let's suppose we wanted to rotate the point plotted above (Radius = 5, Degrees in Angle = 45) 25 degrees. What new polar coordinate would we arrive at?

A point, no matter where it is rotated, retains the same radius. Rotating a point with a radius of 5 produces a point with a radius of 5. It is the angle, then, that changes in a rotated point. Look at the next diagram:



Notice that by adding the degrees of the original point (45) to the degrees of rotation desired (25), you arrive at the degrees that the rotated point will have (70). This gives you all the information necessary to describe the new point in polar coordinates: Radius = 5, Degrees in Angle = 70.

We should point out that the direction of rotation is reversed on your screen. In polar coordinates, rotating a point 25 degrees will rotate it *counter*clockwise around 0,0. The opposite is true on your screen. This was done to correspond to the reversed Y axis in Cartesian coordinates.

We can begin explaining the math now. The formulas we use are all given an "Equation #" for reference throughout the discussion. Basic trigonometry functions (Sine and Cosine) are used without explanation. A description of each variable used in the equations is as follows:

X= X coordinate of original point
Y= Y coordinate of original point
X' = X coordinate of rotated point
Y' = Y coordinate of rotated point
R= Radius of X,Y
D1 = Degrees of Angle created by X,Y
D2 = Degrees of rotation desired

The first formulas to examine are those that convert polar coordinates to Cartesian coordinates:

R*COS(D1) = X [Equation #1]
R*SIN(D1) = Y [Equation #2]

These formulas are explained in greater detail in the appendices. You will need to trust us at this point that they are correct.

Based on Equations #1 and #2, we can arrive at similar equations that describe the point after it has been rotated. We know that the new point will have the same radius. We also know that the new point will have an angle equal to D1 *plus* the desired degrees of rotation (D2). This gives us the following formulas:

R*COS(D1+D2) = X' [Equation #3]
R*SIN(D1+D2) = Y' [Equation #4]

Now let's turn our focus on the formula for X' (Equation #3). This equation can be rewritten as:

X' = R*(COS(D1)*COS(D2) -SIN(D1)*SIN(D2)) [Equation #5]

Equation #3 and Equation #5 will always produce the same results (given the same values for each variable). Each is just a different way of expressing the same mathematical procedure. A more detailed explanation of this conversion is given in the appendices.

The variable R in Equation #5 can be distributed as follows:

X' = R*COS(D1)*COS(D2) -R*SIN(D1)*SIN(D2) [Equation #6]

Look closely at Equation #6. You will see that Equation #1 (R*COS(D1)) and Equation #2 (R*SIN(D1)) are a part of this equation. Since Equation #1 is equal to X, and Equation #2 is equal to Y, the variables can replace portions of the Equation #6 as follows:

X' = X*COS(D2) -Y*SIN(D2) [Equation #7]

The values for X, Y and D2 are known values, so we can now solve for the value of X'.

The next coordinate to find is Y' (the Y coordinate of a rotated point). Again, the formula that will yield this value is:

Y' = R*COS(D1+D2) [Equation #8]

This formula can be re-written as follows:

Y' = R*[SIN(D1)*COS(D2) + COS(D1)*SIN(D2)] [Equation #9]

Distributing the R variable throughout produces:

Y' = R*SIN(D1)*COS(D2) + R*COS(D1)*SIN(D2) [Equation #10]

This formula also contains Equations #1 and #2. Substituting their equivalents (X and Y) in the formula gives us:

Y' = Y*COS(D2) + X*SIN(D2) [Equation #11]

This formula can be solved by inserting the known variable values for Y, X and D2.

There is one more thing, however, that we had to account for in our ROTATE A SHAPE tool: an adjustment for the screen's aspect ratio. The aspect ratio, as discussed in the previous section, is the ratio of a pixel's height to its width. Pixels are approximately 1.23 times higher than they are wide. This means that a horizontal line of 10 pixels is shorter in length than a vertical line of 10 pixels. If we were to rotate a horizontal line 90 degrees, keeping it the same number of pixels in length, we would end up with a line that was physically longer than the one we started with. This can cause many shapes to become misshapen.

Our formulas must include an adjustment so that, as a shape rotates, it does not become elongated. The final formulas, which include the adjustment for the screen's aspect ratio, are:

X' = X (COS(D2)) - Y (SIN(D2)) * 1.23
Y' = Y (COS(D2)) + X (SIN(D2)) / 1.23

TOOL 160 :::::: ROTATE A SHAPE

```
16Ø REM:::::::ROTATE A SHAPE
161 GOSUB 12Ø
162 RD=RO/36Ø*2*π
163 T(Ø,Ø)=COS(RD): T(Ø,1)=SIN(RD)/1.23
164 T(1,Ø)=-SIN(RD)*1.23: T(1,1)=COS(RD)
165 GOTO 13Ø
```

What It Does: This tool will rotate a shape a specified number of degrees. The shape will rotate around 0,0, rotating in a clockwise direction for positive degrees, and rotating in a counterclockwise direction for negative degrees.

Example Use: The single variable that must be set to use this tool is RO. This variable should be set to the number of rotation degrees desired (e.g., RO=45). This should be followed immediately with a GOSUB 160 statement.

The shape that was last retrieved by Tool 800 will be rotated. It will be rotated around 0,0 according to its current position on/off the screen. Clearing out the C matrix with a GOSUB 110 statement will ensure that the original form of the shape is rotated.

Technical Description: Rotation is similar to the other transformation tools, except the T matrix looks like this:

T Matrix

	0	1	2
0	COS(0)	SIN(0)	0
1	-SIN(0)	COS(0)	0
2	0	0	1

Since the aspect ratio of the TV picture becomes a factor when a shape is rotated, we have adjusted this matrix so the shape's size is adjusted as it rotates. The adjusted T matrix looks like this:

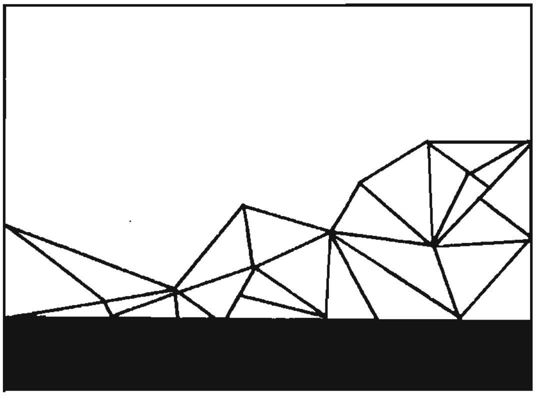
T Matrix

	0	1	2
0	COS(0)	SIN(0)/1.23	0
1	-SIN(0)°1.23	COS(0)	0
2	0	0	1

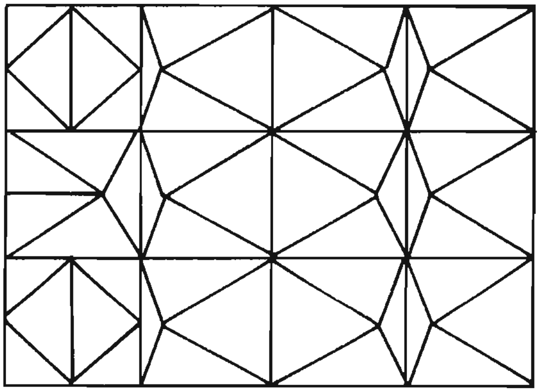
More Design Ideas

The sketches we will be showing here combine translation, scaling, and rotation techniques. These three transformation tools take three time-consuming tasks, that together require considerable expertise in math, and turn them into commands to the computer in the form of GOSUB statements.

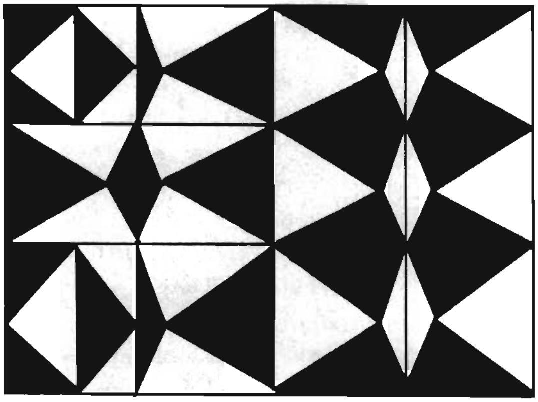
New shapes can be made from different shapes. A triangle was scaled, rotated and translated in order to compose the sculptural-looking design that follows. This design appears to be folded, like a paper model. It suggests depth and dimension.



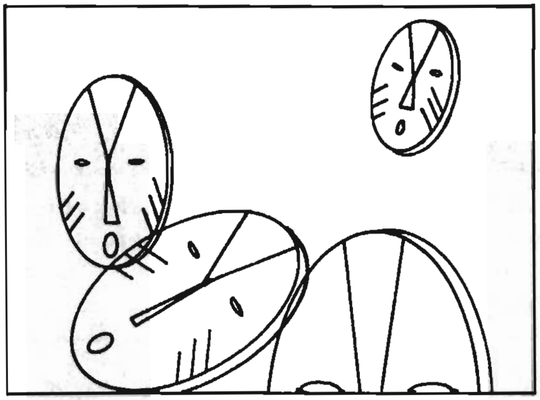
In the next sketch, a triangle has been scaled, rotated and translated to create a pattern. The previous sketch resembled the depth found in folded paper. This design emphasizes the linear qualities of shapes.



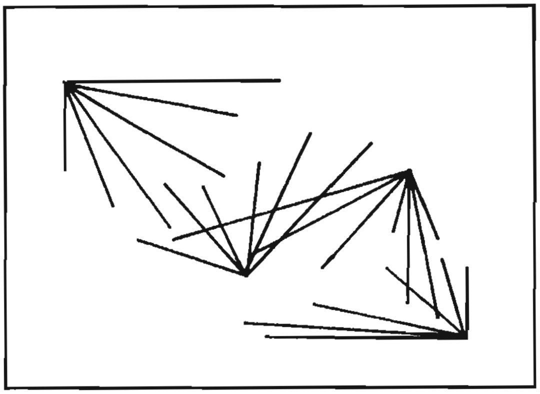
A new look to the last design can be achieved by coloring some of the shapes. The pattern is distinctly visible as a combination of solid shapes (see below). This characteristic is different than the linear features found above. The coloring here emphasizes the contrast between light and dark.



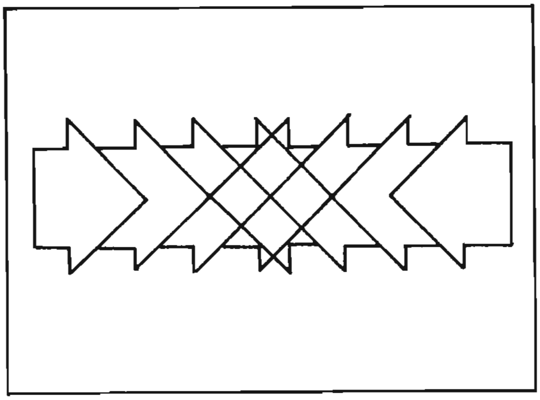
An African mask can be made from a few basic shapes, such as a circle and a triangle. These basic shapes have been scaled into new shapes in the next picture. For example, a circle was scaled to make the oval shape. Transforming and combining shapes can result in entirely new and intriguing designs. In this picture, the same mask is translated, rotated and scaled into different sizes. The rotation provides variation in the arrangement.



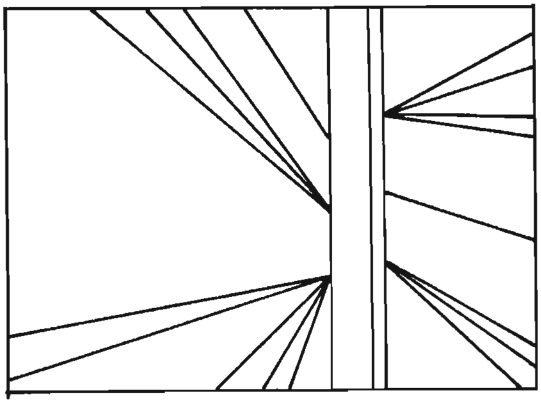
Each of the linear shapes below expresses a visual direction. Each shape directs the movement of our eyes, like a pointed arrow. The shapes appear to be floating in a flat space: no sense of depth is indicated. The overall design is arranged in a well-organized fashion.



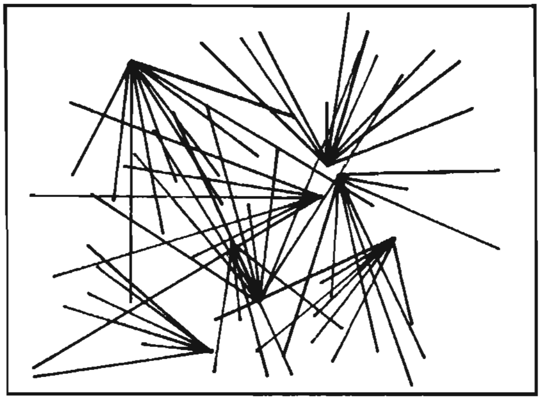
The arrow shapes in the following sketch are intentionally placed to direct attention to the middle of the screen. A diamond pattern is created at the center where the shapes overlap. The design is structured and carefully arranged.



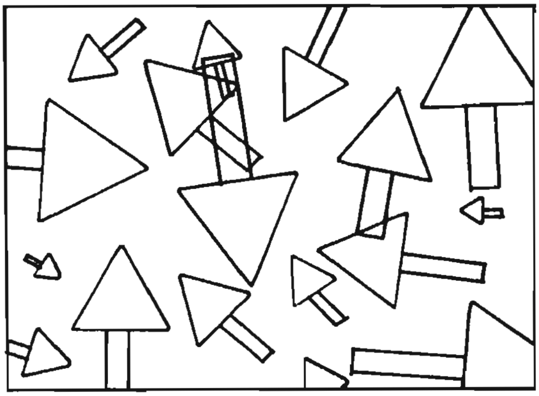
In the next picture, a tension occurs at the points where the angular, directional lines meet the vertical lines. The directional lines appear to be active and alive compared to the calm nature of the vertical lines. A sensation of attraction and repulsion is created by the direction of the lines.



In contrast to the above, the next composition demonstrates chaos, tension and disorganization. There seems to be greater tension and activity where the shapes repeat and overlap. The placement of shapes appears to be random.



Rotating, scaling and translating an arrow created the busy composition shown next. The arrangement of shapes is a random placement, with lines pointing in every direction. The resulting composition is busy with activity.



Summary

Congratulations! You have just completed what might be called "Part I" of this advanced graphics book. Your tool kit is now complete—containing all the advanced tools that don't deal directly with sprites. The basic tools, such as DRAW A SHAPE and PAINT A SHAPE, should be becoming second nature to you. The transformation tools (translate, scale, and rotate) take a little more practice, but in time will be just as easy to control and use as PLOT A LINE.

This chapter dealt specifically with rotation. You learned that two-dimensional rotation can be used to move a shape along a circular path. The exact path of movement was determined by the shape's relation to the screen's origin (0,0).

A shape that rests on 0,0 turns in place, pivoting around the origin. A shape that was unanchored (i.e., point 0,0 was not a part of it), would rotate around 0,0 in a complete and perfect circle. The distance the shape rotates depends on the number of degrees specified by the value of the variable RO.

One degree is 1/360th of the circular rotation path of the shape. If you set RO to 360, the shape will be rotated the entire distance around 0,0, placing it back where it started. Setting RO to 180 will rotate the shape half way around the circle. Similarly, setting RO to 90 will rotate the shape a quarter of the full distance; and setting RO to 45 will rotate the shape an eighth of the full distance.

The direction of rotation (clockwise vs. counterclockwise) is determined by the sign (+ or -) of RO's value. If you assign RO *positive* degrees (RO=+?), then rotation will be *clockwise*. If you assign RO *negative* degrees, then rotation will be *counterclockwise*.

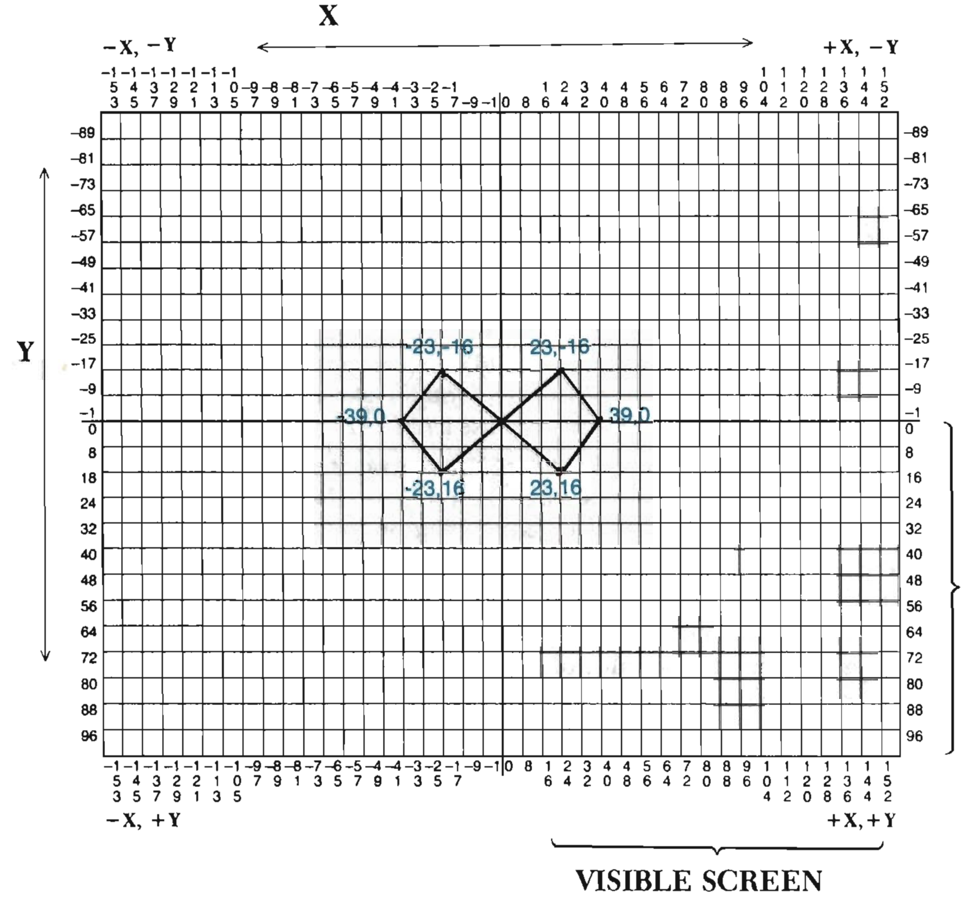
Refer to the above as you take on this chapter's exercise challenge. First SAVE your program under the filename "CHAPTER 6."

Exercise

Define the shape drawn below with DATA statements. In the main routine, begin by scaling the shape three times in height and three times in width. Next, create a loop that:

(1) Rotates the shape 15 degrees *counterclockwise;*
(2) Translates the shape to the center of the screen;
(3) Draws the shape;
(4) Translates the shape back around 0,0.

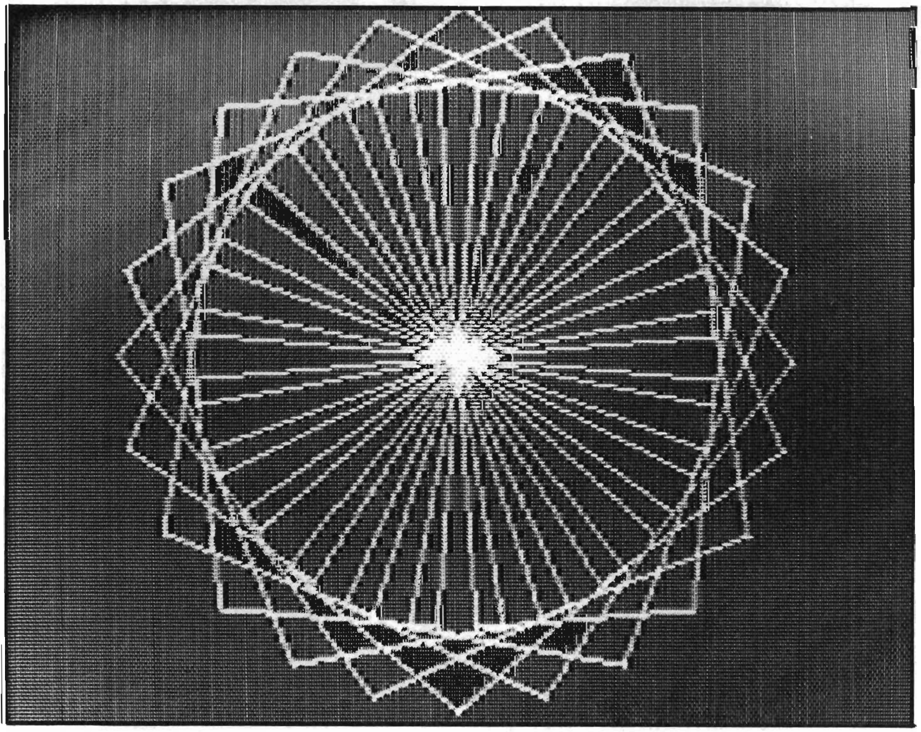
Cause this loop to execute 12 different times. You may draw in any colors you wish. We have chosen purple (4) as the background color, and white (1) as the foreground color.



Solution

A solution program is shown below, followed by a picture of the design intended.

```
1Ø34 DATA "SHAPE",5,5
1Ø36 DATA -23,-16, 23,-16, 39,  Ø
1Ø38 DATA  23, 16,-23, 16,-39,  Ø
1Ø4Ø DATA  Ø, 3, 4, 1, 1, 2, 2, 3
1Ø42 DATA  4, 5, 5, Ø
2ØØØ GOSUB 1Ø: C=4: GOSUB 3Ø
2Ø1Ø SE$="SHAPE": GOSUB 8ØØ
2Ø2Ø POKE 5328Ø,C
2Ø3Ø GOSUB 11Ø: C=1
2Ø4Ø XS=3: YS=3: GOSUB 15Ø
2Ø5Ø FOR L = 1 TO 12
2Ø6Ø RO=-15: GOSUB 16Ø
2Ø7Ø XT=159: YT=99: GOSUB 14Ø
2Ø8Ø GOSUB 9Ø
2Ø9Ø XT=-XT: YT=-YT: GOSUB 14Ø
3ØØØ NEXT L
```



You can enhance the design by painting some of the new shapes created during the rotation. These shapes cannot be painted until the design is complete (try painting as you go and you'll see what we mean). Modify the program as follows, and then RUN it:

```
1Ø34 DATA "SHAPE",7,5
1Ø39 DATA -37,  Ø, 37,  Ø
2Ø5Ø FOR L = 1 TO 24
2Ø85 IF L>12 THEN PP=6: GOSUB 6Ø: PP=7: GOSUB 6Ø
```

Chapters 7 and 8 deal in sprite graphics. Sprites are small movable figures that add animation to your designs. Chapter 7 covers "beginning" sprite concepts (how to create, move and place sprites). You should read this chapter, even if you are already familiar with sprite graphics, because Chapter 8 relies on Chapter 7's program.

Chapter 8 deals with more advanced sprite features, such as sprite collision detection and sprite joystick/keyboard control. If you've ever had a secret desire to create the next best-selling video game for the Commodore 64, you won't want to miss this chapter.

Finally, you are at a point where the appendices can be of real value to you. Perhaps the most immediate benefit will be gained from the SAVE PICTURE and PRINT PICTURE subroutine tools provided in these appendices. Color charts and design grids have also been placed there for easy access.

Chapter 7

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 spacecraft sprite across your picture.

Begin by LOADing the program "CHAPTER 6". Then, set C=1008 in line 172, and RUN the ZAP routine.

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?

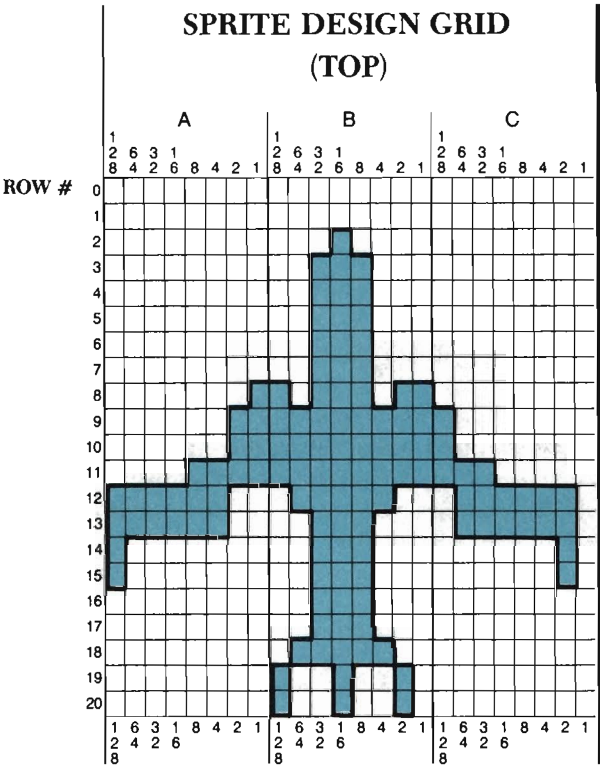
A sprite is like one of the little moving figures on a video arcade screen. It is a small shape that can be moved on the screen to create a cartoon-like picture of animation. "Animation" means that the picture shows movement. The versatility of sprites makes them different from any other shape you have 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 by moving a spacecraft sprite across 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 spacecraft 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 spacecraft sprite.



DATA STATEMENTS

BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
1010	DATA	0	0	0
1012	DATA	0	0	0
1014	DATA	0	16	0
1016	DATA	0	56	0
1018	DATA	0	56	0
1020	DATA	0	56	0
1022	DATA	0	56	0
1024	DATA	0	56	0
1026	DATA	1	187	0
1028	DATA	3	255	128
1030	DATA	3	255	128
1032	DATA	15	255	224
1034	DATA	252	124	126
1036	DATA	252	56	126
1038	DATA	128	56	2
1040	DATA	128	56	2
1042	DATA	0	56	0
1044	DATA	0	56	0
1046	DATA	0	124	0
1048	DATA	0	146	0
1050	DATA	0	146	0

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.

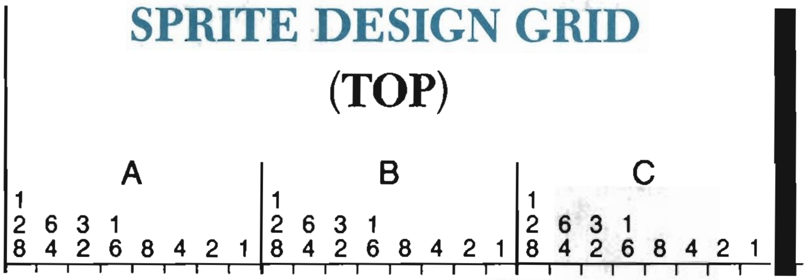
(2) *A sprite can be only one color.* The spacecraft sprite, for example, will be solid yellow.

A sprite can be displayed on the multi-color screen, but only in one color. There is a way to create multi-colored sprites, but this book will not be addressing that topic.

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 spacecraft design. For the first column this would involve the pixels in row 12 through 15. Each of these four 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.

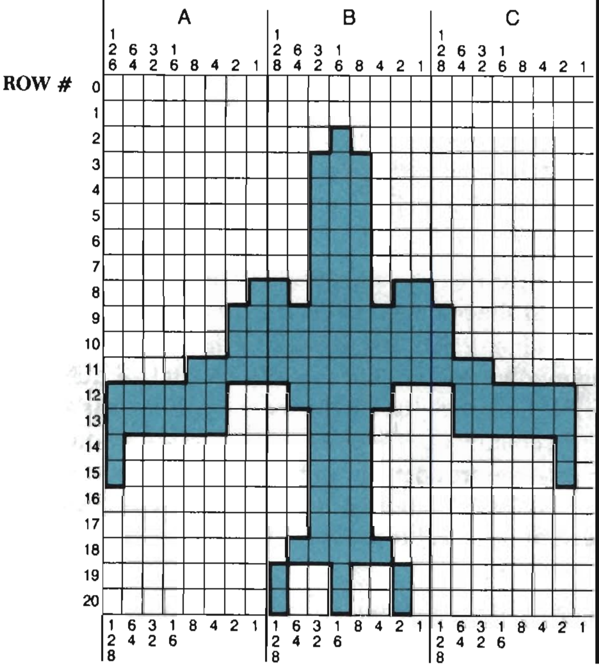
For each *row* in the grid, you must compute three totals using these values. First, you must total the values for the 8 squares in section A. Next, you must total the values for the 8 squares in section B. Finally, you must total the values for the 8 squares in section C. These three totals are written 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 spacecraft sprite, let's see how the 63 data sums were found.

SPRITE DESIGN GRID

(TOP)



DATA STATEMENTS

BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
1010	DATA	0	0	0
1012	DATA	0	0	0
1014	DATA	0	16	0
1016	DATA	0	56	0
1018	DATA	0	56	0
1020	DATA	0	56	0
1022	DATA	0	56	0
1024	DATA	0	56	0
1026	DATA	1	187	0
1028	DATA	3	255	128
1030	DATA	3	255	128
1032	DATA	15	255	224
1034	DATA	252	124	126
1036	DATA	252	56	126
1038	DATA	128	56	2
1040	DATA	128	56	2
1042	DATA	0	56	0
1044	DATA	0	56	0
1046	DATA	0	124	0
1048	DATA	0	146	0
1050	DATA	0	146	0

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
1010	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 spacecraft 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 find that there is now a shaded pixel to sum. Section A of row 2 is empty, so its sum is 0. Section B of this row, however, has a single shaded pixel. This pixel is in a column that has a value of 16. Since there are no other shaded pixels to sum up in this section, the sum of section B (row 2) is 16. Section C, being entirely unshaded, has a sum of 0. The sums for row 2, then, are 0 for SUM OF A, 16 for SUM OF B, and 0 for SUM OF C.

Let's try a more difficult row. Look at row 11. Section A of row 11 contains four shaded pixels. These pixels fall into columns that have values of 8, 4, 2, and 1. The SUM OF A for this row is equal to 8 + 4 + 2 + 1, or 15. All eight pixels in section B are shaded for row 11. The SUM OF B is the sum of all eight column values (128 + 64 +32 + 16 + 8 + 4 + 2 + 1). This sum is 255. Finally, section C of row 11 has only its first three pixels shaded. These pixels have values of 128, 64, and 32. The SUM OF C = 128 + 64 + 32 = 224.

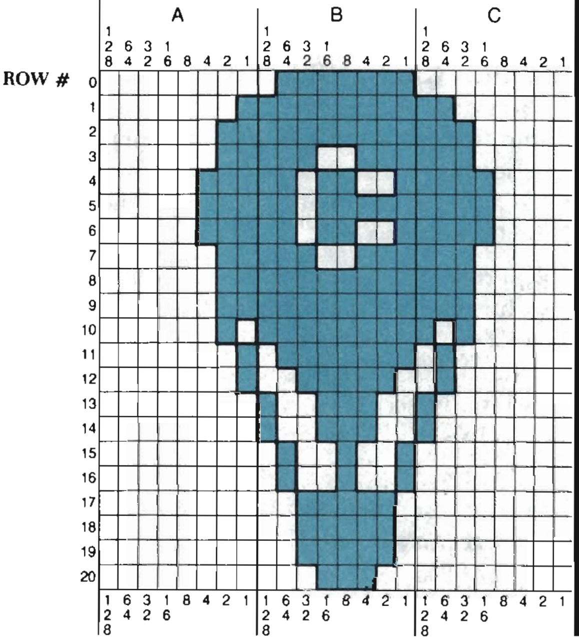
This straightforward and easy data collection method is used for each sum of each row, down through the grid. *A blank section always has a sum of zero, and an entirely shaded section always has a sum of 255.*

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

(TOP)



DATA STATEMENTS

BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
	DATA	0	127	0
	DATA	1	255	192
	DATA	3	255	224
	DATA	3	231	224
	DATA	7	217	240
	DATA	7	223	240
	DATA	7	217	240
	DATA	3	231	224
	DATA	3	255	224
	DATA	3	255	224
	DATA	2	255	160
	DATA	1	127	64
	DATA	1	62	64
	DATA	0	156	128
	DATA	0	156	128
	DATA	0	73	0
	DATA	0	73	0
	DATA	0	62	0
	DATA	0	62	0
	DATA	0	62	0
	DATA	0	28	0

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/retrieve 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/Retrieve 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

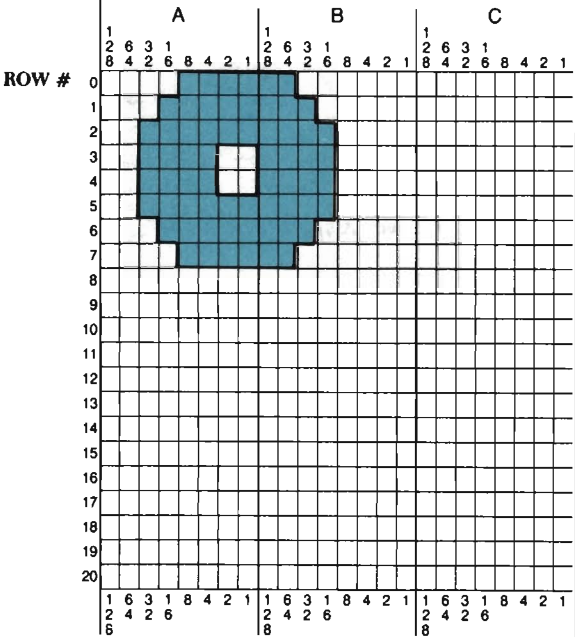
Once a sprite is defined, it must be "turned on." This feature turns on the appearance of a sprite. Note that a sprite must be placed on the screen (see "Place Sprite At X,Y") in order to see it once it is on. Thus, there are three steps to viewing a sprite: defining it, turning it on, and placing it 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.

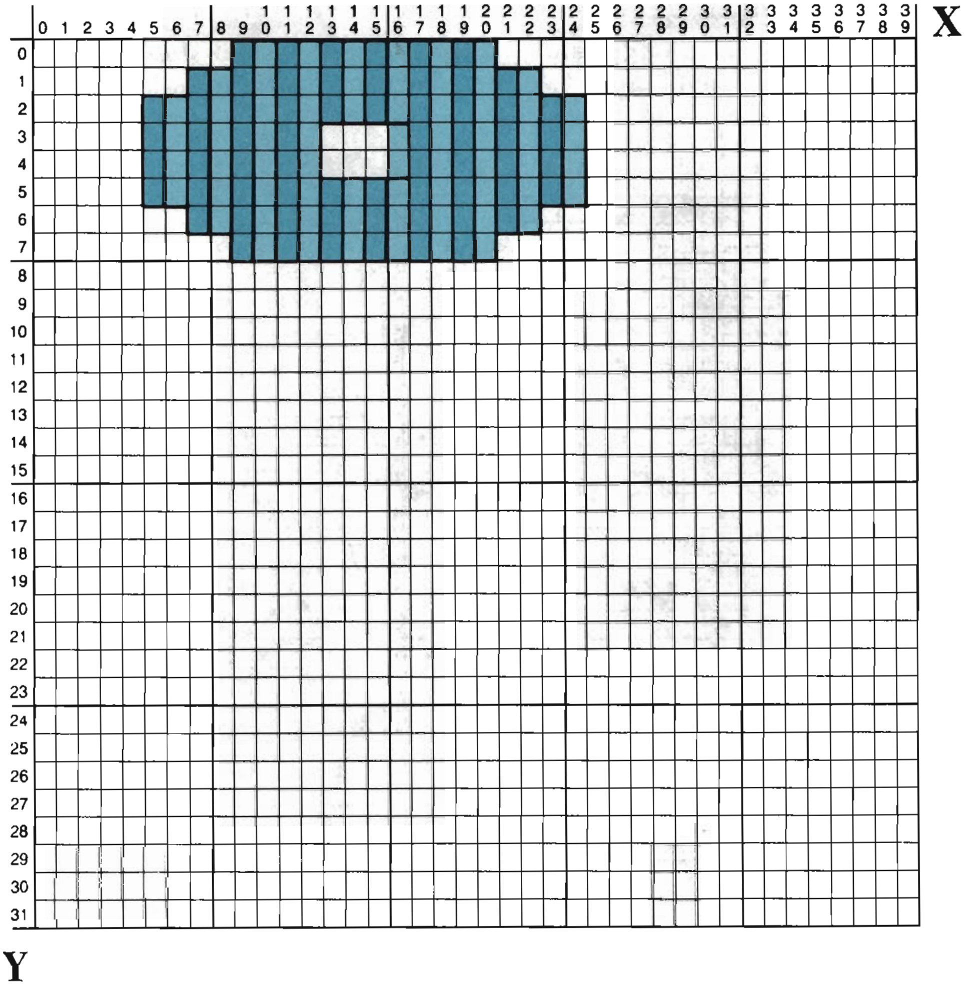
SPRITE DESIGN GRID

(TOP)



DATA STATEMENTS

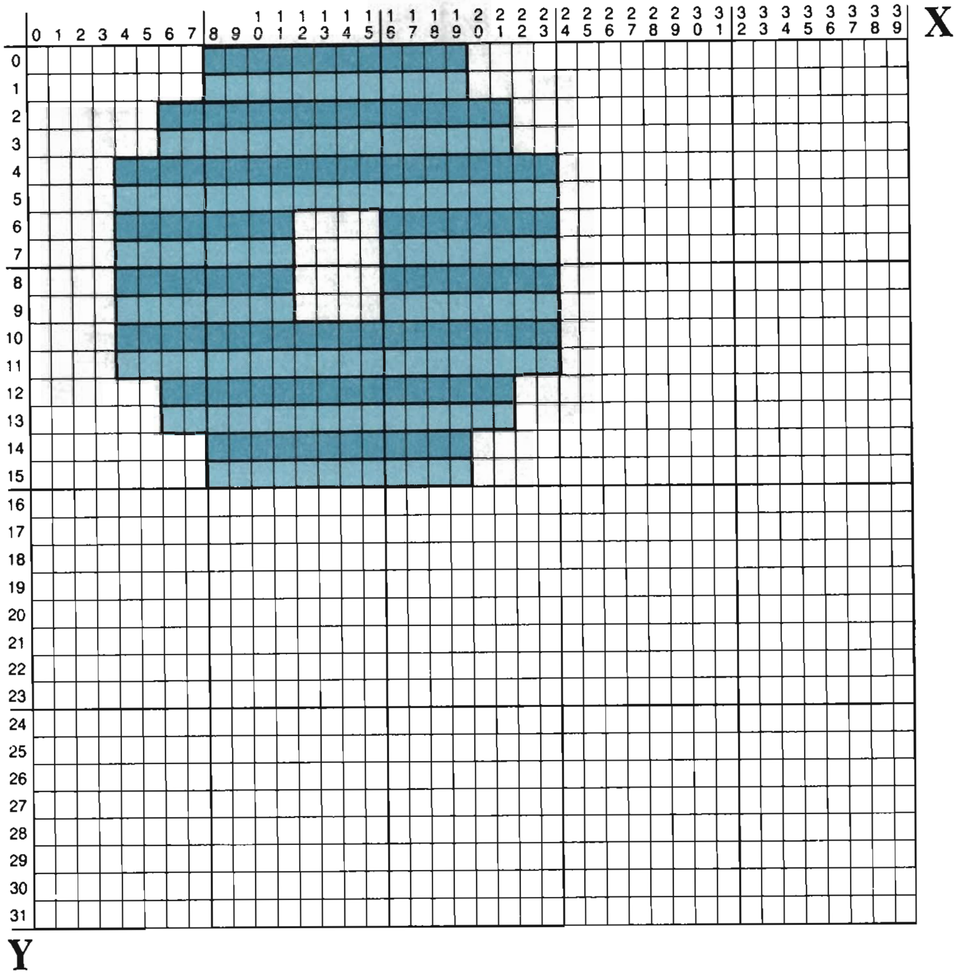
BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C



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:



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 spacecraft sprite, then the spacecraft will appear in front of any other shapes in the picture. 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 values. 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.

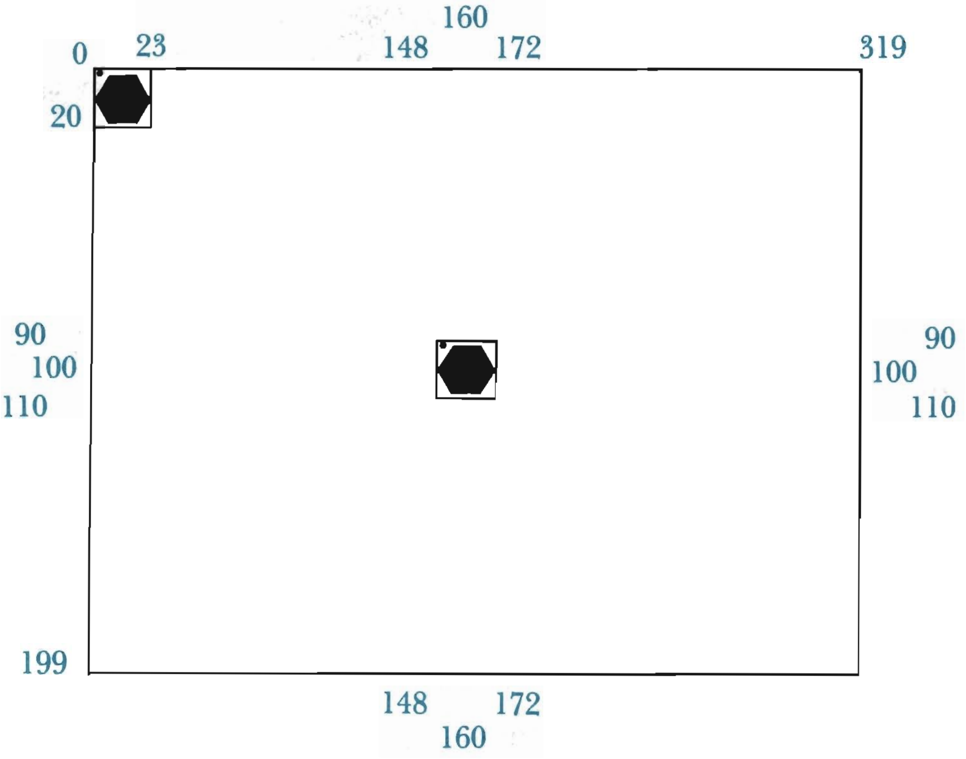
Place Sprite At X,Y

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.

You place a sprite by specifying where the sprite's *origin* should be positioned on the screen. An origin is simply a point of reference. The screen's origin, as you know, is always the pixel at 0,0. A sprite's origin is always the pixel represented by the top left square on the Sprite Design Grid. Recall that data statements will be entered describing all 504 squares on this grid. If you specify where the sprite's origin square should be placed (even if its not actually a part of the sprite), the computer can easily position the rest of the squares accordingly.

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.

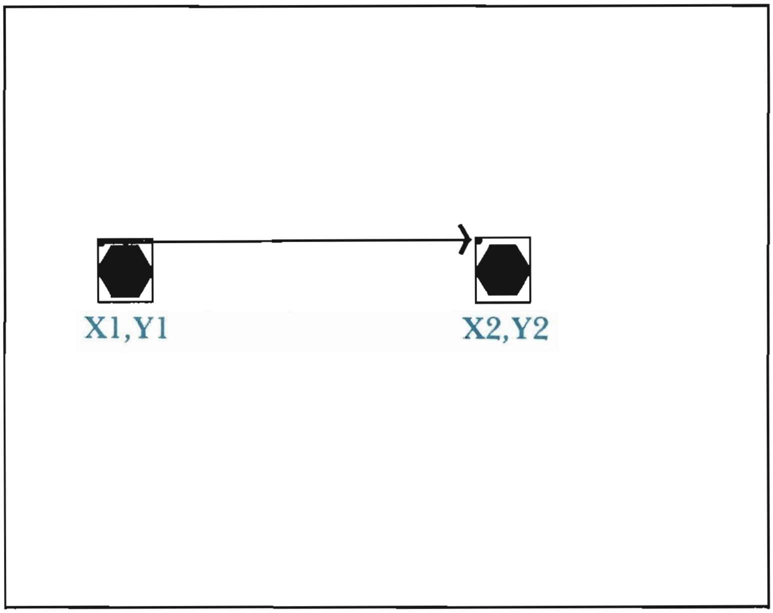
When the origin is re-positioned, the image of the sprite is moved accordingly. Following is an example of two sprites placed on the screen. The boxes around each represent the "invisible" grid the sprite was drawn on. The small dot in each box represents the origin square. The sprite in the top, left-hand 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 sprite could start where X1=40 and Y1=80, and travel across to where X2=200 and Y2=80. 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 Spacecraft Sprite

You are going to create the spacecraft shown earlier, and then experiment with it using the various sprite features we discussed. Start by typing the "setup" portion of your program as follows:

```
2ØØØ GOSUB 1Ø: C=11: GOSUB 3Ø
2Ø1Ø POKE 5328Ø,C
6ØØØ GET A$
6Ø1Ø IF A4 = "←" THEN GOSUB 2Ø: END
6Ø2Ø GOTO 6ØØØ
```

Next, type in the data statements that define the spacecraft:

```
1ØØ8 DATA "CRAFT1",31,Ø
1Ø1Ø DATA   Ø,  Ø,  Ø
1Ø12 DATA   Ø,  Ø,  Ø
1Ø14 DATA   Ø, 16,  Ø
1Ø16 DATA   Ø, 56,  Ø
1Ø18 DATA   Ø, 56,  Ø
1Ø2Ø DATA   Ø, 56,  Ø
1Ø22 DATA   Ø, 56,  Ø
1Ø24 DATA   Ø, 56,  Ø
1Ø26 DATA   1,187,  Ø
1Ø28 DATA   3,255,128
1Ø3Ø DATA   3,255,128
1Ø32 DATA  15,255,224
1Ø34 DATA 252,124,126
1Ø36 DATA 252, 56,126
1Ø38 DATA 128, 56,  2
1Ø4Ø DATA 128, 56,  2
1Ø42 DATA   Ø, 56,  Ø
1Ø44 DATA   Ø, 56,  Ø
1Ø46 DATA   Ø,124,  Ø
1Ø48 DATA   Ø,146,  Ø
1Ø5Ø DATA   Ø,146,  Ø
1Ø52 DATA   .,  .,  .
```

Notice that the sprite description is placed in the Shape Library. Your shape DATA statements and your sprite DATA statements can and should be grouped together in this location.

The sprite is named "CRAFT1" in line 1008. This name is necessary as a search parameter for retrieving the sprite later. The sprite is initially retrieved with Tool 800 and placed in the P% array. This is only a temporary arrangement, and the sprite description is later moved to an appropriate memory location for sprite data.

Recall that the RETRIEVE A SHAPE tool searches for a name, followed by a count of points, followed by a count of lines. Each point count represents two data items (X and Y), and each line count represents two data items ("from" and "to"). This tool will expect a count of points and lines, even when retrieving a sprite.

In order to have all 63 sprite data values read into P%, you must enter the count of points as 31. (Anything less than 31 would read in too few data values.) If the count of points is entered as 31, then thirty-two pairs of data items will be read into P%. Thirty-two pairs (that is, 64 data items) is enough to get the entire sprite description read into P%. The only problem is that an OUT OF DATA error will occur if only 63 data items are present when the computer expects 64.

We solve this problem through the use of a "dummy value," which is read into P% along with the rest of the sprite data. However, once it is read into P%, it is promptly forgotten and never used again. We have chosen a period (".") to be our dummy value. We call it a "value" because a period is equal to the value zero. You will find this first dummy value tacked to the end of our sprite data, at the beginning of line 1052.

It doesn't matter if any data is read into L%. Unfortunately, the lowest count of lines that we can give is 0. This is a zero-based count, which means we are saying we have one line. A line is comprised of two data items ("from" and "to"), so two more dummy values must be tacked onto the sprite's data. This accounts for the two other periods in line 1052.

Let's briefly recap this.

Line 1008 begins the sprite description, starting with the sprite's name ("CRAFT1"). Next, the point count of 31 will cause 64 data values to be read into P%. The line count of 0 will cause 2 data values to be read into L%.

Lines 1010 through 1050 give the data values that describe the spacecraft sprite. The data values were taken directly from the Sprite Design Grid. The data values of row 0 on the grid are entered first (line 1010), the data values of row 1 on the grid second (line 1012), and so on, until all data on the grid has been entered. Three dummy values (line 1052) ensure that an OUT OF DATA error will not occur due to an odd number of sprite data values, and the lack of any data to place in L%.

You should always make sure the count of points and count of lines are 31 and 0, respectively, when describing a sprite. The data values that follow should be a row-by-row replica of the data values written on the Sprite Design Grid. The order in which you enter the values of each row is important: SUM OF A, SUM OF B, and then SUM OF C.

Finally, you must end the sprite's description with three dummy values. We used a period (which is equivalent to 0) for each dummy value because it stands out in the program. If an OUT OF DATA error were to occur, you could easily check the sprite's last three data values to see if they are all periods.

The RETRIEVE A SPRITE subroutine is given below. Type it into your program.

```
810 REM::::::RETRIEVE A SPRITE
811 GOSUB 8ØØ
812 FOR I = Ø TO 31
813 POKE 16384 + 64*SP + I*2,P%(I,Ø)
814 POKE 16385 + 64*SP + I*2,P%(I,1)
815 NEXT I
816 POKE 18424+SP,SP
817 RETURN
```

The next several sprite subroutines control the appearance of a sprite. Subroutines 180 and 190 control whether a sprite is "on" or "off" in memory. Subroutines 200 and 210 determine whether the sprite is displayed at its normal width or at an expanded width. Subroutines 220 and 230 control whether the sprite will be displayed at its normal height or at an expanded height. Add these tools to your program now.

```
18Ø REM::::::TURN ON SPRITE SP
181 POKE 53269,PEEK(53269)OR 2↑SP
182 RETURN
19Ø REM::::::TURN OFF SPRITE SP
191 POKE 53269,PEEK(53269)AND(255-2↑SP)
192 RETURN
2ØØ REM::::::X EXPAND SPRITE SP
2Ø1 POKE 53277,PEEK(53277)OR 2↑SP
2Ø2 RETURN
21Ø REM::::::X UNEXPAND SPRITE SP
211 POKE 53277,PEEK(53277)AND(255-2↑SP)
212 RETURN
22Ø REM::::::Y EXPAND SPRITE SP
221 POKE 53271,PEEK(53271)OR 2↑SP
222 RETURN
23Ø REM::::::Y UNEXPAND SPRITE SP
231 POKE 53271,PEEK(53271)AND(255-2↑SP)
232 RETURN
```

Next come the tools that determine shape versus sprite priority. Tool 240 causes a specified sprite to appear in front of all foreground shapes. Tool 250 causes all shapes to appear in front of a specified sprite. Type these tools now.

```
24Ø REM::::::SP PRIORITY OVER SHAPE
241 POKE 53275,PEEK(53275)AND(255-2↑SP)
```

```
242 RETURN
25Ø REM::::::SHAPE PRIORITY OVER SP
251 POKE 53275,PEEK(53275)OR 2↑SP
252 RETURN
```

The last tools to enter control sprite color and sprite placement. Again, any sprite that is not assigned a specific color will be painted black. Any sprite that is not positioned on the screen will not be visible when turned on. Add these tools to your tool kit.

```
26Ø REM::::::SET SP TO COLOR C
261 POKE 53287+SP,C
262 RETURN
27Ø REM::::::PLACE SP AT X,Y
271 XX=X+24: YY=Y+5Ø: Z%=XX/256
272 V=XX-Z% * 256: W=53248+SP*2
273 WW=53264
274 PR=ABS((PEEK(WW)AND2↑SP)<>Ø)
275 VV=PEEK(WW)AND(255-2↑SP)OR(2↑SP*Z%)
276 IF PR<>Z% THEN GOSUB 19Ø
277 POKE W,V: POKE WW,VV: GOSUB 18Ø
278 POKE 53249+SP*2,YY
279 RETURN
```

That completes the tools for now. The main routine lines which implement these tools are given below. We have included many REM ("remark") statements in this main routine to help you follow what is going on. Type these lines now.

```
2Ø2Ø SP=Ø: REM  HIGH SP PRIORITY
2Ø3Ø SE$="CRAFT1": GOSUB 81Ø
2Ø4Ø GOSUB 18Ø: REM  TURN ON
2Ø5Ø GOSUB 2ØØ: REM  WIDEN
2Ø6Ø GOSUB 22Ø: REM  HEIGHTEN
2Ø7Ø GOSUB 24Ø: REM  SP OVER SHAPES
2Ø8Ø C=7: GOSUB 26Ø: REM  COLOR
2Ø9Ø X=147:Y=89: GOSUB 27Ø:REM PLACE
```

We will discuss these lines momentarily. First, RUN the program to examine the fruits of your effort—a sprite spacecraft.

The sprite should be approximately centered on the screen. It will be yellow in color, against a blackish background screen. This spacecraft should resemble in every detail the one sketched earlier on our Sprite Design Grid, except that the sprite on your screen will be twice as high and twice as wide as that on the grid.

If a few pieces of the sprite have been placed incorrectly, you have probably entered one or more incorrect data values. An OUT OF DATA error, found only by returning to text mode, would also suggest a typing mistake in the data statements. Look for errors in the subroutines if any other problems crop up. Do not continue until your program runs properly.

Return to text mode. Notice that a distorted version of your sprite remains on the screen, even after you've returned to the program listing. This is because the sprite was not "turned off" within the program by a GOSUB 190 statement. Sprites must be turned off to be completely erased from the screen. Move the cursor to an empty line and type: GOSUB 190. Press **RETURN**. The sprite indicated by SP's current value (SP=0) will be turned off. This, of course, is the spacecraft sprite.

List lines 2020 through 2090 on your screen. There are five steps that you must complete any time you wish to see a sprite on the screen. They are:

(1) The sprite must be assigned a sprite pointer number (SP=?). This determines its priority in relation to all other sprites that are or might be placed in the picture.

(2) The sprite's description must be retrieved (SE$="?": GOSUB 810).

(3) The sprite must be turned on (GOSUB 180).

(4) The sprite must be assigned a color (C=?: GOSUB 260).

(5) The sprite must be placed on the screen (X=?: Y=?: GOSUB 270).

Step (1) is accomplished with program line 2020. Setting the value of SP does two things. First, it determines the priority of the next sprite retrieved (if any). This is the sprite's priority in relation to all the other sprites. When two sprites are placed at the same screen location, the sprite with the lowest sprite number (SP) appears in front of the other.

Second, the current value of SP determines which sprite will be affected by the various sprite features that are called upon. The sprite that is being "pointed at" by SP will be the sprite turned on/off, painted, heightened, etc, as each sprite tool is called.

Step (2) is accomplished by line 2030 in your program. The description retrieved here determines what the current sprite will look like. The current sprite is the one being pointed at by SP. You can retrieve the same description for as many of the eight sprites as you desire. You will need to remember, though, to change SP whenever necessary. For example, suppose we left SP set to 0, and then retrieved a second sprite description that looked like a planet. That would cause the computer to erase our spacecraft description for Sprite #0 and replace it with a planet description. We would not have created a second sprite.

You need to turn on a sprite (GOSUB 180) after retrieving it. This step causes the computer to display the sprite at its current position. Program line 2040 turns on your spacecraft sprite in memory.

The color of the spacecraft is assigned in line 2080. This color can be set at any time, but should most often be done before positioning the sprite on the screen.

Finally, step (5) is executed by line 2090 in your program. This places the sprite on the visible screen. The sprite's origin will be placed at the X,Y location given by X's current value and Y's current value. We placed the origin at 147,89 (approximately centering the sprite).

The other sprite features are not essential to viewing a sprite on the screen, but they are nonetheless worthwhile. Program line 2050 calls the X EXPAND subroutine. This produces a sprite that is twice its normal (defined) width. Program line 2060 calls the Y EXPAND subroutine. This produces a sprite that is twice its normal height.

The last feature called by this program is the SP PRIORITY OVER SHAPE tool (GOSUB 240). This tool is necessary in the event a sprite is placed at the same location as a foreground shape. Calling this tool ensures that the sprite has priority over the shape, and thus appears in front of it. You can give all sprites priority over shapes, some sprites priority over shapes, or no sprites priority over shapes. This is all determined by the sprite being pointed at, and the tool that is called.

Let's add a foreground shape to the picture to see this priority tool in action. Add the following to your program:

```
1Ø54 DATA "PLANET",2,Ø
1Ø56 DATA  5Ø,-4,5Ø, 4,48, Ø
1Ø58 DATA   Ø, 1
21ØØ REM:::::::DRAW PLANET
211Ø SE$="PLANET": GOSUB 8ØØ
212Ø C=13: GOSUB 11Ø
213Ø YT=18Ø: XT=55: GOSUB 14Ø
214Ø GOSUB 9Ø
215Ø FOR L = 1 TO 36
216Ø YT=-YT: XT=-XT: GOSUB 14Ø
217Ø RO=1Ø: GOSUB 16Ø
218Ø YT=-YT: XT=-XT: GOSUB 14Ø
219Ø GOSUB 9Ø: NEXT L
22ØØ REM:::::::PAINT PLANET
221Ø PP=2: GOSUB 6Ø
```

Modify line 2090 so that the sprite will be placed in the same location as the planet:

```
2Ø9Ø X=3Ø:Y=155: GOSUB 27Ø:REM PLACE
```

RUN the program. The sprite is almost instantly placed in the screen's lower left-hand corner. Next, a green planet should gradually be added to the picture. This planet is formed by rotating and drawing a small line thirty-six different times. (The combination of rotating and translating is a bit time-consuming. Please be patient.) The planet will be painted after it has been completely drawn.

The sprite will appear as if it is in front of the planet. This is because it has been given priority over all foreground shapes it encounters.

Sprites and shapes are very different compositions, and they are treated differently by the computer. The PAINT A SHAPE routine painted the planet as if the sprite were not even there. Notice that even the color blocks went undisturbed by the appearance of two foreground colors in the same area.

Return to text mode and change line 2090 back to:

```
2Ø9Ø X=147:Y=89: GOSUB 27Ø:REM PLACE
```

The program lines below use the same sprite DATA statements to define a new sprite, Sprite #1. Add these lines to your program.

```
2300 REM::::::PLACE SECOND SPRITE
2310 SP=1: REM  ASSIGN SP #
2320 SE$="CRAFT1": GOSUB 810
2330 GOSUB 180
2340 C=1: GOSUB 260
2350 X=275: Y=50: GOSUB 270
```

So that the graphics screen is not erased by a GOSUB 30, and so that the planet is not unnecessarily re-drawn, modify line 2000 and add line 2105 as follows:

```
2000 GOSUB 10: C=11: REM  GOSUB 30
2105 GOTO 2300
```

RUN the program. A second, smaller spacecraft will be placed in the top right section of the screen. It is smaller only because it is displayed exactly as defined in the DATA statements (i.e., it is not expanded in height or width).

Return to text mode again, and list lines 2300 through 2350 on the screen.

Line 2310 sets SP to a new value of 1. This means that any further sprite subroutines that are called will effect Sprite #1, and not Sprite #0. In addition, it means that any sprite description that is retrieved will be used to describe Sprite #1. Program line 2320 retrieves the description intended for this sprite. It is the same description that was retrieved for Sprite #0.

TOOL 810 :::::: RETRIEVE A SPRITE

```
810 REM::::::RETRIEVE A SPRITE
811 GOSUB 800
812 FOR I = 0 TO 31
813 POKE 16384 + 64*SP + I*2,P%(I,0)
814 POKE 16385 + 64*SP + I*2,P%(I,1)
815 NEXT I
816 POKE 18424+SP,SP
817 RETURN
```

What It Does: This tool retrieves a sprite description. The description named by SE$ (SE$="?") will be the one retrieved. It will be read into P% and will define the sprite you have specified by SP's current value.

Up to eight sprites can be defined in memory. Each one must be given its own Sprite Pointer number. The priority of one sprite over another sprite is determined by the sprite numbers. The lower the sprite number, the greater screen priority the sprite will have. Sprite #0 will appear in front of all other sprites it encounters. Sprite #3 will appear in front of Sprites 4, 5, 6, and 7. Sprite #7 has no screen priority, and will be placed behind any other sprite it meets.

Example Use: You will need to take the following steps to define a sprite in memory:

(1) Draw the sprite on the "Sprite Design Grid."

(2) Add up the three sums (A,B,C) for the shaded pixels in each row of the grid.

(3) The sprite's name and point/line counts should be typed as a DATA statement in the Shape Library. The name may be any construction you desire, but must be within quotes. The count of points should immediately follow the name, and should *always* be 31. Next, the count of lines should be given as 0.

(4) There will be three data values for each of the 21 rows on the Sprite Design Grid. These data values should be typed into the Shape Library (beneath the line giving the sprite's name) as DATA statements. They should be typed in the same order as they are listed on the grid.

(5) The last DATA statement from the Sprite Design Grid should be followed by a "dummy" DATA statement. This DATA statement will have three dummy values, and can simply contain:DATA.,.,.

(6) In the main routine, set SP to the sprite number you wish to assign the sprite you are retrieving (SP=?). In addition, set SE$ to the "NAME" of the sprite description to retrieve.

(7) Call the RETRIEVE A SPRITE tool (GOSUB 810).

Technical Description: Sprite data is typed into the program much as you type shape data into the program. This setup allows us to search for sprites at the beginning of the DATA statements, without doing any special processing in order to skip over shape data.

Line 811 calls the RETRIEVE A SHAPE tool to retrieve the sprite data, since that tool already searches for a specified section of data. Doing this will place the sprite data in the P% array. Lines 812 through 815 then take the sprite data in the P% array and store it in a memory block set aside for that specific sprite.

Line 816 sets up a pointer for the sprite. This pointer "points" to the data that describes the sprite just read into memory.

Each sprite is given a sixty-four byte block of memory, located at the beginning of Bank 1. Sprite 0 gets Block 0, Sprite 1 gets Block 1, etc. This number corresponds to the Sprite Pointer used in line 816.

TOOL 180 :::::: TURN ON SPRITE SP

```
180 REM::::::TURN ON SPRITE SP
181 POKE 53269,PEEK(53269)OR 2↑SP
182 RETURN
```

What It Does: This tool turns on a sprite's description in memory. The sprite that is turned on will be the sprite specified by SP's current value. A sprite must be retrieved (Tool 810), turned on (Tool 180), and positioned (Tool 270) in order to be seen on the screen.

Example Use: SP must be set to the sprite number to turn on (SP=?). A GOSUB 180 statement will then turn on the sprite. If the sprite has been placed on the screen, turning it on will make it appear.

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 180 to 252 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 260 controls the color for each sprite. This is an example of a subroutine where each sprite has its own register. The subroutines from 180 to 252 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.

7	6	5	4	3	2	1	Ø

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 = Ø 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 180 to 252 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 190-192. Tool 190 will erase a sprite as easy as you turn one one. All that is needed is the sprite to erase (SP=?) and a GOSUB 190 statement. This tool is described in the following tool box.

TOOL 140::::::TURN OFF SPRITE

```
190 REM::::::TURN OFF SPRITE SP
191 POKE 53269,PEEK(53269)AND(255-2↑SP)
192 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 190 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 180 turns on a sprite, and tool 190 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.

Add the following program lines to your program:

```
24ØØ SP=Ø
241Ø GOSUB 19Ø: REM  TURN OFF
242Ø FOR I = 1 TO 5Ø: NEXT I
243Ø GOSUB 18Ø: REM  TURN ON
244Ø GOTO 241Ø
```

These lines create an endless loop that will turn Sprite #0 off, count to 50, and then turn Sprite #0 back on again.

RUN the program. After both sprites are placed on the screen, Sprite #0 will begin flashing. It will flash at a rapid rate, and will continue to do so until you press **RUN/STOP**. Press **RUN/STOP** and tap **RESTORE** to return to text mode.

An important thing to notice is that sprites can be controlled almost entirely by GOSUB statements. Very few variables need to be set. Your biggest chore is remembering which tool does what. All of the tools are listed on a cut-out card at the back of this book, so this task does not amount to much.

Delete lines 2400, 2410, 2420, 2430 and 2440.

Sprite #0 is currently set to display at twice its defined height and width. Sprite #1 is set to display at the normal height and width. The next two program lines will change Sprite #0 so that only its height is expanded, and Sprite #1 so that its width is expanded.

Try these new lines out by entering them and running the program:

```
2Ø5Ø GOSUB 21Ø: REM  X UNEXP
2335 GOSUB 2ØØ: REM  X EXP
```

The yellow spacecraft should be appear tall and narrow. The white spacecraft should appear short and fat. The expand/unexpand tools are similar in effect to your SCALE A SHAPE tool. The major difference is that sprites can only be "scaled" up one size in height, and up one size in width. Also, a sprite cannot be "scaled" down to a size smaller than its original size as defined in the DATA statements.

Program line 2060, below, calls upon the Y UNEXPAND SPRITE to return Sprite #0 to its original height. Change this program line as shown, and run the program:

```
2Ø6Ø GOSUB 23Ø: REM  Y UNEXP
```

Each sprite's size is individually controlled. Tools 200 and 210 control the expansion or contraction of a sprite's width. Tools 220 and 230 control the expansion/contraction of a sprite's height. The sprite whose number is represented by SP's current value is the sprite whose size will be affected by these four tools.

TOOL 200:::::::X EXPAND SPRITE

```
2ØØ REM:::::::X EXPAND SPRITE SP
2Ø1 POKE 53277,PEEK(53277)OR 2↑SP
2Ø2 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 200 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.

TOOL 210:::::::X UNEXPAND SPRITE

```
21Ø REM:::::::X UNEXPAND SPRITE SP
211 POKE 53277,PEEK(53277)AND(255-2↑SP)
212 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 200.

Example Use: To use this tool, you need to set SP equal to the sprite number to expand, and then insert a GOSUB 210 statement.

Technical Description: This subroutine is the opposite of the X expand subroutine (Tool 200). It uses Form #2 of the statement 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.

TOOL 220:::::::Y EXPAND SPRITE

```
22Ø REM:::::::Y EXPAND SPRITE SP
221 POKE 53271,PEEK(53271)OR 2↑SP
222 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 220 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.

TOOL 230::::::Y UNEXPAND SPRITE SP

```
23Ø REM::::::Y UNEXPAND SPRITE SP
231 POKE 53271,PEEK(53271)AND(255-2↑SP)
232 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 230 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.

You have already seen what happens when a sprite has priority over shapes. The yellow spacecraft had such priority, and was placed in front of the green planet. What would happen if the priority were reversed? To find out, delete line 2105 so that the planet will again be drawn, and then modify the program as follows:

```
2ØØØ GOSUB 1Ø: C=11: GOSUB 3Ø
2Ø7Ø GOSUB 25Ø: REM  SHAPE OVER SP
2Ø9Ø X=95:Y=175: GOSUB 27Ø:REM PLACE
```

Line 2070 gives all foreground shapes (in this case, the planet) priority over Sprite #0 (the yellow spacecraft). Line 2090 moves the yellow spacecraft so that is placed partially on the planet. RUN the program.

Watch carefully as the planet is painted. The PAINT A SHAPE routine will paint right over top of the sprite. This is because the sprite has a lower screen priority than the foreground pixels that are painted.

You should decide the priority of your sprites ahead of time. Decide how the sprites should stack up on the screen (sprite versus sprite priority), and then assign each an appropriate sprite number. Next, decide which sprites should have priority over shapes, and which sprites should not have priority over shapes. A GOSUB 240 statement or a GOSUB 250 statement ought to be given for each individual sprite you create.

TOOL 240:::::::SPRITE PRIORITY OVER SHAPE

```
24Ø REM:::::::SP PRIORITY OVER SHAPE
241 POKE 53275,PEEK(53275)AND(255-2↑SP)
242 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 240 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.

TOOL 250:::::::SHAPE PRIORITY OVER SPRITE

```
25Ø REM:::::::SHAPE PRIORITY OVER SP
251 POKE 53275,PEEK(53275)OR 2↑SP
252 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 250statement 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.

Next, we can experiment with the various sprite colors. Modify program lines 2080 and 2340 as follows:

```
2080 C=0: GOSUB 260
2340 C=8: GOSUB 260
```

Re-insert program line 2105, and modify 2000 so that the computer is sent past the planet-drawing portion of your program:

```
2000 GOSUB 10: C=11: REM  GOSUB 30
2105 GOTO 2300
```

RUN the program. The first sprite should be displayed in black. A black sprite on a dark-grey background shows up well. Sprite #1 will be painted orange. This is because C was set to 8 before the SET SP TO COLOR C tool was called.

Return to text mode and experiment with your own sprite colors. The tool box for this sprite feature is given below.

TOOL 260::::::SET SPRITE TO COLOR C

```
260 REM::::::SET SP TO COLOR C
261 POKE 53287+SP,C
262 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 260 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).

The final feature in your program is the PLACE SP AT X,Y. This tool will place a sprite's origin at the X,Y pixel specified by the current values of X and Y. There is no "clip a sprite" tool, so the origin cannot be placed at all offscreen locations. A sprite can, however, be placed partially or fully off the screen if you adhere to these X and Y ranges:

X placement must be greater than or equal to -24 and less than or equal to 487
Y placement must be greater than or equal to -50 and less than or equal to 205

The Sprite Design Grid is only twenty-four pixels wide, so this X range will allow any sized sprite to be placed completely off the screen to the right. However, there is only enough room (-1 to -24) off the screen's left side to hold an X

Unexpanded sprite. A sprite whose width has been expanded can never be fully placed off the left side of the screen.

The sprite design grid is only twenty-one pixels high, so any spite (expanded or not) can be placed off the top or off the bottom of the screen.

Try this out by changing the following program lines:

```
2Ø9Ø X=-23: Y=-5Ø: GOSUB 27Ø
235Ø X=-23: Y=5Ø: GOSUB 27Ø
```

RUN the program. Sprite #0 should not appear at all, and Sprite #1 should be only partially displayed. When you later begin animating your sprites (which you will do in the next section), the ability to start a sprite off the screen and then move it into the viewing area will be a good feature to have. Return to your program listing.

At this point we have covered all of the basic sprite features except MOVE SP FROM X1,Y1 TO X2,Y2. A section on this form of sprite animation follows the tool box below. Before going on to that section, take some time to experiment with the sprite tools you just learned. All tools are called with a GOSUB statement, and will affect the sprite indicated by SP's current value. The only other variables to set are SE$ (before calling Tool 810), C (before calling Tool 260), and X and Y (before calling Tool 270).

TOOL 270::::::PLACE SPRITE AT X,Y

```
27Ø REM::::::PLACE SP AT X,Y
271 XX=X+24: YY=Y+5Ø: Z%=XX/256
272 V=XX-Z% * 256: W=53248+SP*2
273 WW=53264
274 PR=ABS((PEEK(WW)AND2↑SP)<>Ø)
275 VV=PEEK(WW)AND(255-2↑SP)OR(2↑SP*Z%)
276 IF PR<>Z% THEN GOSUB 19Ø
277 POKE W,V: POKE WW,VV: GOSUB 18Ø
278 POKE 53249+SP*2,YY
279 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.

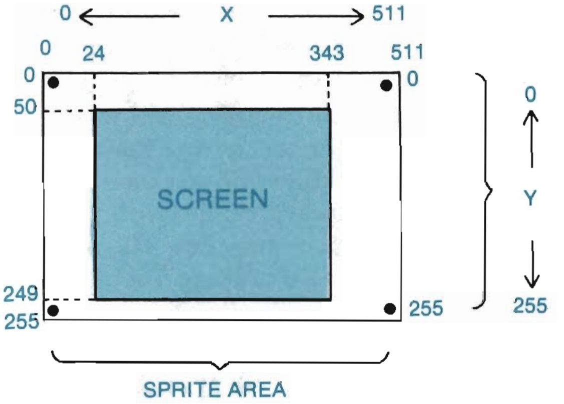
Example Use: You need to first set SP equal to number of the sprite that you wish to place on the screen. This would be followed by a main routine line in the form of:

X=#: Y=#: GOSUB 270

In this line, the first "#" should be replaced with an X position for the origin. This value must be between -24 and 487 (inclusive). The second

"#" should be replaced with a Y position for the origin. This value must be between -50 and 205 (inclusive). The sprite's origin is placed on (or off) the screen at this X,Y coordinate.

Technical Description: You know from the text how to position a sprite on the screen. We stated that the X position could range from -24 to 487, and the Y position could range from -50 to 205. Actually, a sprite has its own set of coordinates. The screen's 0,0 pixel is a sprite's 24,50 pixel. A sprite's 0,0 pixel is a screen's -24,-50 pixel. This can be diagrammed as follows:



In order to maintain a consistency between screen and sprite coordinates, the subroutine at 270 will accept screen coordinate values (X=?: Y=?) and will convert them to sprite coordinate values (X=X+24: Y=Y+50). Line 271 does this for you:

```
271 XX=X+24: YY=Y+5Ø: Z%=XX/256
```

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.

```
272 V=XX-Z% * 256: W=53248+SP*2
```

Line 272 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.

```
273 WW=53264
```

In line 273, WW is the memory location of the register which is shared by all the sprites.

```
274 PR=ABS((PEEK(WW)AND2↑SP)<>Ø)
275 V=PEEK(WW)AND(255-2↑SP)OR(2↑SP*Z%)
```

Line 274 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 275 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 -2↑SP)." This technique was explained in the "TURN ON SPRITE SP" and "TURN OFF SPRITE SP" tool boxes. The "OR (2↑SP*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 2↑SP by Z will result in the bit being set or not.

```
276 IF PR<>Z% THEN GOSUB 19Ø
```

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 190 will do this.

```
277 POKE W,V: POKE WW,VV: GOSUB 18Ø
```

Line 277 changes the X position and turns the sprite back on with a GOSUB 180.

```
278 POKE 53249+SP*2,YY
```

Finally, line 278 changes the Y position value to the new position.

Animating The Spacecraft Sprite

Now, the section you've been waiting for: sprite animation. This type of animation concerns moving a sprite on, around, and off the visible screen area. The sprite will remain visible at all times, while moving in a linear path from Point A to Point B.

You must first add the MOVE SPRITE subroutine to your tool kit. Type it as follows:

```
28Ø REM::MOVE SP FROM X1,Y1 TO X2,Y2
281 DX=X2-X1: DY=Y2-Y1
282 L=ABS(DX):IF ABS(DY)>L THEN L=ABS(DY)
283 IF L>Ø THEN XI=DX/L: YI=DY/L
284 X=X1: Y=Y1: SD=SD+ABS(SD=Ø)
285 FOR I = Ø TO L STEP SD
286 GOSUB 27Ø
287 X=X+XI*SD: Y=Y+YI*SD
288 NEXT I
289 RETURN
```

There is surprisingly little involved in the movement of a sprite. You need only set four variables (X1=?: Y1=?: X2=?: Y2=?), call the new tool (GOSUB 280), and the sprite will be moved accordingly. You can see how this is done by changing line 2090 and adding line 2095:

```
2Ø9Ø X1=5Ø: Y1=25: X2=285: Y2=25: SD=5
2Ø95 GOSUB 28Ø: GOTO 2Ø95
```

Check your typing, and then RUN the program. After a moment, you should see the black spacecraft travel across the screen, from left to right. Each time the spacecraft completes its journey across the screen, it momentarily disappears, and then reappears on the screen's left side again. The trip across the screen then repeats. No other sprites will appear on the screen.

The program is in an endless loop, so you must press **RUN/STOP** and tap **RESTORE** to break out of it. List lines 2090 and 2095 on your screen.

The spacecraft's movement starts with its origin placed at X1,Y1 (50,25). The movement continues in a straight path to the X2,Y2 point (285,25). It is the sprite's origin (upper left-hand grid square) that follows the straight line from X1,Y1 to X2,Y2. The remainder of the sprite is moved relative to this origin. A sprite can be moved horizontally, vertically, and diagonally.

The speed at which the sprite moves is determined by the value of the variable SD (SpeeD). This variable was set at the end of line 2090 in your program (SD=5). The number assigned to SD specifies how many pixels to skip over between each placement of the sprite as it travels from X1,Y1 to X2,Y2. To move a sprite, the computer simply places it on the screen, erases it, moves it over SD number of pixels, places it, erases it, moves it over SD number of pixels, and so on. This happens very fast, and continues until the destination point is reached.

The first thing line 2095 does is call the MOVE SPRITE subroutine. This tool will move the *last* sprite pointed to in the program along the linear path specified by the *last* X1,Y1 and X2,Y2 values given in the program.

The second part of the line 2095 (GOTO 2095) repeatedly moves the sprite across the screen. This GOTO statement continually sends the computer back to the beginning of line 2095, which calls the MOVE SPRITE tool, and in turn moves the sprite from X1,Y1 to X2,Y2.

Let's experiment with the path and speed of movement. The speed is currently set to 5. This means that five pixels will be skipped between each placement of the sprite. Change the program so that the speed is increased to 20. In addition, change the path of movement so that the sprite starts off the screen's left edge, moves horizontally across the screen, and moves off the screen's right edge. Type the following:

```
2Ø9Ø X1=-24: Y1=25: X2=32Ø: Y2=25: SD=2Ø
```

Before running this program, list line 2050. Make sure that it calls Tool 210 (X UNEXPAND SPRITE SP). RUN the program. The spacecraft should move onto the screen from the left side, move across the screen, and then move off the screen on the right side. Its movement will be jerky. This is due to the increased speed. The sprite is only being placed every twenty pixel locations instead of every five. This moves the sprite across the screen quicker, but the extra space between each sprite placement is more easily detected by the eye.

Return to the program (**RUN/STOP-RESTORE**) and modify it as follows:

```
2Ø5Ø GOSUB 2ØØ: REM  X EXP
2Ø9Ø X1=-24: Y1=25: X2=32Ø: Y2=199: SD=2
```

RUN the program and watch for any new or unexpected events.

The sprite, which was expanded in width, does not start its journey completely off the left side of the screen. This is because the sprite now takes up forty-eight pixel columns in width, and we can only move twenty-four of those pixels (-1 to -24) off the screen.

The movement of the sprite will be slower than before, and jagged. The slower movement is due to a decrease in the speed variable. The jagged movement is due to the requested path of movement. The sprite's origin is moving along the *straightest* path from X1,Y1 to X2,Y2. This path is not a truly straight line, since there is no straight path from -24,25 to 320,199. Always consider where a plotted line would appear between the X1,Y1 and X2,Y2 path. This imagined line is the exact path the sprite's origin will take.

Try moving the sprite along a vertical path:

```
2Ø9Ø X1=159: Y1=2ØØ: X2=159: Y2=-3Ø: SD=3
```

Here we will be moving the sprite from bottom to top. RUN the program, and the sprite will move up through the screen and off the top edge.

Below is the tool box for the MOVE SPRITE subroutine.

TOOL 280:::::::MOVE SPRITE FROM X1,Y1 TO X2,Y2

```
28Ø REM::MOVE SP FROM X1,Y1 TO X2,Y2
281 DX=X2-X1: DY=Y2-Y1
282 L=ABS(DX):IF ABS(DY)>L THEN L=ABS(DY)
283 IF L>Ø THEN XI=DX/L: YI=DY/L
284 X=X1: Y=Y1: SD=SD+ABS(SD=Ø)
285 FOR I = Ø TO L STEP SD
286 GOSUB 27Ø
287 X=X+XI*SD: Y=Y+YI*SD
288 NEXT I
289 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.

(3) Enter a number for the speed (SD=?). The higher the number, the faster the speed.

(4) Follow all the above with a GOSUB 280.

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. 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 284, 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.

```
287 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 its travels, thus speeding things up.

```
285 FOR I = Ø 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.

Summary

We hope you've enjoyed this chapter introducing you to sprites. Although there is much more to learn in this area of computer graphics (some of which is covered in the final chapter), you can now create, place and move up to eight sprites within your picture.

Don't look upon sprites as a means to animation alone. There are many other equally good reasons to create a sprite figure rather than a foreground shape figure. To name just a few:

—sprites can be placed on the screen faster than foreground shapes;
—sprites do not disturb the color blocks they are placed in;
—sprites can change colors faster than foreground shapes;
—sprites can "flash" (on/off loop); and
—sprite priority can be used to create interesting visual effects.

So you see, there is much more to sprite graphics than the ability to move sprites around the screen—although movement is a significant feature.

Below is a brief review of the steps required in making a sprite. These steps are followed with a short description of the various sprite features, and then this chapter's exercise. Wait until the exercise is complete before saving the program under "CHAPTER 7".

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 a DATA statement in the Shape Library that names the sprite, gives a point count of 31, and gives a line count of 0. The name may be any construction, but must be within quotes. An example data statement would be: 1008 DATA "CRAFT1",31,0

(4) Following the above DATA statement, enter the twenty-one rows of data from the Sprite Design Grid. These should be entered as DATA statements, in the same order as listed on the grid.

(5) Assign the sprite a sprite pointer number in the main routine. This is done by setting the variable SP to a value between 0 and 7. One sprite's priority over other sprites is determined by its pointer number. The lower a sprite's number, the greater sprite priority it has.

(6) Select the sprite's color by setting C to a color code between 0 and 15, and then calling Tool 260.

(7) Retrieve the sprite with a GOSUB 810 statement.

(8) Type in the desired GOSUB statements to call various sprite features. (A complete list of all sprite features is given below.)

(9) Place the sprite by setting X and Y to the desired X,Y placement of the sprite's origin. Follow this with a GOSUB 270.

Summary of Sprite Features

Here we list the various sprite features and give a brief explanation of each. For a more thorough discussion of any sprite feature, refer to the beginning of this chapter.

SP=0: SE$="SPRITE NAME": GOSUB 810

This line first sets the sprite's pointer number (SP=0). The sprite number determines sprite priority. Next, the search variable (SE$) is set to the sprite's name, as given in its defining DATA statements. Finally, GOSUB 810 calls the RETRIEVE SPRITE tool.

GOSUB 180
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 190
This tool "turns off" a sprite so that it disappears from the screen.

GOSUB 200
The sprite's width is doubled in size with Tool 200.

GOSUB 210
This tool is used with a sprite whose width has been doubled with tool 200. This tool will return the sprite to its normal width.

GOSUB 220
The sprite's height is doubled in size with Tool 220.

GOSUB 230
This tool is used with a sprite whose height has been doubled with tool 220 This tool will return the sprite to its normal height.

GOSUB 240
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 250
Tool 250 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 260
Tool 260 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 270
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 270 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 280
Tool 280 uses the given X1,Y1 and X2,Y2 coordinate values to move the sprite in a straight line.

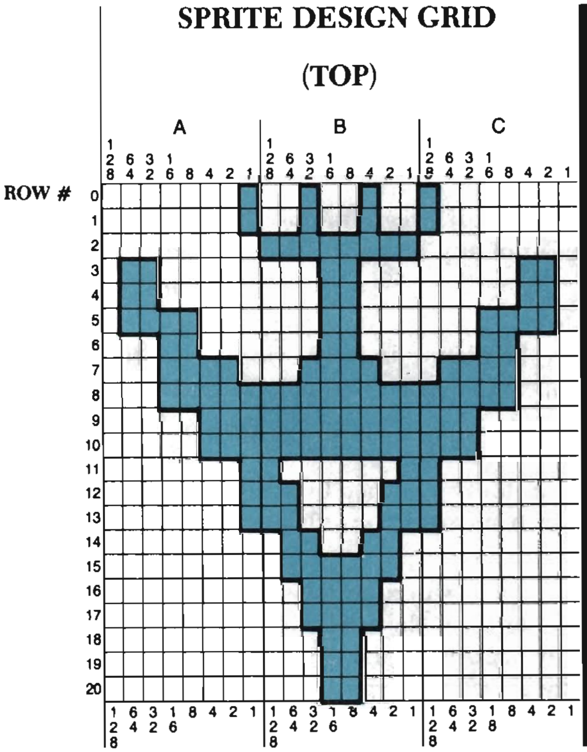
Exercise

This chapter's challenge is to figure out the necessary DATA statements for the spacecraft designed below. Write the data values on the grid itself, if you like, or on a separate piece of paper. Then, enter the data beginning with:

1060 DATA "CRAFT2",31,0

Try placing and moving both CRAFT1 and CRAFT2 on the screen. The required DATA statements, and a sample program, are given under "Solution."

> IMPORTANT: SAVE your program when you have entered a solution to CRAFT2 that works properly. Use the filename "CHAPTER 7".



DATA STATEMENTS

BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			

Solution

```
1060 DATA "CRAFT2",31,0
1062 DATA   1, 36,128
1064 DATA   1, 36,128
1066 DATA   0,255,  0
1068 DATA  96, 24,  6
1070 DATA  96, 24,  6
1072 DATA 120, 24, 30
```

```
1Ø74 DATA  24, 24, 24
1Ø76 DATA  3Ø, 6Ø,12Ø
1Ø78 DATA  31,255,248
1Ø8Ø DATA   7,255,224
1Ø82 DATA   7,255,224
1Ø84 DATA   1,129,128
1Ø86 DATA   1,195,128
1Ø88 DATA   1,195,128
1Ø9Ø DATA   Ø,1Ø2,  Ø
1Ø92 DATA   Ø,126,  Ø
1Ø94 DATA   Ø, 6Ø,  Ø
1Ø96 DATA   Ø, 6Ø,  Ø
1Ø98 DATA   Ø, 24,  Ø
11ØØ DATA   Ø, 24,  Ø
11Ø2 DATA   Ø, 24,  Ø
11Ø4 DATA   .,  .,  .
2ØØØ GOSUB 1Ø: C=11: GOSUB 3Ø
2Ø1Ø POKE 5328Ø,C
2Ø2Ø SP=Ø: REM  FIRST SP
2Ø3Ø SE$="CRAFT1": GOSUB 81Ø
2Ø4Ø GOSUB 18Ø: REM  ON
2Ø5Ø GOSUB 2ØØ: REM  WIDER
2Ø6Ø GOSUB 22Ø: REM  TALLER
2Ø7Ø GOSUB 24Ø: REM  SP OVER SHAPES
2Ø8Ø C=1: GOSUB 26Ø
2Ø9Ø X=11Ø: Y=75: GOSUB 27Ø
23ØØ REM::::::SECOND SPRITE
231Ø SP=1: REM  LOWER SP PRIOR
232Ø SE$="CRAFT2": GOSUB 81Ø
233Ø GOSUB 18Ø
234Ø C=7: GOSUB 26Ø
235Ø X=275: Y=75: GOSUB 27Ø
```

Chapter Eight

ADVANCED SPRITE GRAPHICS

Sprites, as you have seen already, are an interesting alternative to drawing foreground shapes. Sprites can move, flash, change colors, and reside in a color block already displaying a foreground color. There are many directions you can go with sprites, now that you understand what a sprite is and how to create one. In this chapter, we concentrate on controlling sprites.

The first step is to learn to connect a sprite to the keyboard and to joysticks. This will allow you to control the movement of up to two sprites while the program is running. Under keyboard control, a sprite will move on the screen in one of four directions, depending on the key you press. Under joystick control, a sprite will move in the direction the joystick is positioned. We call a sprite that is connected to a joystick or the keyboard a *player sprite*.

You will also learn how to control *missile sprites* and *target sprites*. A missile sprite can be shot from a player sprite. This is done either by pressing a key, or by pressing a joystick button. The computer will note the key or button pressed, and will then shoot the correct missile sprite from the correct player sprite.

A target sprite is more "passive" than the other sprite types. It is placed at an X,Y "starting position," and then sent off in one of eight directions. From that point on it is a moving target. It moves in a straight path until it reaches the screen's edge. The target then wraps back around to the opposite screen side, and moves across the screen again. This same journey is taken over and over again by the target sprite.

You will also learn how to detect when a sprite has collided with a shape or sprite, and what visual effects you can create when it does. Finally, you will learn how to turn the Commodore's sound on and off, as well as keep one or two-player scoring totals.

If all this sounds like you're headed for an arcade game, you're right. The two spacecraft created in Chapter 7 will be turned into player sprites, and step-by-step instructions on creating your own video game will be given.

Introduction to the Interrupt System

Before we supply you with this chapter's collection of tools, you need to become acquainted with the term "interrupt system." Some of the advanced sprite features were entered as machine language data in Chapter 1. This machine language makes use of the Commodore 64's interrupt system.

An interrupt system contains a set of tasks that the computer performs over and over again during the course of *one second*. The Commodore 64 has three interrupt tasks:

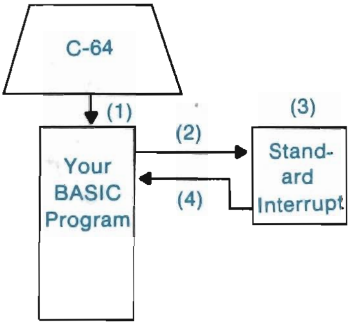
(1) Increment TI$ (this is a time clock);

(2) Check to see if its time to flash the cursor (if so, the cursor flashes once);

(3) Check the keyboard to see if a key has been pressed.

Sixty times each second, the Commodore stops (interrupts) whatever it is doing, saves its place, and performs steps (1), (2) and (3) above. It performs these tasks at such an incredible speed that the person operating the computer is usually unaware that it is even happening. The flashing cursor and the computer's response to a keypress are the only hints that an interrupt has taken place.

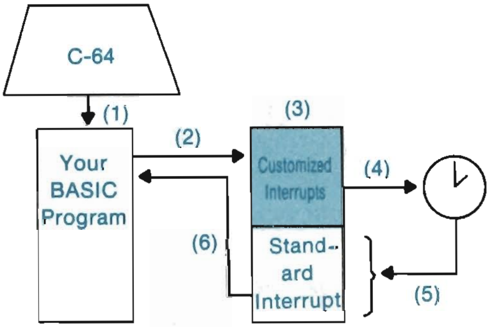
When the computer completes the standard interrupt tasks, it returns to whatever it was doing prior to the interrupt. The diagram below illustrates how the interrupt system operates during a program's execution.



In step (1), the computer begins excuting the BASIC lines in your program. Then, when 1/60th of a second has passed, the computer temporarily abandons your program (step 2) and performs the three standard interrupt tasks (step 3). The program is returned to (step 4) as soon as the interrupt tasks have all been performed. Steps (2) through (4) are continually processed the entire time the computer is on.

There is a way to add your own "customized" tasks to the interrupt system. This is what you did, unknowingly, in Chapter 1. Some of the machine language data entered in that chapter breaks into the Commodore's interrupt system and adds several customized tasks that deal with sprite graphics.

Now *you* can control how often an interrupt occurs. You do this by assigning a value to the variable VE. The higher the value of VE, the longer the wait between interrupts. When an interrupt does occur, the list of customized interrupt tasks is processed *first*, before the standard interrupt tasks are. This can be diagrammed as follows:



Step (1) is where the computer begins executing the program. In step (2), the program is interrupted every *nth* of a second (according to the current value of VE). All customized interrupt tasks are processed at each interrupt (step 3). The computer then looks at a clock (step 4) to see if it's time for the standard interrupt tasks. If 1/60th of a second has passed since they were last performed, then the standard interrupt tasks are again executed (step 5). If it's not time for the standard interrupt commands, step (5) is skipped, and step (6) comes up. At step (6) the computer returns to the program, exactly where it was prior to the interrupt.

You will learn how to "call" your customized interrupt commands into action in the next section. There, too, you will learn more about the variable VE. The interrupt system is important in that it virtually allows two separate things (your BASIC program and your list of interrupt commands) to be performed simultaneously—or so they will appear. You now know, though, that the computer is actually going back and forth between the two, running part of the program and all of the customized interrupts as it goes.

Making and Moving Action Sprites

The three sprite types discussed earlier (player sprites, missile sprites, and target sprites) can all be categorized as *action sprites*. An action sprite is one that is controlled through the customized interrupt commands, and involves movement of one kind or another.

The first action sprites to create are the two player sprites. Begin by LOADing your "CHAPTER 7" program. Note: Chapter 7's exercise solution (lines 1060-1104) must be a part of this program. If you did not complete Chapter 7's exercise, type the solution DATA statements from that exercise before continuing.

Next, list line 172 and set C equal to 2000 in it (we will be using the CRAFT1 and CRAFT2 DATA statements to define our player sprites). RUN the ZAP routine.

Many of the programming techniques we will use to create this chapter's game are already familiar to you. For example, you will enter three program lines that plot random points on the graphics screen. This will be done with the RND(1) command, which is a command we have used before. Any command that has been introduced before will be used here without any explanation, in an effort to keep this chapter's size reasonable.

Type program lines 2000 through 2130, below, which set up the program for our game. These lines clear the graphics screen, set the background color, and also plot 50 random "stars" across the dark sky.

```
2ØØØ REM::::::SET UP PROGRAM
2Ø1Ø GOSUB 1Ø: C=11: GOSUB3Ø
2Ø2Ø POKE 5328Ø,C
21ØØ REM::::::STARS
211Ø C=1: FOR L = 1 TO 5Ø
212Ø X=RND(1)*32Ø: Y=RND(1)*2ØØ
213Ø GOSUB 4Ø: NEXT L
```

We are not going to use the planet in this program, primarily because of the time it takes to draw it. You may, however, prefer the wait over no planet at all. If you add the lines that draw and paint the planet (see Chapter 7 to find out what they were), use line numbers 2140 through 2162, every even line number.

Controlling Player Sprites Through the Keyboard

You now need to "set up" your player sprites. A player sprite is set up just as you learned in the previous chapter:

(1) Assign the sprite a pointer number;
(2) Apply all the sprite features you desire;
(3) Color the sprite;
(4) Place the sprite;
(5) Turn on the sprite.

Player sprites *must* be assigned a sprite pointer number of 0 or 1. If you are only using one player sprite in your program, you must assign it sprite pointer #0. If you are using two player sprites, as we will here, then you must assign one SP #0, and one SP #1.

Type the program lines below to define two player sprites. Sprite #0 (lines 2210-2230) will be defined with CRAFT1 DATA statements. Sprite #1 (lines 2240-2260) will be defined with CRAFT2 DATA statements.

```
2200 REM::::::SET UP PLAYER SPRITES
2210 SP=0: SE$="CRAFT1": GOSUB 810
2220 C=0: GOSUB 260: GOSUB 240
2230 X=179: Y=170: GOSUB 270: GOSUB 180
2240 SP=1: SE$="CRAFT2": GOSUB 810
2250 C=1: GOSUB 260: GOSUB 250
2260 X=159: Y=20: GOSUB 270: GOSUB 180
```

Notice that each sprite is given a color (C=?: GOSUB 260), given shape versus sprite priority (GOSUB 240 or GOSUB 250), placed on the screen (X=?: Y=?: GOSUB 270); and turned on (GOSUB 180). So far, this setup procedure is no different than one you might expect for a non-action sprite. To let the computer know that these are to be action sprites, you must add a new tool to your tool kit:

```
290 REM:::HOOK UP ACTION SPRITES
291 SYS 50023,KB,P1,P2,255-M1,255-M2,255-T1,VE
292 RETURN
```

This tool does many things, one of which is to connect player sprites to the keyboard or to joysticks. We will begin by connecting Sprite #0 and Sprite #1 to the keyboard. Four variables must be set in order to do this:

(1) VE = 0 *to* 65 (VElocity of travel; 0=fastest, 65=slowest)

(2) KB = 1 (hook up player sprites to KeyBoard) *or* 0 (hook up player sprites to joysticks)

(3) P1 = 1 (use Sprite #0 as Player 1 action sprite) *or*
0 (use Sprite #0 as non-action sprite)

(4) P2 = 1 (use Sprite #1 as Player 2 action sprite) *or*
0 (use Sprite #1 as non-action sprite)

The variable VE stands for VElocity, or speed, and should be set to a speed code between 0 and 65. All action sprites travel at the *same* speed, as determined by the value of this variable. Keep in mind that zero is the fastest speed code, and sixty-five is the slowest. (You will find that the speed can be set to codes higher then sixty-five, but movement becomes incredibly slow when it is.)

The variable KB stands for KeyBoard, and it is used to "enable" (KB=1) or "disable" (KB=0) the keyboard. When the keyboard is enabled, all player sprites will be under keyboard control. When the keyboard is disabled, all player sprites are under joystick control.

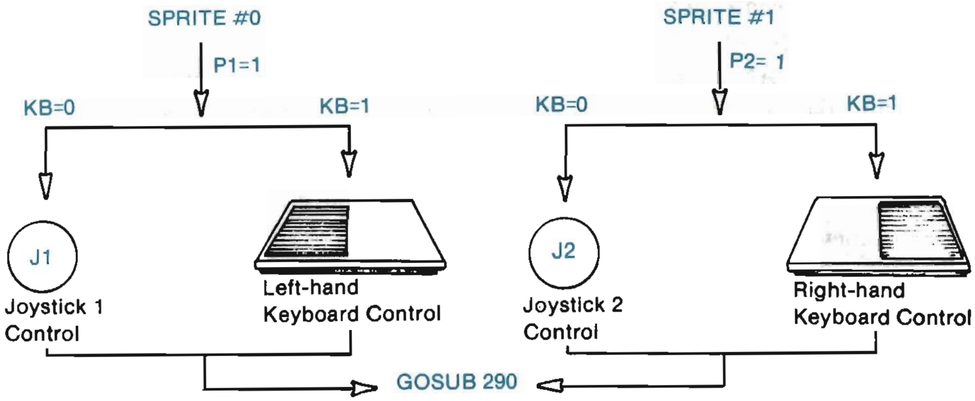
The variable P1 stands for Player #1, and represents one of two possible game players. When P1 is set to 1, Player #1 is "enabled." This gives Player #1 joystick/keyboard control over Sprite #0. Player #1 is "disabled" when P1 is set to 0, and thus has no control over Sprite #0 (i.e., Sprite #0 is a non-action sprite).

The type of control Player #1 will have depends on whether the keyboard is enabled (KB=1), or the joysticks are enabled (KB=0). Sprite #0 is controlled by Joystick #1 when the joysticks are enabled. Sprite #0 is controlled by the left-hand control keys (explained below) when the keyboard is enabled.

The variable P2 stands for Player #2, and represents the second of two possible game players. Setting P2=1 gives Player #2 keyboard/joystick control over Sprite #1. Setting P2=0 returns Sprite #1 to a non-action sprite. Player #2 controls his/her sprite with Joystick #2 (if KB=0), or with the right-hand control keys (if KB=1).

You may enable only Player #1, or both Player #1 and Player #2. You cannot enable Player #2 by itself. If Player #1 is disabled (i.e., P1=0), *all* other action sprites are automatically disabled.

The diagram below reviews how to set the variables that control your player sprites.



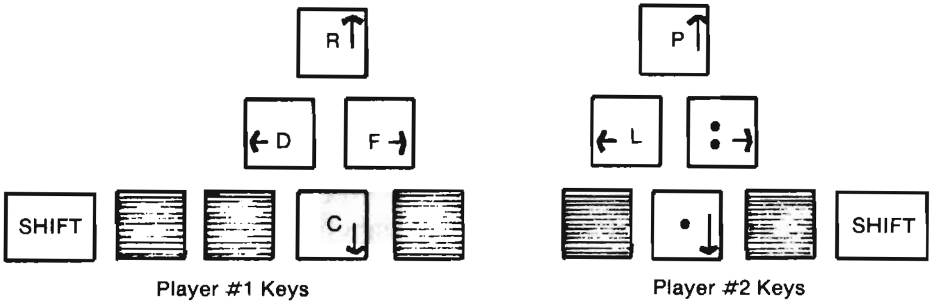
Compare the above chart with the program lines below. Line 2610 enables the keyboard and sets the speed of all action sprites to 15. Line 2620 enables Player #1 and Player #2, turning Sprites 0 and 1 into action sprites. Finally, line 2630 follows all this with a GOSUB 290 to call the necessary HOOK UP ACTION SPRITES tool.

```
2600 REM::::::ENABLE ACTION SPRITES
2610 KB=1: VE=15
2620 P1=1: P2=1
2630 GOSUB 290
```

Add these lines to your program, and then RUN it.

The screen should turn a dark grey, and then 50 points will be randomly plotted in white (they may appear to be plotted in random colors). It will take several moments for the player sprites to be placed on the screen, at which time you will have keyboard control over them.

What does it mean to have "keyboard control" over a sprite? It means that you can move the sprite while the program is running. You can move the sprite up, down, right, or left by the press of a single key. Each player (#1 and #2) has his own set of five controlling keys:



Take Player #1's position as we discuss moving Sprite #0. Begin by tapping the **R** key several times. The black spacecraft should jump towards the top of the screen each time you tap the key. Continue to tap this key until the sprite moves entirely off the top screen edge. Finally, hold the **R** key down, and watch as the black spacecraft re-appears at the *bottom* of the screen.

> IMPORTANT: An incorrectly running program could be the result of typing mistakes in the subroutine, typing mistakes in the main routine, or *typing mistakes in your machine language.* Check the subroutine and main rountine for errors first. If none are found, refer to Chapter 1 to refresh your memory on checking and correcting the machine language. You should find the error somewhere in blocks 7-11.

Unless your program instructs otherwise, a player sprite will always wrap around the screen to the opposite edge when a direction key is held down for an extended period. Try holding down the **C** key, and the sprite will move in a downward motion. When the sprite crosses the bottom screen edge, it will wrap around to the top screen edge. Finally, try the **D** and **F** keys. They move Sprite #0 horizontally. The list of Player #1 keyboard direction controls is as follows:

Right	—	**F**
Down	—	**C**
Left	—	**D**
Up	—	**R**

The **SHIFT** keys are set up to function as "fire" buttons. They can shoot missile sprites out of the player sprites. You will learn more about the use of these keys after you create some missile sprites.

You should experiment with all four of the Player #1 sprite directions now. If you lose your orientation and are not sure where the sprite is, hold down the **R** key for a few seconds, and then hold down the **F** key for a few seconds. Switch back and forth between these two keys until you see the sprite again (this should only take a moment).

Player #2 (Sprite #1) can also be moved in four different directions. Press **P** to move the sprite up. Hold this key down, and the sprite will wrap around just as the other had. Tap the **.** key several times, and the white spacecraft takes several small leaps downward. **L** and **:** control this sprite's horizontal movement. The list of this sprite's keyboard direction controls is:

Right	—	**:**
Down	—	**.**
Left	—	**L**
Up	—	**P**

Move both sprites to the same screen location, and test sprite versus sprite priority. You will find that Sprite #0 (the black spacecraft) appears on top of Sprite #1 (the white spacecraft). This is due to the sprite numbers. Sprite #0, having the lower sprite number, will always appear in front of Sprite #1.

You should take about fifteen minutes to practice sprite keyboard control. It will take some time to become comfortable with this form of sprite movement, but, in time, you will. You should practice with different sprite velocities, too. Compare the difference between a velocity of 0 and a velocity of 65. Notice that it's harder to control your sprite's movement at faster speeds.

It is a little more difficult returning to text mode than before. You will have to press **RUN/STOP** and tap **RESTORE**, and you may have to do this several times. Later you will be adding the necessary loop that allows a more graceful exit out of graphics mode. The next paragraph discusses problems you could be encountering. Generally speaking, there are few "hard to find" problems where sprites are concerned.

If your sprites move, but not in the direction you specify: (1) make sure you are pressing the right key for the direction and sprite you desire; (2) make sure you set KB to 1 in the main routine. Both the keyboard and the joysticks are always "operating," and will move the player sprites. The keyboard, however, will only operate correctly when KB=1, and the joysticks will only operate correctly when KB=0.

If you can't get your sprites to move at all, check lines 2620 and 2630. An easy error would be to call a subroutine other than 290 in line 2630. The only other possible problem areas are the subroutines, the machine language, or the computer itself. *It is unlikely* that you have come this far with machine language errors or a problem with your computer.

Check your subroutines thoroughly. Don't just look at the ones typed in this chapter, look at each subroutine your program is currently using. It is not that hard to erase a subroutine line inadvertantly. All you have to do is start to type a line number (say, 221 to begin line 2210), change your mind about adding the line, and press **RETURN**. This simple act would delete line 221 from your subroutines.

Controlling Player Sprites with Joysticks

If you own one or more joysticks, you are probably eager to connect the player sprites to them. (If you don't own joysticks, continue to the next section where you will learn how to create and enable "missile" sprites.) You may have already figured out the one simple change that turns keyboard control into joystick control. Change line 2610 to the following:

```
2610 KB=0: VE=15
```

The value of KB is the only thing that changes from keyboard control to joystick control. The player sprites are set up the same, and all other variables (P1, P2, and VE) serve the same purpose from one control method to the other. Before running the program, check the following:

(1) Is there a joystick connected to CONTROL PORT 1 on the right side of the Commodore 64 computer? This joystick is considered Joystick #1, and it must be connected to Port 1 in order to operate Sprite #0.

(2) Is there a joystick connected to CONTROL PORT 2 on the right side of the Commodore 64 computer? This joystick is considered Joystick #2, and it must be connected to Port 2 in order to operate Sprite #1.*

RUN the program. Wait as the stars are plotted and the spacecraft are placed. As soon as the second sprite appears, you are the master of the game. Pick up Joystick #1 and begin moving Sprite #0 around the screen. The sprite will move in the direction you push the joystick. If you become disoriented, and the sprite is somewhere off the visible screen, push the joystick in any diagonal direction. This should bring your sprite back on the screen.

Joystick #2 (in Port 2) will operate the white spacecraft. Try moving it around the screen area as well. You should set VE to a speed that is comfortable for you, and spend a few minutes getting a feel for how the sprites respond to a change in the joystick direction.

The rest of this chapter assumes you are using keyboard control (since every reader has a keyboard, but not every reader has a joystick). If you like, you can continue with KB set to 0, using your joysticks as the control mechanisms. Just ignore references to keyboard control keys from now on.

*Joystick #2 is optional if you only intend to operate one player sprite. Joystick #1 is not optional when using joystick control on player sprites.

Creating and Controlling Missile Sprites

Your program can contain up to two missile sprites. Missile #1 must be assigned sprite #2, and Missile #2 (if any) must be assigned sprite #3. Player #1 has control over Missile #1, and Player #2 has control over Missile #2. Set KB to 1 (enable keyboard) in line 2610 so that we may explore this new area of action sprites.

The design of a missile sprite may be anything you wish. You can make the missiles resemble small spacecraft, a group of bullets, one long shaft—anything that fits on the Sprite Design Grid. We use the DATA statements below to describe each missile. These data statements describe a sprite made up of only three tiny points in a row. Add this missile shape to your library.

```
1106 DATA "MISSILE",31,Ø
1108 DATA  Ø,  Ø,  Ø
111Ø DATA  Ø,  Ø,  Ø
1112 DATA  Ø,  Ø,  Ø
1114 DATA  Ø,  Ø,  Ø
1116 DATA  Ø,  Ø,  Ø
1118 DATA  Ø,  Ø,  Ø
112Ø DATA  Ø,  Ø,  Ø
1122 DATA  Ø,  Ø,  Ø
1124 DATA  Ø, 24,  Ø
1126 DATA  Ø,  Ø,  Ø
1128 DATA  Ø, 24,  Ø
113Ø DATA  Ø,  Ø,  Ø
1132 DATA  Ø, 24,  Ø
1134 DATA  Ø,  Ø,  Ø
1136 DATA  Ø,  Ø,  Ø
1138 DATA  Ø,  Ø,  Ø
114Ø DATA  Ø,  Ø,  Ø
1142 DATA  Ø,  Ø,  Ø
1144 DATA  Ø,  Ø,  Ø
1146 DATA  Ø,  Ø,  Ø
1148 DATA  Ø,  Ø,  Ø
115Ø DATA  .,  .,  .
```

You set up a missile sprite in almost the same fashion as player sprites, except that you do *not* place it on the screen or turn it on. These two tasks are done automatically by the computer whenever the player presses the missile's fire button. Each time a player presses his fire button, the computer:

(1) Connects the missile to its player sprite;
(2) Turns the missile on; and
(3) Shoots it in a specified direction.

The steps you have to take to set up a missile sprite are:

(1) Set SP to a pointer number of 2 or 3;
(2) Retrieve the missile description you desire;

(3) Set the missile's color code (followed by GOSUB 260);
(4) Call any other sprite features you want for the missile; and
(5) Set the shape versus sprite priority for the missile.

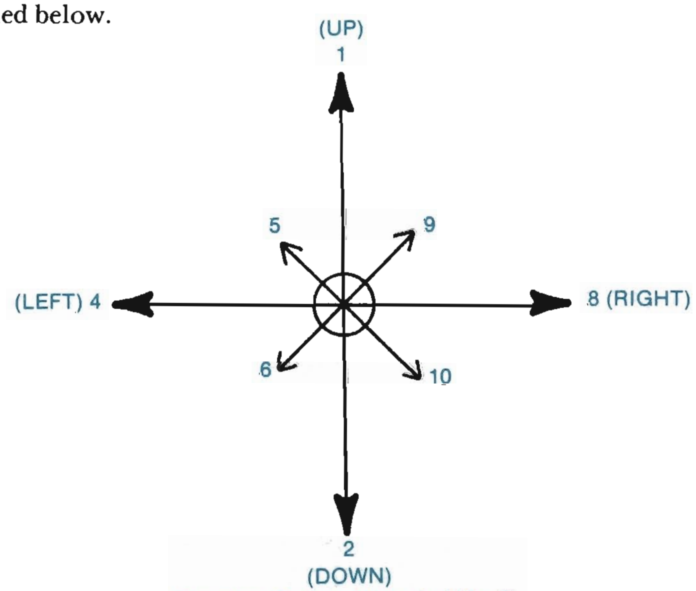
Enter the following program lines that take care of these steps for our missile sprites:

```
2300 REM::::::::SET UP MISSILE SPRITES
2310 SP=2: SE$="MISSILE": GOSUB 810
2320 C=0: GOSUB 260: GOSUB 240
2330 SP=3: SE$="MISSILE": GOSUB 810
2340 C=1: GOSUB 260: GOSUB 250
```

There can be up to two missile sprites operating at one time: one for each player sprite. Each missile sprite may have its own set of DATA statements, or they can both use the same set of DATA statements. This depends on what you would like each missile to look like. Our program uses the same DATA statements to describe both of the missiles.

A missile sprite must be "enabled" in order for it to be fired from a player sprite. Two variables are involved with enabling/disabling your missiles: M1 and M2. Missile #1 is disabled when M1=0, and Missile #2 is disabled when M2=0. Again, a disabled sprite is simply a non-action, normal sprite.

Enabling missile sprites is done a little differently than enabling player sprites. You enable a missile sprite by setting its variable to a *direction* code. This code tells the computer in which direction (one of eight) you want the missile sprite to travel when fired. The computer knows that if M1 is assigned a direction code, you must want Missile #1 enabled. Likewise, if M2 is assigned a direction code, you want Missile #2 enabled. The eight possible shooting directions and their codes are diagrammed below.



Direction Codes (M1 = ? : M2 = ?)

The eight shooting directions have codes ranging from one to ten. "Up" has a code of 1, "down" has a code of 2, "left" has a code of 4, and "right" has a code of 8 (these are all powers of two). Diagonal direction codes are a result of adding vertical and horizontal codes together. Diagonally "up and to the right" (code 9) is the result of adding the codes for "up" and "right" together (1 + 8 = 9). Diagonally "down and to the left" (code 6) is a result of adding the codes for "down" and "left" together (2 + 4 = 6). This principle is the same for the remaining two diagonal directions.

Player #1's sprite currently points upward. We, thus, want this player's missile to shoot upward. Setting M1 to 1 (M1=1) before GOSUB 290 will enable Missile #1 and point it in an upward direction.

Player #2's sprite currently points downward. Its missile should then logically shoot downward. It will if you set M2 equal to 2 before calling Tool 290.

Change line 2620 to the following:

```
262Ø P1=1: P2=1: M1=1: M2=2
```

That's all that is required to set up and enable missile sprites. RUN the program.

The computer must now plot the stars, place the player sprites, and then place the missile sprites. You will not be able to see the missile sprites as they are placed, but they won't become active until they are. Wait just a few moments after the spacecraft are on the screen, and then try pressing your fire buttons (**SHIFT** keys). The left-hand **SHIFT** key will fire the missile from Player #1's sprite. The right-hand **SHIFT** key will fire the missile from Player #2's sprite.

If you are using joysticks to control your sprites, the joystick button is the fire button. Use this button whenever we refer to a **SHIFT** key here. (*Note:* Joysticks can have more than one button. Press each one until you locate the fire button.)

The missiles will fly quickly, so it will be hard to see exactly what they look like. They resemble three small bullets shooting from the spacecraft. Try holding down a **SHIFT** key to "rapid-fire" a missile. You will find that a player's first missile must travel off the screen before that player can fire another missile. Grab a friend or relative and try firing at each other. As yet, nothing will happen when you score a "hit," but you can see that the game is advancing quite rapidly.

A missile sprite will move in a straight path until it reaches the screen edge. The "customized" interrupt commands cause the computer to turn that sprite off once it crosses the screen border. The missile sprite is then automatically turned on and placed with its player sprite the next time the appropriate fire button is pressed.

Return to text mode and try changing a missile's direction and color. Also, you might enjoy replacing our "MISSILE" description with one of your own.

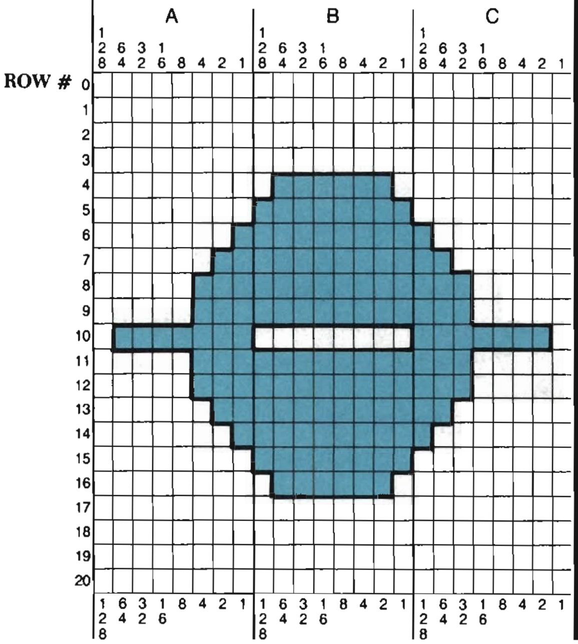
A missile sprite can travel in only one direction. That direction is determined by the value of its corresponding variable (M1 or M2). The vertical/horizontal direction codes are 1 (up), 2 (down), 4 (left), and 8 (right). Diagonal firing codes are found by adding the corresponding horizontal and vertical direction codes together. A missile is enabled whenever its variable has been set to a direction code, and Tool 290 is called. (Again, if P1=0, *all* action sprites are automatically disabled.)

Creating and Controlling a Target Sprite

A target sprite is the last of the action sprites that can be added to your game. There can be only one target sprite, and it must have a sprite number of 4. A target sprite cannot shoot missiles, but it can get hit. This adds a twist to a two-player game, and an "opponent" to a one-player game. The target sprite we use looks like this:

SPRITE DESIGN GRID

(TOP)



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	62	0
	DATA	0	255	0
	DATA	1	255	128
	DATA	3	255	192
	DATA	7	255	224
	DATA	7	255	224
	DATA	127	0	254
	DATA	7	255	224
	DATA	7	255	224
	DATA	3	255	192
	DATA	1	255	128
	DATA	0	255	0
	DATA	0	62	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0
	DATA	0	0	0

The DATA statements below must be added to your program to define this sprite:

```
1152 DATA "TARGET",31,Ø
1154 DATA   Ø,  Ø,  Ø
1156 DATA   Ø,  Ø,  Ø
1158 DATA   Ø,  Ø,  Ø
1160 DATA   Ø,  Ø,  Ø
1162 DATA   Ø, 62,  Ø
1164 DATA   Ø,255,  Ø
1166 DATA   1,255,128
1168 DATA   3,255,192
1170 DATA   7,255,224
```

```
1172 DATA   7,255,224
1174 DATA 127,  Ø,254
1176 DATA   7,255,224
1178 DATA   7,255,224
118Ø DATA   3,255,192
1182 DATA   1,255,128
1184 DATA   Ø,255,  Ø
1186 DATA   Ø, 62,  Ø
1188 DATA   Ø,  Ø,  Ø
119Ø DATA   Ø,  Ø,  Ø
1192 DATA   Ø,  Ø,  Ø
1194 DATA   Ø,  Ø,  Ø
1196 DATA   .,  .,  .
```

A target sprite is set up exactly as a player sprite is. The only difference is that a target sprite must be assigned a sprite number of 4. Add the program lines below that set up a target sprite in your program:

```
24ØØ REM::::::SET UP TARGET SPRITE
241Ø SP=4: SE$="TARGET": GOSUB 81Ø
242Ø C=7: GOSUB 26Ø: GOSUB 24Ø
243Ø X=159: Y=99: GOSUB 27Ø: GOSUB 18Ø
```

Finally, you need to enable the target sprite. This is done by setting the variable T1 to a direction code. A target sprite may travel in one of eight different directions. Not surprisingly, the list of eight target direction codes is the same as the list of missile direction codes:

Up = 1
Up and Right = 9
Right = 8
Right and Down = 10
Down = 2
Down and Left = 6
Left = 4
Left and Up = 5

Let's have the target move horizontally to the left (T1=4). Change line 2620 to the following:

```
262Ø P1=1: P2=1: M1=1: M2=2: T1=4
```

RUN the program. Now there are five different action sprites that must be placed on the screen before you have control. The last sprite will be the yellow target sprite. It will be placed near the center of the screen, and will immediately begin its travel leftward.

This target sprite will continue to the left screen edge, and then wrap around back to the opposite side of the screen. It will then move across the screen again, and again, and again. Your "aim" is to shoot the target. This is not as easy as it looks.

Try a few shoots. With practice, you will get the timing down and may need to speed up the game to increase the challenge.

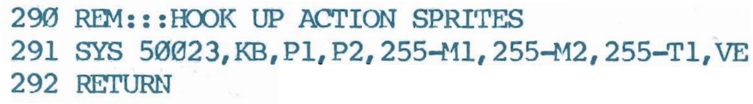
Change the target's direction in the main routine and RUN the program again. Be certain to try a diagonal direction, because the "wrap around" feature may not work as expected for this path of motion.

That is all there is to creating and controlling a target sprite. Basically, you create a sprite, assign it sprite #4, set T1 to a direction code, and call Tool 290.

All three action sprite types have now been covered. These sprites are moved by "customized" commands that have been added to the Commodore 64's interrupt system. Sprites 5, 6, and 7 may be placed in the game, but they can only operate as normal, non-action sprites.

Tool 290 executes the machine language necessary to put action sprites into motion. There are seven variables associated with this tool. These variables are reviewed in the tool box below, as well as at the end of the chapter.

TOOL 290 :::::: HOOK UP ACTION SPRITES



```
29Ø REM:::HOOK UP ACTION SPRITES
291 SYS 5ØØ23,KB,P1,P2,255-M1,255-M2,255-T1,VE
292 RETURN
```

What It Does: This tool adds some "customized" commands to the Commodore 64's interrupt system. These commands allow you to turn Sprites #0 and #1 into "player sprites." A player sprite can be moved around the graphics screen under keyboard or joystick control. The value of variables P1 and P2 determine whether Sprites #0 and #1 are player sprites.

You can also turn Sprites #2 and #3 into "missile sprites." These sprites can be shot from your player sprites by the touch of a key or the press of a joystick button. Sprite #2 can be shot from Sprite #0, and Sprite #3 can be shot from Sprite #1. Variables M1 and M2 enable/disable missile sprites.

Finally, you can turn Sprite #4 into a target sprite. A target sprite is placed on the screen and sent along one of eight linear paths. The sprite makes the trip along this path over and over again, serving as a moving target for your players sprites. The variable T1 enables/disables the target sprite.

Example Use: Seven variables are checked by the computer when it calls Tool 290. These seven variables determine: (1) which action sprites are enabled; (2) the method of control you will have over the player sprites; (3) the direction of travel for the missile sprites and the target sprite; and (4), the speed at which all action sprites will travel. These variables are:

KB : Stands for Keyboard. KB=1 enables keyboard control. KB=0 disables keyboard control (enabling the joysticks).

VE : Stands for Velocity. Value can range from 0 (fastest) to 65 (slowest).

P1 : Stands for Player #1 and controls Sprite #0. P1=0 disables *all* action sprites. P1=1 connects Sprite #0 to Joystick #1 or the left-hand key controls.

P2 : Stands for Player #2 and controls Sprite #1. P2=0 disables Sprite #1, leaving it a non-action sprite. P2=1 connects Sprite #1 to Joystick #2 or the right-hand control keys.

M1 : Stands for Missile #1 and controls Sprite #2. M1=0 disables Sprite #2, leaving it a non-action sprite. Setting M1 to one of eight direction codes connects Sprite #2 to Sprite #0, making it a firing missile for Player #1. The direction code determines this missile's direction of fire.

M2 : Stands for Missile #2 and controls Sprite #3. M2=0 disables Sprite #3, leaving it a non-action sprite. Setting M2 to one of eight direction codes connects Sprite #3 to Sprite #1, making it a firing missile for Player #2. The direction code determines this missile's direction of fire.

T1 : Stands for Target #1 and controls Sprite #4. T1=0 disables Sprite #4, leaving it a non-action sprite. Setting T1 to one of eight direction codes will enable the target sprite. It can then be placed on the screen at an X,Y position, and will travel in the direction of its "enable" code. It follows that linear path over and over again.

Example program lines that will enable all action sprites, connect the player sprites to the keyboard, and set the speed at 15 are:

```
2610 KB=1: VE=15
2620 P1=1: P2=1: M1=1: M2=2: T1=4
2630 GOSUB 290
```

Technical Description: In order to better understand interrupts, let's look at a simple analogy.

Fred Baker owns a bakery. The bakery has two ovens, and each oven has a timer with an alarm. From January through November Fred uses only one oven, in which he bakes one loaf at a time. He places the dough in oven "A," sets timer "A" for one hour, and then goes in the back room to do his bookkeeping. After one hour the timer reaches "0" and the alarm goes off. Fred then marks his place in his bookkeeping, shuts off the alarm, takes the bread out of the oven, and starts the process all over again.

When December rolls around, however, Fred always gets a big order for Christmas cookies. He still bakes his usual amount of bread, but now he must use both ovens. Oven "A" is still used for bread, and oven "B" is now used for cookies. The cookies, however, only need to bake for twenty minutes, and therefore Fred sets timer "B" for twenty minutes. Realizing that baking three batches of cookies will take the same amount of time as baking one loaf of bread, Fred concludes that there is no need for both alarms to go off. Therefore, when he sets the timer for the bread, he sticks a wad of paper between the bell and the striker.

Now when an alarm goes off, Fred knows that the cookies are done. He again marks his place, shuts off alarm "B," removes the cookies, puts in a fresh batch, and resets the timer. Fred also checks oven "A" to see if the bread timer has reached "0," and then proceeds to remove the bread if it has.

If you are wondering what this has to do with interupts, the answer is: everything. Inside the Commodore are two timers. Timer "A" is at memory locations 56324 and 56325, and timer "B" is at memory locations 56326 and 56327. Each timer owns a bit at location 56333. These two bits correspond to the "alarms" in our analogy. Normally, when an alarm is set, an interrupt occurs. It is possible, however, to "stick a wad of paper in" so the alarm goes off but an interrupt doesn't occur.

The normal interrupt routine "bakes the bread," or in this case, increments TI$, flashes the cursor, and checks the keyboard. Our custom interrupt, however, "bakes the cookies," or in this case, moves the action sprites. Since we want this to happen more frequently than the bread (normal alarm) we "stick a wad of paper" in timer "A," and check the timer after the action sprites have been moved.

Collision Detection

You are now going to add tools that make the game worth playing. These tools deal with sprite *collisions*. A collision occurs whenever two objects come together, normally with some amount of force. We are not concerned with the amount of force behind a sprite collision, only that a sprite has run into, bumped, or touched something. There are two types of sprite collisions:

(1) Sprite-to-Sprite collisions; and
(2) Sprite-to-Foreground collisions.

You want to be able to detect Sprite-to-Sprite collisions in the event a missile sprite hits a player sprite, or a missile sprite hits the target sprite. If you can detect the exact instance such a collision occurs, you can have the computer respond appropriately (e.g., by adding points to a player's score).

Detecting when a sprite touches a foreground shape allows greater flexibility and creativity in the design of the game. For example, you could have the player sprites "blow up" if they run into foreground shapes. You could even draw a border around the graphics screen, and deduct points from a player's score if he tries to travel off the screen.

Sprite-to-Sprite Collisions

Memory location 53278 holds the answer to whether or not a sprite has collided with another sprite. This location has seven little "slots," numbered 128, 64, 32, 16, 8, 4, 2, and 1. These slot numbers should look familiar. They are the same numbers assigned to each section on your Sprite Design Grid. All eight sprites have their own slot and slot number, as shown below:

SLOT #:	128	64	32	16	8	4	2	1
SPRITE #:	7	6	5	4	3	2	1	0
ACTION VAR.:	——	——	——	T1	M2	M1	P2	P1

A sprite's slot number will "turn on" whenever it collides with another sprite. It takes two sprites to make a collision, so at least two slot numbers will always be on after a collision has taken place. For example, if sprites 1 and 4 collide, the following slots in memory location 53278 will be turned on:

				“ON” ↓			“ON” ↓	
SLOT #:	128	64	32	16	8	4	2	1
SPRITE #:	7	6	5	4	3	2	1	0
ACTION VAR.:	——	——	——	T1	M2	M1	P2	P1

If you had the computer look (PEEK) into memory location 53278 after this collision, it would find a value of 18 there:

16 (Sprite 4's slot) + 2 (Sprite 1's slot) = 18

In many instances, the ability to PEEK into this memory location provides you with enough information to decide what to do. For example, you could have a program line reading: IF PEEK(53278)=18 THEN ... (whatever you wanted to happen when sprites 1 and 4 collide). However, there will be times when more than two sprites collide at the same time. There will also be times when two separate collisions occur at the same time. These instances would cause several slots to be turned on at the same time, making it difficult for you to determine the exact collisions that took place.

Suppose that you have variable C1 holding a count of Player #1's current score. You decide that if Player #1 hits Player #2, 25 points should be added to Player #1's score. You look at the above diagrams, see that Missile #1 has a slot value of 4, and Player #2 has a slot value of 2. Based on these two slot numbers, you insert the following program line into your program:

```
27ØØ IF PEEK(5327Ø)=6 THEN C1=C1+25
```

Now, suppose that Missile #1 does hit Player #2, but, at the same time, Player #1 runs into the target sprite. The computer would look into memory location 53278 and find the following:

Collision

Collision

SLOT #:	128	64	32	16	8	4	2	1
SPRITE #:	7	6	5	4	3	2	1	0
ACTION VAR.:	—	—	—	T1	M2	M1	P2	P1

16 + 4 + 2 + 1 = 23

The value of this memory location would be 23 and not 6, so Player #1 would lose out on his/her 25 points—even though they did indeed hit Player #2's ship. Clearly, there must be a way to see if two slots are turned on, ignoring any others that might also be.

There is. A BASIC statement called "AND" compares the slots that are currently "on" to the slots you want to check. It then returns a value representing the "matching" slots found in the comparison. For example, you might have a program line that contains the following:

```
IF (PEEK(53278) AND 6)=6 THEN C1= C1+25
```

This tells the computer to check memory location 53278, compare the slots that are "on" to slots 4 and 2 (i.e., 6), and return the total value of any matches. The comparison is performed as follows:

SLOTS ON:	128	64	32	16	8	4	2	1	= 23
SLOTS TO CHECK:	128	64	32	16	8	4	2	1	= 6
MATCHES:	128	64	32	16	8	4	2	1	= 6

Because the returned value equals 6, the computer sets C1 equal to C1 + 25, as instructed by the program.

Let's look at another example. Suppose you wanted to check to see if Missile #2 collided with Player #1. This collision involves slots 8 and 1. You check for this with the following program line:

```
IF (PEEK(53278) AND 9) = 9 THEN C2=C2+25
```

The computer might make the necessary comparison and find this:

SLOTS ON:	128	64	32	16	8	4	2	1	= 7
SLOTS TO CHECK:	128	64	32	16	8	4	2	1	= 9
MATCHES:	128	64	32	16	8	4	2	1	= 1

The returned value of "total matches" would be 1, which does not equal 9. Player #2's score would not be increased.

The AND command is very simple to understand and use once you've seen it work several times. All you need to do is supply the correct total of the slot numbers you want to check. The rest of the formula stays the same from collision detection to collision detection.

We have a COLLISION DETECTION tool that saves you the trouble of typing PEEK(53278) over and over again in your program. This tool will check memory location 53278 for you, and set the variable SS to the value of that memory location.

Add this tool to your program now:

```
300 REM:::COLLISION DETECTION
301 SS=PEEK(53278)
302 SF=PEEK(53279)
303 RETURN
```

(Line 302 checks for Sprite-to-Foreground collisions, which is discussed later in the chapter.)

You want the computer to be continuously checking for collisions, so that it will react each time one occurs. This is accomplished with a continuous loop. Type these program lines into your main routine:

```
27ØØ REM::::::GAME PLAY
271Ø GOSUB 3ØØ
29ØØ GOTO 271Ø
```

Notice that the loop begins at line 2710, and jumps straight to 2900. Line 2710 calls the COLLISION DETECTION subroutine. Upon returning from this tool SS holds the answer to whether one or more Sprite-to-Sprite collisions have occurred. Between lines 2710 and 2900, you may insert program lines that increase/decrease scores, explode spacecraft, or whatever else you deem appropriate for the collisions that have taken place. Then line 2900 returns the computer to line 2710 to look for more collisions.

There is one thing not yet addressed. The appropriate slots in memory location 53278 will turn on each time a Sprite-to-Sprite collision occurs. However, once you have checked this memory location and found the collision, you must turn those slots back *off*. If you don't, each time you PEEK into that location, you will be told about that same collision. Poking a 0 into memory location 53278 will turn off all of its slots. Add this subroutine to your program:

```
31Ø REM::::RESET COLLISION REGISTER
311 POKE53278,Ø: POKE53279,Ø
312 RETURN
```

Now you are set. You have a subroutine that checks for collisions and a subroutine that resets the collision register. Let's add some main routine lines that flash the player sprites on and off whenever they get hit. Again, the slot numbers assigned to each sprite are:

SLOT #:	128	64	32	16	8	4	2	1
SPRITE #:	7	6	5	4	3	2	1	0
ACTION VAR.:	—	—	—	T1	M2	M1	P2	P1

A collision between Missile #1 and Player #2 involves slots 4 and 2, totaling 6. A collision between Missile #2 and Player #1 involves slots 8 and 1, totaling 9. The program lines below check for these two collisions and take the necessary action.

```
28ØØ IF(SS AND 6)=6 THEN SP=1: GOSUB 5ØØØ
281Ø IF(SS AND 9)=9 THEN SP=Ø: GOSUB 5ØØØ
5ØØØ REM::::::COLLISION RESULT
5Ø2Ø FOR I = 1 TO 25
5Ø3Ø GOSUB 19Ø: GOSUB 18Ø
5Ø4Ø NEXT I
5Ø6Ø RETURN
```

Finally, add lines 5010 and 5050 so that memory location 53278 is reset before the loop is processed again, and the action sprites are all temporarily disabled while they flash:

```
5Ø1Ø P1=Ø: GOSUB 29Ø
5Ø5Ø GOSUB 31Ø: P1=1: GOSUB 29Ø
```

RUN the program and play the game. You should find that each time a spacecraft is hit by an opponent's missile, the spacecraft flashes on and off twenty-five times. The program doesn't check to see if the target sprite is hit, so nothing will happen when a missile collides with it.

Return to text mode and list lines 2700 through 5060 on your screen. Briefly, line 2710 checks the Sprite-to-Sprite collision register and places its total value in variable SS. Line 2800 checks SS to see if slots 4 and 2 (totaling 6) are turned on. If so, SP is set to 1 so that the correct sprite is affected, and a simple subroutine at 5000 executes. This subroutine flashes sprite SP twenty-five times.

Line 2810 checks SS to see if slots 8 and 1 (totaling 9) are turned on. If so, SP is set to 0 so that the correct sprite is affected, and the subroutine at 5000 is called. Notice that the last thing the loop at 5030 does is turn sprite SP *on*. If GOSUB 190 and GOSUB 180 were reversed here, the sprite that was hit would never be turned back on to continue the game.

Let's add program lines that detect when the target has been hit. Lines 2820 and 2830 check to see if either missile has hit the target. If so, SP is set to 4 (the target sprite), and the subroutine at 5000 is called. Add these program lines and run the program again.

```
282Ø IF(SS AND 24)=24 THEN SP=4: GOSUB 5ØØØ
283Ø IF(SS AND 2Ø)=2Ø THEN SP=4: GOSUB 5ØØØ
```

The game is identical, except that the target sprite now flashes when hit. Spend as much time as you like dodging and/or shooting your opponent, or taking pot shots at the target sprite.

The game can be easily changed whenever you like. The important thing is summing the correct slot numbers. (Don't add up *sprite* numbers by accident.) Determine the slot numbers of the two sprites whose collision you want to watch for. Add the slot numbers together, and insert them in a program line similar to this (replacing "x" with the total obtained by the slot numbers):

IF (SS AND x) = x THEN SP =...

Follow "SP =" with the sprite number that was hit, and then insert a GOSUB 5000 statement. The subroutine at 5000 can be customized to perform any "collision result" that you like, and there are many things that can be done. For example, you might try having the subroutine:

—disable all action sprites (P1=0: GOSUB 290).
—turn both missile sprites off (SP=2: GOSUB 190: SP=3: GOSUB 190)
—flash the sprite that was hit twenty-five times (same as before)
—enable all action sprites again (P1=1: GOSUB 290)

The missile sprites do not have to be (and should not be) turned back on. They will be automatically turned on by the press of the correct fire button.

TOOL 300 :::::: COLLISION DETECTION

```
300 REM:::COLLISION DETECTION
301 SS=PEEK(53278)
302 SF=PEEK(53279)
303 RETURN
```

What It Does: This tool PEEKs into two memory locations: 53278 and 53279. The value of memory location 53278 is placed in variable SS, and the value of memory location 53279 is placed in variable SF.

These two memory locations each have eight slots, numbered 128, 64, 32, 16, 8, 4, 2, and 1. Each sprite is assigned one slot from each memory location:

Sprite 0 — Slot 1
Sprite 1 — Slot 2
Sprite 2 — Slot 4
Sprite 3 — Slot 8
Sprite 4 — Slot 16
Sprite 5 — Slot 32
Sprite 6 — Slot 64
Sprite 7 — Slot 128

Whenever a sprite collides with another sprite, its slot will be turned "on" in memory location 53278, thus increasing the value that will be found when PEEKing into that memory location. Whenever a sprite collides with a foreground shape, its slot will be turned "on" in memory location 53279, thus increasing the value of that memory location. You can, by continually checking these memory locations, see exactly when a sprite collides with another sprite or with a foreground shape.

Example Use: This subroutine is called with a GOSUB 300 statement. You should place this GOSUB statement at the beginning of a loop that checks for specific collisions. A loop ensures that collisions are constantly monitored. Example program lines that form such a loop are:

```
2710 GOSUB 300
2800 IF(SS AND 24)=24 THEN SP=4: GOSUB 5000
2810 GOTO 2710
```

Technical Description: Memory location 53278 is a special register that contains information about Sprite-to-Sprite collisons. Each sprite has one bit in this register. Any time a collision occurs between two sprites, the two bits corresponding to the two involved sprites are set to "1".

Memory location 53279 is a similar register which contains information about Sprite-to-Foreground collisions. Any time a sprite collides with a foreground pixel, the bit corresponding to the involved sprite is set to "1" in this location.

By checking these locations very quickly, we can detect a collision almost as soon as it happens. If, for some reason, we aren't checking quickly enough and a collision has occurred and ended before we can check it, that collision will still register in the proper memory location. This ensures that we will never miss a collision.

Sprite-to-Foreground Collisions

The COLLISION DETECTION tool checks two memory locations. The first location stores Sprite-to-Sprite collisions. The second location, 53279, stores Sprite-to-Foreground shape collisions. You won't be able to detect the exact foreground shape with which a sprite collides. You will, however, be able to detect the exact sprite that has collided with a foreground shape.

When a sprite collides with a foreground shape, its slot number is turned "on" in memory location 53279. The sprite slot numbers are the same as before, so all you need to do is compare the slot numbers with a new memory location.

A GOSUB 300 statement will return the value of memory location 53279 for you. This value is stored in variable SF (Sprite-to-Foreground). You can compare this value to any sprite slot number and find out if that sprite collided with a foreground shape. You don't need to add slot numbers together when checking for foreground collisions. You aren't checking to see if two sprites collided with a foreground shape, only one. Thus, only one slot number needs to be considered in each AND statement.

Let's draw a border around the graphics screen. This border will be considered a foreground shape by the computer, and memory location 53279 will be affected if a sprite crosses over it. The program lines that will draw this border are given below, replacing those that plot the stars. Delete the program lines that draw the planet, if you entered them. (If you leave the stars or the planet on the screen, Sprite-to-Foreground collisions will happen all the time.)

```
2100 REM :::::::DRAW BORDER
2110 C=7: X1=Ø: Y1=Ø
2120 X2=319: Y2=Ø: GOSUB 5Ø
2130 X1=319: Y1=199: GOSUB 5Ø
2140 X2=Ø: Y2=199: GOSUB 5Ø
2150 X1=Ø: Y1=Ø: GOSUB 5Ø
```

We know that the target sprite will constantly cross over this border, so it won't be necessary to watch slot #16 to see when it turns on. We can, however, "penalize" the player sprites if they try to leave the screen. Player #1's sprite has a slot number of 1. We should, therefore, watch register 53279 to see when slot 1 turns on. Player #2's sprite has a slot number of 2, so this slot must be watched also. Try to figure out what AND statements are necessary to check for these two Sprite-to-Foreground collisions.

You're right if you came up with these program lines:

```
2840 IF(SF AND 1)=1 THEN SP=Ø:  GOSUB 5ØØØ
2850 IF(SF AND 2)=2 THEN SP=1:  GOSUB 5ØØØ
```

Type program lines 2840 and 2850 into your program. Next, type the following:

```
2710 GOSUB 3ØØ: IF SS>Ø OR SF>Ø THEN 28ØØ
2720 GOTO 271Ø
```

This prevents the computer from reading lines 2800 through 2850 when it is evident that no collisions have taken place (Tool 300 returned a value of 0 for both collision registers).

RUN the program. The stars are gone, and in their place a bright yellow border is plotted around the graphics area. This border is made up of foreground pixels, and is hence considered to be a foreground shape by the computer. Try "sneaking" a player off the screen. The sprite will be able to move off the edge, but not without first facing the consequences. It will flash as it tries to leave the screen, and again when it tries to return to the screen.

Imagination is the biggest factor in successfully using Sprite-to-Sprite and Sprite-to-Foreground collisions. You should spend time designing foreground shapes that present obstacles to your players. For example, you might allow one player sprite to go unharmed when it collides with foreground objects, but blow up the other player sprite when it collides with foreground shapes.

The "sprite priority over shapes" feature is an even bigger consideration now. By setting this feature properly, you can permit one player (or both) the luxury of "hiding" behind foreground shapes as part of the game. The rewards, consequences, and game strategy are all up to you—all we can provide are the tools to work with.

Write the sprite slot numbers on a piece of scratch paper. On that same paper, write down the following:

Sprite-to-Sprite Collisions:
GOSUB 300
IF (SS AND x) = x THEN ...
(where "x" equals total of two sprite slot numbers)

Sprite-to-Foreground Collisions:
GOSUB 300
IF (SF AND x) = x THEN ...
(where "x" equals slot number of sprite)

Scoring and Special Effects

The remaining tools in this chapter deal with the visual and audio effects that can be created when a collision occurs. There is not enough room in one chapter to cover all the possibilies; but we will show you enough to give you a good start.

Game Suspension, Scoring, and Game Continuation

The task of keeping a player's score is a relatively easy one, and can be handled in the main routine. We will assign each player a variable: Player #1 will have variable C1, and Player #2 will have variable C2. Every time a missile sprite collides with another sprite we will increase the value of one of these variables.

The number of points awarded per hit can vary from game to game, or even from player to player. Generally, it's harder to hit an opponent than it is the target sprite. Let's start by awarding twenty-five points for hitting an opponent, and ten points for hitting the target. We'll also deduct points from any player who crosses the screen's border. Change lines 2800 through 2850 to the following:

```
28ØØ IF(SS AND 6)=6 THEN C1=C1+25: SP=1: GOSUB 5ØØØ
281Ø IF(SS AND 9)=9 THEN C2=C2+25: SP=Ø: GOSUB 5ØØØ
282Ø IF(SS AND 24)=24 THEN C2=C2+1Ø: SP=4: GOSUB 5ØØØ
283Ø IF(SS AND 2Ø)=2Ø THEN C1=C1+1Ø: SP=4: GOSUB 5ØØØ
284Ø IF(SB AND 1)=1 THEN C1=C1-15: SP=Ø: GOSUB 5ØØØ
285Ø IF(SB AND 2)=2 THEN C2=C2-15: SP=1: GOSUB 5ØØØ
```

Increasing and decreasing the value of variables C1 and C2 can easily keep track of the score. The next step is to make it easy for players to check the score.

A tool called SUSPEND GAME will temporarily save the game being played and take the players to text mode. This tool can be "called" based on the condition that a particular key has been pressed. Once in text mode, the playing scores can be displayed on the screen.

Another tool, called CONTINUE GAME, returns the players to the game exactly where they left off. This tool can be called based on the condition that *any* key is pressed. You will see how this works once you add the tools to your tool kit:

```
32Ø REM::::::SUSPEND GAME
321 P1=Ø: GOSUB 29Ø: TP=PEEK(53269)
322 POKE 53269,Ø: GOSUB2Ø
323 RETURN
33Ø REM::::::CONTINUE GAME
331 GOSUB 1Ø: POKE 53269,TP: P1=1: GOSUB 29Ø
332 RETURN
```

Look at the program lines below. The characters "CHR$(133)" in line 2730 are codes representing the top function key (f1) on the right-hand side of your keyboard. Every key on the keyboard has a CHR$ code to represent it. In line 2740, the **CLR/HOME** key is represented by "CHR$(147)." Add lines 2720 through 2790 to your program.

```
272Ø GET A$
273Ø IF A$ <>CHR$(133) THEN 271Ø
274Ø GOSUB 32Ø: PRINT CHR$(147)
275Ø PRINT "PLAYER 1: "C1: PRINT "PLAYER 2: "C2
276Ø PRINT: PRINT "PRESS STOP TO QUIT; OR"
277Ø PRINT "PRESS ANY KEY TO CONTINUE."
278Ø GET A$: IF A$= "" THEN 278Ø
279Ø GOSUB 33Ø: GOTO 271Ø
```

Think about what these program lines will do. Line 2720 checks the keyboard for a keypress. Line 2730 sends the computer back up to line 2710 if the **f1** key was not pressed. However, if **f1** was pressed, line 2740 is reached. This line suspends the game (GOSUB 320), and clears the text screen (PRINT CHR$(147)—or press the **CLR/HOME** key).

Lines 2750 through 2770 print general information on the text screen, as well as the current scores (C1 and C2). Line 2780 causes the computer to again check to see if a key is pressed. If no keys are pressed, line 2780 is read again. Line 2790 will only be read if a key (any key) is pressed. This line calls Tool 330, which returns the players to the game.

RUN the program, and spend a few moments racking up points for each side. Remember: scores are increased by hitting an opponent or the target sprite, and scores are decreased by crossing the screen's border lines. Press the **f1** key when you want to check the scores.

The text screen should then be displayed, with a message giving scores for Player 1 and Player 2. In addition, a message stating how to return to the game (PRESS ANY KEY), or how to quit (PRESS RUN/STOP) should be present. A press of the **RUN/STOP** key will return you to the program. A press of any other key will return you to your game.

We choose to call the SUSPEND GAME tool based on a press of the **f1** key. You may change the condition to any other that you like. For instance, when one player reaches a certain number of points, you could automatically call this tool and display the scores (warning the underdog of his plight).

You could also change what the computer does when this tool is called. For example, you could exit the game entirely instead of only temporarily. You could also display different messages on the text screen, depending on the score of each player (e.g., GOOD JOB!, BETTER LUCK NEXT TIME, PLAY AGAIN?, etc).

TOOL 320 :::::: SUSPEND GAME

```
32Ø REM::::::SUSPEND GAME
321 P1=Ø: GOSUB 29Ø: TP=PEEK(53269)
322 POKE 53269,Ø: GOSUB 2Ø
323 RETURN
```

What It Does: This tool will save the current form of your action sprite game, and take you to text mode. The program will still be running, but you can display any messages or scores you want to on the text screen. Pressing **RUN/STOP** would end the program.

Example Use: This tool should be called based on meeting a certain condition; that is, a specific keypress. We chose a press of the **f1** key (far right side of keyboard) as the condition for calling this tool. Example program lines that suspend the game based on a press of the **f1** key are:

```
272Ø GET A$
273Ø IF A$ <> CHR$(133) THEN 271Ø
274Ø GOSUB 32Ø
```

Technical Description: P1=0: GOSUB 290 disables the action sprites. TP=PEEK(53269) saves the "sprite enable" byte. This allows the sprites which were turned off to be easily restored to their previous positions. POKE 53269,0 turns off all the sprites, and GOSUB 20 turns on the text screen. This routine allows the score to be displayed without ending the game.

TOOL 330 :::::: CONTINUE GAME

```
33Ø REM:::::CONTINUE GAME
331 GOSUB 1Ø: POKE 53269,TP: P1=1: GOSUB 29Ø
332 RETURN
```

What It Does: This tool allows you to resume a suspended game. It will take you from the text screen back to the game. The game will be set up exactly as it was left off. This tool only works while the program is running. Once the game *and* program are abandoned, you cannot return to the game where you left off.

Example Use: This tool should be called based on the condition that any key on the keyboard is pressed. That way, players can easily return to their game. Before calling this tool, any scores or other messages should have been printed in the text screen for the players to see. Example program lines that would call this tool based on a press of any key are:

```
2780 GET A$: IF A$="" THEN 2780
2790 GOSUB 330
```

Technical Description: GOSUB 10 turns on the graphics screen. POKE 53269,TP restores the sprites which were turned off with the "suspend game" subroutine. P1=1: GOSUB 290 re-enables (i.e., turns on) the interrupt which was also turned off with the "suspend game" subroutine.

Creating Sound

There is much we could provide in the way of sound. Indeed, there are entire books written on the subject of creating and controlling sounds on the Commodore 64. Unfortunately, the space we have to devote to this topic is not adequate to cover the whole subject. We knew that the video game wouldn't be the same without at least one noise, so, rather than skip the subject, we are going to give you tools that control *one* sound. This sound resembles an explosion.

The SOUND ON and SOUND OFF tools are given below. Tool 340 activates an explosion noise when it is called. Tool 350 turns the explosion noise off. Add these tools to your tool box.

```
340 REM::::::SOUND ON
341 POKE54273,2: POKE54277,255: POKE54278,255
342 POKE54276,129: POKE54296,15
343 RETURN
350 REM::::::SOUND OFF
351 FOR I=15 TO 0 STEP-.1: POKE54296,I:NEXTI
352 RETURN
```

The only thing you need to do with these tools is decide the appropriate time to call each one. Obviously, the sound should be turned on after a significant collision has occurred. By "significant," we mean one deserving of consequences. The target sprite colliding with the foreground border is not a significant collision. However, Missile #1 colliding with Player #2 is definitely significant, and an explosion would be appropriate.

The penalty of a significant collision is currently being handled by the subroutine at 5000. This subroutine will flash the wounded sprite on and off twenty-five times. It is during this period of flashing that the sound should be activated. Once the penalty has been paid, the flashing should stop, the sound should be turned off, and the game should continue running. Two short program lines will add this new aspect to the game:

```
5Ø15 GOSUB 34Ø
5Ø45 GOSUB 35Ø
```

Immediately after a significant collision has been found and subroutine 5000 called, the explosion noise is activated. Then, the sprite that was hit flashes on and off. Finally, line 5045 turns the sound off before returning to the game (main routine).

RUN the program; but before you start shooting, turn the volume *down* on your monitor. Otherwise, you and anyone else in the vicinity might be in for a rude surprise.

Shoot at the target, adjusting the volume as you do.

Advanced sprite features are generally no more complex to operate than inserting a GOSUB statement in the program. Hence, there are not many hints we can give you regarding program problems. If you are experiencing difficulties, make sure you are calling the right subroutine, and that the subroutine was typed correctly.

TOOL 340 :::::: SOUND ON

```
34Ø REM::::::SOUND ON
341 POKE54273,2: POKE54277,255: POKE54278,255
342 POKE54276,129: POKE54296,15
343 RETURN
```

What It Does: This tool will cause a sound to be emitted from the monitor connected to your Commodore 64. This sound will resemble a small explosion, and will remain "on" until Tool 350 is called.

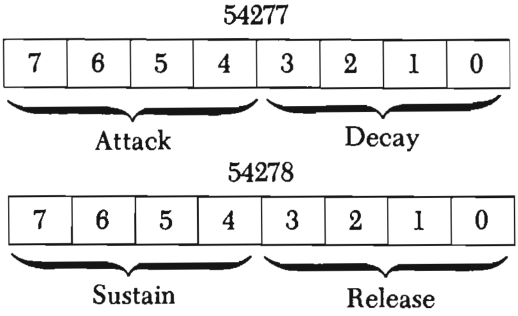
Example Use: A GOSUB 340 is all that is necessary to use this tool. It is a good idea to call this tool immediately following a significant collision. You should be prepared to adjust the volume on your monitor whenever you run a program that makes use of this tool.

Technical Description: Memory location 54273 is used as your computer's sound frequency control. Putting a low number (e.g., "2") in this location will produce a deep "booming" sound, while putting a higher number in this location (e.g., "255") will produce a higher pitched "snapping" sound similar to a firecracker.

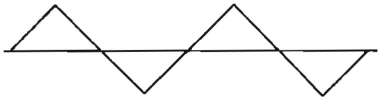
Memory locations 54277 and 54278 are used to define the "envelope" of the sound. There are four parts to an envelope, and each memory location handles two of these parts. Those four parts are "attack," "decay," "sustain," and "release."

"Attack" is the amount of time it takes the volume to go from 0 to maximum volume. "Decay" is the amount of time it takes to go from maximum volume to a specified intermediate volume level which lies between "0" volume and the maximum volume. "Sustain" is the specified intermediate volume level, and "release" is the amount of time it takes to go from the specified "sustain" level to "0" volume.

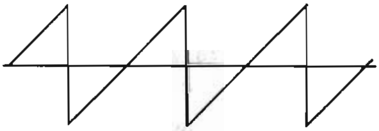
The memory locations are allocated in the following manner:



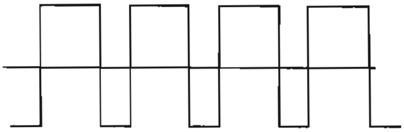
Memory location 54276 controls the waveform of the sound. There are four possible types of waveforms:



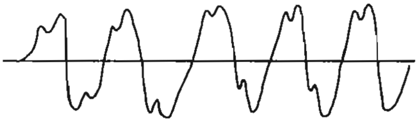
TRIANGLE



SAWTOOTH



PULSE



NOISE

The first three waveforms are used to simulate musical instruments. The last waveform, noise, is used to simulate special effects such as surf, explosions, jets, cars, etc. Since we are simulating the sound of a collision, the "noise" waveform is of special interest to us.

Memory location 54296 controls volume. By entering a value which ranges from 0 to 15 in this location you can raise and lower the volume without adjusting the volume of your television. "0" volume is the lowest volume level, while "15" is the maximum volume level.

TOOL 350 :::::: SOUND OFF

```
350 REM::::::SOUND OFF
351 FOR I=15 TO Ø STEP-.1: POKE54296,I: NEXT I
352 RETURN
```

What It Does: This tool will turn off the noise created by Tool 340.

Example Use: A GOSUB 350 is all that is necessary to use this tool. Most often, this tool will be called at the end of an explosion just before returning the players to their game.

Technical Description: This subroutine gradually lowers the volume of the sound after an explosion. This gives a more realistic, life-like quality to our explosions.

Creating Crash Sprites

This final section will show you how to make an explosion more visually exciting. You already have the "sound" of an explosion, and even the "flashing" effect of an explosion, but you are still in need of the "look" of an explosion. When the target sprite or a player sprite is hit by a missile, that sprite needs to be temporarily replaced with something that resembles an exploding spacecraft.

That is what we will be doing here. In fact, we will be creating two "crash" sprites that, when alternating between the two, present a more accurate picture of an explosion. This will be handled by the final tool in this book: Tool 360 COLLISION PENALTY.

The COLLISION PENALTY tool provides a quick way to make a fancy explosion on the screen. You may choose to use it in some games, and not in others. You may even decide to add to this tool, making the penalty more severe for the unfortunate victim of a missile. In any event, this tool is handy and useful. It performs six different steps:

(1) Disables all action sprites;
(2) Turns on the explosion sound;
(3) Flashes Sprites 5 and 6, alternating between the two, fifty times;
(4) Turns off the sound;
(5) Enables any action sprite it disabled in step (1); and
(6) RETURNs the computer to the main routine (game).

You can see that this takes care of much in the way of penalties for the players. Type this tool as follows:

```
360 REM::::::COLLISION PENALTY
361 P1=Ø: GOSUB 29Ø: GOSUB 34Ø
362 FOR I=1 TO 5Ø: POKE18424+SP,5: POKE18424+SP,6: NEXT
363 GOSUB 35Ø: POKE 18424+SP,SP: P1=1: GOSUB 29Ø
364 GOSUB 31Ø
365 RETURN
```

This tool makes use of Sprites 5 and 6. It will *not* place sprites 5 and 6 at the "scene of the accident," flashing them on and off when it does. Instead, it has the computer use these two descriptions to display the sprite that was hit. This is important, so we will go over it again.

The computer continually looks at a sprite's DATA stored in memory as it places and moves the sprite around the screen. Take Sprite #0, for example. The program currently has the computer look at CRAFT1 data to define this sprite. Now, suppose Sprite #0 is hit with a missile. Your new tool will redirect the computer, telling it to look at Sprite #5's data to define Sprite #0. This changes the appearance of Sprite #0 for a moment (it does *not* change the color of the sprite). Next, the tool has the computer look at Sprite #6's data to define Sprite #0. The appearance of Sprite #0 will again change. The computer is directed back and forth fifty times between Sprite 5's and Sprite 6's data to define the sprite that was hit. At the end of the subroutine, the sprite returns to its original form.

The importance of all of this is that we are not locating the collision and moving Sprites 5 and 6 to it. Instead, we merely redefine what a sprite looks like, using the descriptions of Sprites 5 and 6 to do so. The explosion will always be the same color as the sprite that was hit, because it *is* the sprite that was hit.

The DATA statements for our crash sprites follow. This involves a bit of typing, but the game is enhanced enough to justify the work.

```
1198 DATA "CRASH1",31,Ø
1200 DATA   Ø, 48,  3
1202 DATA   Ø, 6Ø, 12
1204 DATA  6Ø,195, 6Ø
1206 DATA  51,24Ø,2Ø4
1208 DATA  48, 48, 12
121Ø DATA 24Ø, 48, 6Ø
1212 DATA 192, 6Ø,2Ø4
1214 DATA   Ø,195,2Ø4
1216 DATA 2Ø4,2Ø4,2Ø4
1218 DATA 2Ø7, 63, 12
122Ø DATA   3, 6Ø,195
1222 DATA   3, 51,192
1224 DATA  51,  3,195
1226 DATA  51, 6Ø, 6Ø
1228 DATA 2Ø4,2Ø4, 12
123Ø DATA  63, 12, 48
1232 DATA 194, 48, 48
1234 DATA 192, 48, 12
1236 DATA 255,  Ø, 12
1238 DATA   3, 15, 12
124Ø DATA   3, 48,24Ø
1242 DATA   .,  .,  .
```

```
1244 DATA "CRASH2",31,Ø
1246 DATA   Ø,  Ø,  Ø
1248 DATA   Ø,  Ø,  Ø
125Ø DATA   Ø, 6Ø,  Ø
1252 DATA  12, 15, 48
1254 DATA  15,2Ø7, 6Ø
1256 DATA  15,2Ø7,192
1258 DATA  63,195, 48
126Ø DATA 255, 6Ø, 48
1262 DATA  51, 51, 48
1264 DATA  48,192,24Ø
1266 DATA 252,195, 6Ø
1268 DATA 252,2Ø4, 63
127Ø DATA 2Ø4,252, 6Ø
1272 DATA 2Ø4,195,16Ø
1274 DATA  51, 51,24Ø
1276 DATA   Ø,243,16Ø
1278 DATA  61,2Ø7,16Ø
128Ø DATA  63,2Ø7,24Ø
1282 DATA   Ø,255,24Ø
1284 DATA   Ø,24Ø,24Ø
1286 DATA   Ø,2Ø7,  Ø
1288 DATA   .,  .,  .
```

You need to set up the crash sprites, but this only involves retrieving one as Sprite 5, and the other as Sprite 6. The color and placement of these sprites do not matter, because these sprites are never placed on the screen. Their descriptions are needed by the tool, though, so they must be retrieved. Type the following:

```
2500 REM::::::SET UP CRASH SPRITES
2510 SP=5: SE$="CRASH1": GOSUB 810
2520 SP=6: SE$="CRASH2": GOSUB 810
```

Finally, you must change the program so that Tool 360 is called each time a significant collision occurs. This involves lines 2800 through 2850:

```
2800 IF(SS AND 6)=6 THEN C1=C1+25: SP=1: GOSUB 360
2810 IF(SS AND 9)=9 THEN C2=C2+25: SP=0: GOSUB 360
2820 IF(SS AND 24)=24 THEN C2=C2+10: SP=4: GOSUB 360
2830 IF(SS AND 20)=20 THEN C1=C1+10: SP=4: GOSUB 360
2840 IF(SB AND 1)=1 THEN C1=C1-15: SP=0: GOSUB 360
2850 IF(SB AND 2)=2 THEN C2=C2-15: SP=1: GOSUB 360
```

The only change in the above lines is that GOSUB 5000 has been replaced with GOSUB 360. The custom subroutine at line 5000 may be erased entirely, or you may keep it for other purposes later.

RUN the program. The COLLISION PENALTY tool is called each time a missile strikes a player sprite or the target sprite. At that time, the explosion sound will be turned on, and what appears to be a puff of smoke will flash on the screen. This "smoke" is created by alternating between two "crash" descriptions for the sprite that was hit. Finally, the sound will turn off and the game will resume as before.

That completes the "game construction tools" for this book. There are many, many things that can be done with these tools to increase the challenge and excitement of the game. Take a few moments to come up with design ideas to try out. For starters, you might:

—modify the game for a single player

—add foreground shapes for the players to hide behind (you must remove the penalty for colliding with foreground shapes if you do this)

—provide "bonus points" to the player who hits the target sprite every 10th time it is hit

—turn the player sprites "off" once in a while (they will still move and shoot—you just can't see where they are!)

—increase the speed (VE variable) after one or both players achieve a certain score, or after the target sprite has been hit every "x" number of times

—decrease the size or number of "bullets" in a missile as the game progresses

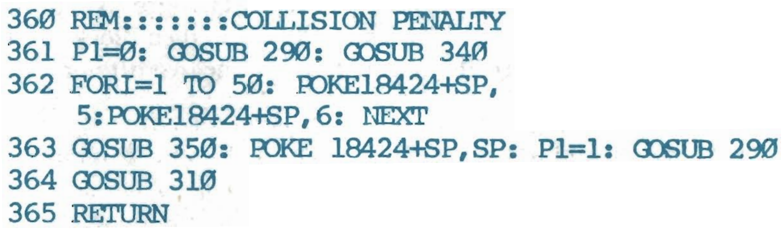
—decrease the size of the target sprite as the game progresses (this entails retrieving a new description for that sprite each time a switch is wanted).

Most of these suggestions will involve a "counter" variable. For example, you might have "SC=SC+1" in line 2860. Then, in line 2870, you could have "IF SC=10 THEN VE=VE-1: GOSUB 290: 3C=0". This would gradually increase the speed of the game.

You could turn a player sprite "off" temporarily with a program line something like: "IF C1 >200 AND <250 THEN SP=0: GOSUB 190". This could be followed on the next line with, say, "IF C1 > 249 THEN SP=0: GOSUB 180". This would turn Player #1's sprite off as soon as he reached a score greater than 200. This sprite would be invisible on the screen (still operating in every other respect) until Player #1's score reached at least 250.

Jot down ideas as they come to you. Some will be implausible or maybe even impossible, but many will be worth employing. We encourage you to take whatever time is needed to formulate, refine, and program your game features. The rewards are many.

TOOL 360 :::::: COLLISION PENALTY



```
360 REM::::::COLLISION PENALTY
361 P1=Ø: GOSUB 29Ø: GOSUB 34Ø
362 FORI=1 TO 5Ø: POKE18424+SP,
     5:POKE18424+SP,6: NEXT
363 GOSUB 35Ø: POKE 18424+SP,SP: P1=1: GOSUB 29Ø
364 GOSUB 31Ø
365 RETURN
```

What It Does: This tool performs all of the following:

(1) Disables all action sprites;
(2) Turns on the explosion sound;
(3) Alternates displaying Sprite 5's and 6's description at the collision location, a total of fifty times;
(4) Turns off the sound;
(5) Enables any action sprite it disabled in step (1); and
(6) RETURNs the computer to the main routine (game).

Example Use: This tool is called with a GOSUB 360 statement. Usually, it will be called just after a significant collision has been detected.

Technical Description: The secret to this tool can be found in lines 362 and 363. Memory locations 18424 through 18431 hold the sprite pointers. Each of these memory locations "points" to the place in memory where a specific sprite's defining data has been stored.

There are 256 "blocks" in Bank 1 (where we are storing our graphics data). These blocks are numbered 0 through 255. Although most of these blocks were already in use, the first eight blocks (numbered 0 -7) were free. This made it simple for our RETRIEVE A SPRITE tool to assign one block to each sprite: Sprite #0 is assigned Block #0, Sprite #1 is assigned Block #1, Sprite #2 is assigned Block #2, and so on.

Memory locations 18424 through 18431 store which sprite is assigned which block number. Memory location 18424 stores the information that Sprite #0's data is in Block #0. Memory location 18425 stores the information that Sprite #1's data is in Block #1. Memory location 18426 stores the information that Sprite #2's data is stored in Blocks #2. And so on.

By switching the block that one of these memory locations point to, we can quickly change the definition of a sprite. For example, by changing memory location 18424 (Sprite #0's pointer) so that it points to Block #6, we can redefine the appearance of Sprite #0: it will take on the appearance of Sprite #6. Likewise, by changing memory location 18427 (Sprite #3's pointer) so that it points to Block #7, we redefine the appearance of Sprite #3: it will take on the appearance of Sprite #7.

You see that more than one memory location can point to the same block of data to define a sprite. In the COLLISION PENALTY tool, a "penalized" sprite's memory location will be changed so that it points to Block #5 (storing Sprite #5's description), and then Block #6 (storing Sprite #6's description), alternating between the two fifty times.

Summary

Congratulations! You are now a graduate of our "class" on Commodore 64 Advanced Graphics. We began in Chapter 1 by reviewing how to PLOT A POINT. Since then, you have mastered:

—plotting lines (Tool 50)
—retrieving shapes (Tool 800)
—drawing shapes (Tool 90)
—painting shapes (Tool 60)
—clipping shapes (Tool 70)
—translating shapes (Tool 140)
—scaling shapes (Tool 150)
—rotating shapes (Tool 160)
—the complete list of beginning and advanced sprite graphics...

That was no small task. But by storing each new graphics technique as a subroutine "tool," you eliminate many of the repetitious and difficult programming chores. Take the time to save your final program under "CHAPTER 8", or "GAME". We then suggest you run the ZAP routine, and save the complete list of tools under the filename "TOOLKIT".

We again remind you to browse through the appendices. You'll be glad you did. Also, a "Tool Kit Reference Card" can be found just inside the back cover. This card briefly reviews each tool, and is meant as a handy aid when you are programming your own artwork and arcade-like games. For a more thorough discussion of each tool, refer to chapter summaries and tool boxes. A review of this chapter's tools is given below.

Tool 290 ::: Hook Up Action Sprites

This tool is used to enable/disable action sprites, as well as to select a method of control (keyboard/joystick) and set the action sprite speed. Seven variables need to be set when this tool is called:

VE = 0 *to* 65 (VElocity of speed)

KB = 1 (hook up player sprites to KeyBoard) *or*
0 (hook up player sprites to joystick)

P1 = 1 (use Sprite #0 as Player 1 action sprite) *or*
0 (use *all* sprites as non-action sprites)

P2 = 1 (use Sprite #1 as Player 2 action sprite) *or*
0 (use Sprite #1 as non-action sprite)

M1 = * (use Sprite #2 as Player 1 missile sprite, shooting it in the direction of "*") *or*
0 (use Sprite #2 as non-action sprite)

M2 = * (use Sprite #3 as Player 2 missile sprite, shooting it in the direction of "*") *or*
0 (use Sprite #3 as non-action sprite)

T1 = * (use Sprite #4 as target sprite, moving it in the direction of "*") *or*
0 (use Sprite #4 as non-action sprite)

*The target sprite and the missile sprites are enabled by setting their variables to one of eight possible direction codes. The codes are: 1 (up), 2 (down), 4 (left), 8 (right), 9 (up/right), 10 (down/right), 6 (down/left), and 5 (up/left).

Action sprites must be "set up" before they are enabled. The target sprite and the player sprites are set up by:

(1) Setting the sprite variable (SP=?);

(2) Retrieving a sprite description (SE$=?: GOSUB 810);

(3) Setting the sprite's color (C=?: GOSUB 260);

(4) Calling any other sprite feature desired;

(5) Placing the sprite on the screen (X=?: Y=?: GOSUB 270);

(6) Turning the sprite "on" (GOSUB 180).

Missile sprites are set up by following only steps (1) through (4) above.

The keys that are "active" under keyboard control are:

	Player #1:	Player #2:
UP:	R	P
DOWN:	C	.
RIGHT:	F	:
LEFT:	D	L

Tool 300 ::: Collision Detection

This tool PEEKs memory location 53278 and places its value into variable SS (Sprite-to-Sprite collisions). In addition, it PEEKs memory location 53279 and places its value in variable SF (Sprite-to-Foreground collisions).

You can check the value of SS to see if two specific slot numbers were on in location 53278 by using the following form of program line:

IF (SS AND x) = x THEN ...

Replace "x" with the sum of the two sprite slot numbers to check. The eight sprite slot numbers are as follows:

Slot 128 (Sprite #7)
Slot 64 (Sprite #6)
Slot 32 (Sprite #5)
Slot 16 (Sprite #4)
Slot 8 (Sprite #3)
Slot 4 (Sprite #2)
Slot 2 (Sprite #1)
Slot 1 (Sprite #0)

You can check the value of SS to see if two specific slot numbers are on in shape by using the following form of program line:

IF (SF AND x) = x THEN ...

Replace "x" with the slot number of the sprite to check.

Tool 310 ::: Reset Collision Register

This tool POKEs a 0 into both memory locations 53278 and 53279. It should be called just after all collisions have been checked for in the main routine.

Tool 320 ::: Suspend Game

This tool suspends the current form of the game, taking the players to text mode. The program is still running at this point.

Tool 330 ::: Continue Game

This tool returns the players to their game, exactly as they left off.

Tool 340 ::: Sound On

This tool activates one sound of the Commodore 64's. This sound resembles a small explosion.

Tool 350 ::: Sound Off

This tool turns off the sound created by calling Tool 340.

Tool 360 ::: Collision Penalty

This tool disables all player sprites, calls the Sound On tool, and begins flashing Sprite #5's and Sprite #6's descriptions at the collision location (alternating between the two). These sprite descriptions flash fifty times, and then the sound is turned off and the game resumed.

APPENDIX A: ADDITIONAL PROGRAMS

PRINT A PICTURE and SAVE A PICTURE are two separate programs which will be of great help to you. Because these are *programs*, and not tools, you will need to load and run each program before it can be used.

Print a Picture

The PRINT A PICTURE program will print a copy of the graphics screen to a VIC-1525 printer. If you have such a printer, type in this program as:

```
1ØØ OPEN1,4: BA=24888
11Ø A$=CHR$(15)+CHR$(16)+"2Ø "+CHR$(8)
12Ø FORJ=ØTO45: IF(JAND7)>ØTHEN BA=BA-8
13Ø BY=BA:PRINT#1,A$;
14Ø B1%=JAND7:B2%=8-B1%:FOR K=Ø TO 199
15Ø T=PEEK(BY)*2^B1%AND127
16Ø B=INT(PEEK(BY+8)/2^B2%)
17Ø PRINT#1,CHR$(128+T+B);
18Ø BY=BY+1:IF(KAND7)=7 THEN BY=BY+312
19Ø NEXTK:PRINT#1:NEXTJ:CLOSE1
```

Now, save these program lines under the filename "PRINTPIC".

To use this program, take the following steps:

(1) Connect the printer as described in the VIC-1525 User's Manual.

(2) Check to be sure you have paper and that the printer is on.

(3) Check to make sure there is a picture on the graphics screen.

(4) Load "PRINTPIC",8 if you are using a disk drive, or "PRINTPIC",1 if you are using a cassette recorder.

(5) Type: RUN and press **RETURN**.

This program takes advantage of the VIC-1525 printer's graphics ability. By chopping the graphics screen into 7 x 7 pixel segments, this program can make the VIC-1525 print each segment as a character. When all of the sections are printed, you end up with a complete picture.

Save a Picture

The SAVE A PICTURE program is used to save a picture created by a graphics program. That picture can then be easily and quickly loaded into memory at any time.

The program lines for the SAVE A PICTURE routine are shown below:

```
1ØØ INPUT"ENTER FILENAME";FI$
11Ø INPUT"ENTER 8 FOR DISK, OR 1 FOR CASSETTE";DE
12Ø SYS 57812 FI$+".PIC",DE
13Ø POKE 174,64: POKE 175,127: POKE 193,Ø: POKE 194,96
14Ø SYS 62954
15Ø SYS 57812 FIL$+".COL1",DE
16Ø POKE 174,232: POKE 175,71: POKE 193,Ø: POKE 194,68
17Ø SYS 62954
18Ø SYS 57812 FIL$+".COL2",DE
19Ø POKE 174,232: POKE 175,219: POKE 193,Ø: POKE 194,216
2ØØ SYS 62954: END
```

Now, save this program under the filename “SAVEPIC”.

To save a picture that currently resides in memory, simply load “SAVEPIC”,8 if you own a disk drive, or “SAVEPIC”,1 if you own a cassette recorder, and then type: RUN **RETURN**.

The computer should respond with the prompt “ENTER FILENAME”. You must enter the filename you want to attach to the picture. The name can be a maximum of twelve characters in length. An important thing to understand is that the computer will save your picture in three different files. The first file contains the pixel patterns that make up the picture. The second and third files contain the colors that make up the picture. Each picture will have a filename that begins with whatever you type now. However, your first file will have “.PIC” appended to the name, and the second and third files will have “.COL1” and “.COL2” appended to the name.

Once you’ve entered the filename, the computer will respond 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 will be no way to directly verify that your picture was stored on your disk/tape. The only way to find out is to go through the normal sequence of loading a picture.

> IMPORTANT: Loading picture files affects your program listings. Essentially, your computer loses its program “orientation” when a picture file is loaded. Whenever you load a picture you *must* carefully follow the instructions below.

If you have a program in memory:

(1) Save your program.
(2) Load your picture as explained below.
(3) Re-load your program.

If you do not have a program in memory:

(1) Load your picture as explained below.
(2) Type: NEW **RETURN** or load a program.

To load the picture you must load three files, two for the picture's colors, and one for its pixel patterns. Do this in the following manner:

Disk Drive Users Type:

```
LOAD "FILENAME.PIC",8,1
LOAD "FILENAME.COL1",8,1
LOAD "FILENAME.COL2",8,1
```

Cassette Recorder Users Type:

```
LOAD "FILENAME.PIC",1,1
LOAD "FILENAME.COL1",1,1
LOAD "FILENAME.COL2",1,1
```

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

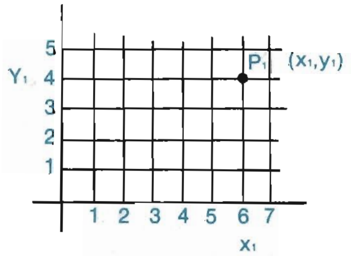
The above files are loaded almost the same as program files, except that a ",1" needs to be appended to the end. The ",1" that you type at the end of your keystrokes tells the computer to load the picture on the screen instead of as a BASIC program. Thus, the ",1" should be typed regardless of whether 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. When all files have been loaded, type GOSUB 10 and press **RETURN** to see the picture.

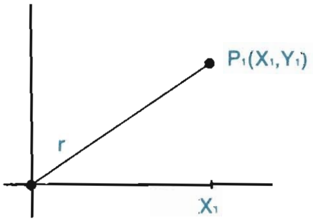
APPENDIX B: ROTATION MATH

Rotating an object appears to be an almost magical capability. In fact, it is a fairly simply mathematical process. The key to understanding rotation lies in converting the points you wish to rotate from Cartesian to polar coordinates. We'll now demonstrate the steps you'll need to take in order to convert your points from the Cartesian to the polar system, rotate those points, and then convert your new points back to the Cartesian system.

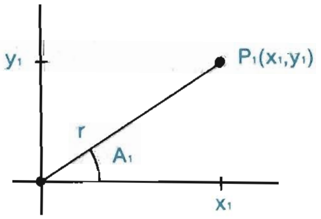
The following diagram shows a point "P1" as it would be displayed in the Cartesian coordinate system:



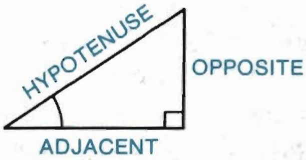
Since P1 is a real point on the grid, we know that it is a set distance from the origin. We will use the variable "r" to represent this distance, as shown in the diagram below. We won't measure that distance right now; but notice that we could if we wanted to do so.



Notice also that there is a set angle between "r" and the X-axis. We'll use the variable "A1" to represent this angle, as shown in the diagram below.



You are now in the world of polar coordinates. We haven't calculated the components of our polar coordinate diagrams (i.e., "r" and "A1"), but we know that those components can be measured. We can reformulate these components as a right triangle. The diagram shown below illustrates the concepts that we have been working with as they would be used in geometric and trigonometric functions:



From basic trigonometry we can conclude that:

α = A1
Hypotenuse = r
Adjacent = X1
Opposite = Y1

We can therefore conclude that:

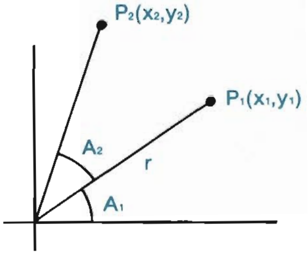
SIN A1 = Y1/r and COS A1 = X1/r

After multiplying by "r", we arrive at equations 1 and 2:

(1) Y1 = rSIN(A1) and (2) X1 = rCOS(A1)

Notice that we now have a way to convert polar coordinates to Cartesian coordinates. We will save these equations until they are needed.

If we wish to rotate point P1 "A2" degrees to produce a resulting point "P2" at (X2,Y2), we can again use equations 1 and 2 to determine the proper X,Y Cartesian coordinates. The diagram shown below illustrates this process.



Notice that "r" doesn't change, regardless of the rotation distance. Looking back at equation 1, if Y1 = rSIN(A1), then we can conclude that Y2 = rSIN (A1 + A2). Similarly, if X1 = rCOS(A1), then we can conclude that X2 = rCOS (A1 + A2).

It can be proven mathematically that:

(3) SIN(A1 + A2) = SIN(A1)COS(A2) + COS(A1)SIN(A2)
and
(4) COS(A1 + A2) = COS(A1)COS(A2) -SIN(A1)SIN(A2)

By combining what we know from equations 1 and 2 with the equations just presented, we can state that:

X2 = rCOS(A1 + A2), and therefore
X2 = r(COS(A1)COS(A2) -SIN(A1)SIN(A2)).

By distributing the variable "r" across the equation, we can state that X2 = rCOS(A1)COS(A2) -rSIN(A1)SIN(A2).

Similarly, we can state that: Y2 = rSIN(A1 + A2), and by substituting equation 3 for the expression SIN(A1 + A2), we can conclude that: Y2 = r(SIN(A1)COS(A2) + COS(A1)SIN(A2)).

By distributing the variable "r" across the equation we can state that Y2 = rSIN(A1)COS(A2) + rCOS(A1)SIN(A2).

If all this theory looks intimidating to you, notice that, by using equations 1 and 2, we can substitute the variable "X1" for the expression rCOS(A1), and the variable "Y1" for the expression rSIN(A1). Thus, by substituting our "P1" coordinate values for these expressions, our equations now read:

X2 = X1COS(A2) -Y1SIN(A2) and
Y2 = Y1COS(A2) + X1SIN(A2).

We are ultimately solving for the "P2" coordinates (i.e., X2,Y2). Since we know the coordinate values for "P1" ("X1" and "Y1" are known), and since we know the number of degrees that we want to rotate from "P1" ("A2" is known), then we now have all the information needed to solve for the Cartesian coordinate values for "X2" and "Y2". Notice that these equations have taken us through the cycle described at the beginning of this section—from Cartesian coordinates to polar coordinates and then back to Cartesian coordinates.

APPENDIX C: BIBLIOGRAPHY

DeSausmarez, M., *Basic Design: The Dynamics of Visual Form*, New York, NY: Reinhold Publishing Corporation, 1964

Dondis, D.A., *A Primer of Visual Literacy*, Cambridge, MA: The M.I.T. Press, 1973

Itten, Johannes, *The Elements of Color*, New York, NY: Van Reinhold Co., 1970

McFee, J.K. and R.M. Degge, *Art, Culture, and Environment*, Dubuque, IO: Kendall/Hunt Publishing Co., 1977

Sparkes, R., *Teaching Art Basics*, New York, NY: Watson-Guptill Publications, 1973

Wolchonok, L., *The Art of Pictorial Composition*, New York, NY: Dover Publications, Inc., 1961

APPENDIX D: MACHINE LANGUAGE LISTING

```
LINE# LOC   CODE       LINE

00001  0000                                      ;     VARIABLES
00002  0000              *      = $033C
00003  033C              XLO    = *
00004  033C              XHI    = *+$1
00005  033C              YFIX   = *+$2
00006  033C              C      = *+$3
00007  033C              XI     = *+$4
00008  033C              DOWN   = *+$5
00009  033C              UP     = *+$6
00010  033C              STKPTR = *+$7
00011  033C              MU     = *+$8
00012  033C              C1     = *+$9
00013  033C              C2     = *+$A
00014  033C              ON     = *+$B
00015  033C              OFF    = *+$C
00016  033C              LLO    = *+$D
00017  033C              LHI    = *+$E
00018  033C              KEYEN  = *+$F
00019  033C              EN1    = *+$10
00020  033C              DIR1   = $035D
00021  033C              DIR2   = $035E
00022  033C              DIR3   = $035F
00023  033C              JOY1   = $0360
00024  033C              JOY2   = $0361
00025  033C              VARHI  = $03
00026  033C              XFLOT  = $4E
00027  033C              YFLOT  = $53
00028  033C              DXFLOT = $58
00029  033C              DYFLOT = $62
00030  033C              LFLOT  = $67
00031  033C              *      = $C000
00032  C000              TABLE
00033  C000   80         .BYTE  $80,$40,$20,$10
00033  C001   40
00033  C002   20
00033  C003   10
00034  C004   08         .BYTE  $08,$04,$02,$01
00034  C005   04
00034  C006   02
00034  C007   01
00035  C008   C0         .BYTE  $C0,$30,$0C,$03
00035  C009   30
00035  C00A   0C
00035  C00B   03
00036  C00C              .LIB   UTILITY
00037  C00C                                      ;  MACROS
00038  C00C                                      ;
00039  C00C              .MAC   FIXF1            ;TRANSFER FIXED
00040  C00C                     LDY ?1           ;     ?1 (LO)
00041  C00C                     LDA ?2           ;     ?2 (HI)
00042  C00C                     JSR $B391        ;TO FAC#1
00043  C00C              .MND
00044  C00C              .MAC   F2ADD            ;TRANSFER MEMORY
00045  C00C                     LDA #?1          ;AT ?1 (LO) ?2 (HI)
00046  C00C                     LDY #?2          ;TO FAC#2
00047  C00C                     JSR $B867        ;FAC#1=FAC#1 + FAC#2
00048  C00C              .MND
00049  C00C              .MAC   F2DIV            ;TRANSFER MEMORY
00050  C00C                     LDA #?1          ;AT ?1 (LO) ?2 (HI)
00051  C00C                     LDY #?2          ;TO FAC#2
00052  C00C                     JSR $BB0F        ;FAC#1=FAC#2/FAC#1
```

```
LOC   CODE        LINE

C00C              .MND
C00C              .MAC   COMPAR           ;COMPARE FAC#1 TO
C00C                     LDA #?1          ;MEMORY AT ?1 (LO)
C00C                     LDY #?2          ;          ?2 (HI)
C00C                     JSR $BC5B
C00C              .MND
C00C                                      ;
C00C                                      ;    ROM ROUTINES
C00C                                      ;
C00C                     F1F2   = $BC0F    ;FAC#1 TO FAC#2
C00C                     F1FIX  = $B7F7    ;FAC#1 TO FIXED
C00C                     BASF1  = $AD8A    ;BASIC TO FAC#1
C00C                     YF1    = $B3A2    ;Y REGISTER TO FAC#1
C00C                     SUBT   = $B853    ;PERFORM SUBTRACT
C00C                     EATCOM = $AEFD    ;EAT COMMA
C00C                     GETXY  = $B7EB    ;GET 2 BYTE AND 1 BYTE
C00C                     GETY   = $B7F1    ;GET 1 BYTE
C00C                     SGNF1  = $BC2B    ;GET SIGN OF FAC#1
C00C                     CHRGET = $0073    ;GET CHARACTER FROM BASIC
C00C              .END
C00C              .LIB   CLEAR2
C00C                                      ;
C00C              SCREEN                  ;SCREEN CLEAR
C00C                                      ;
C00C  20 F1 B7           JSR $B7F1        ;GET C
C00F  8A                 TXA              ;COLOR CHARACTER
C010  8D 21 D0           STA $D021        ;BACKGROUND
C013  0A                 ASL A            ;*2
C014  0A                 ASL A            ;*4
C015  0A                 ASL A            ;*8
C016  0A                 ASL A            ;*16
C017  18                 CLC
C018  65 65              ADC $65          ;C=C+C*16
C01A  A0 44              LDY #$44         ;SCREEN PAGE
C01C  A2 04              LDX #$04         ;NUMBER OF PAGES
C01E  20 27 C0           JSR FILL
C021  A9 00              LDA #$00         ;FILL CHARACTER
C023  A0 60              LDY #$60         ;SCREEN PAGE
C025  A2 20              LDX #$20         ;NUMBER OF PAGES
C027              FILL                    ;FILL X PAGES
C027  84 FC              STY $FC          ;AT Y LOCATION
C029  A0 00              LDY #$00         ;WITH A
C02B  84 FB              STY $FB
C02D              FILMOR
C02D  91 FB              STA ($FB),Y
C02F  C8                 INY
C030  D0 FB              BNE FILMOR
C032  E6 FC              INC $FC
C034  CA                 DEX
C035  D0 F6              BNE FILMOR
C037  60                 RTS
C038              .END
C038              .LIB   FIND2
C038              FIND                    ;FIND POINT ADDRESS
C038  A9 00              LDA #$00          ;HIGH BYTE OF ADDRESS
C03A  85 FC              STA $FC
C03C  AD 3E 03           LDA YFIX          ;GET Y COORDINATE
C03F  48                 PHA
C040  29 F8              AND #%11111000
C042  85 FB              STA $FB
C044  0A                 ASL A             ;*2
C045  26 FC              ROL $FC
C047  0A                 ASL A             ;*4
C048  26 FC              ROL $FC
C04A  65 FB              ADC $FB           ;*5
C04C  90 02              BCC SKP
C04E  E6 FC              INC $FC
C050              SKP
```

```
LINE# LOC   CODE        LINE
00121  C050  0A                 ASL A            ;*10
00122  C051  26 FC              ROL $FC
00123  C053  0A                 ASL A            ;*20
00124  C054  26 FC              ROL $FC
00125  C056  0A                 ASL A            ;*40
00126  C057  26 FC              ROL $FC
00127  C059  85 FB              STA $FB          ;(YAND248)*40
00128  C05B  AD 3C 03           LDA XLO
00129  C05E  29 F8              AND #%11111000
00130  C060  65 FB              ADC $FB          ;(YAND248)*40+(XAND248)
00131  C062  85 FB              STA $FB          ;PTR:PIXELS
00132  C064  85 FD              STA $FD          ;PTR:COLORS
00133  C066  AD 3D 03           LDA XHI
00134  C069  65 FC              ADC $FC
00135  C06B  48                 PHA
00136  C06C  4A                 LSR A            ;/2
00137  C06D  66 FD              ROR $FD
00138  C06F  4A                 LSR A            ;/4
00139  C070  66 FD              ROR $FD
00140  C072  4A                 LSR A            ;/8
00141  C073  66 FD              ROR $FD
00142  C075  18                 CLC
00143  C076  85 FE              STA $FE
00144  C078  68                 PLA
00145  C079  69 60              ADC #$60         ;PAGE $60
00146  C07B  85 FC              STA $FC
00147  C07D  68                 PLA
00148  C07E  29 07              AND #$07
00149  C080  65 FB              ADC $FB          ;ADD BLOCK ROW
00150  C082  90 02              BCC SKIP
00151  C084  E6 FC              INC $FC
00152  C086              SKIP                   ;GET BLOCK COLUMN
00153  C086  85 FB              STA $FB
00154  C088  AD 3C 03           LDA XLO
00155  C08B  29 07              AND #$07
00156  C08D  AE 44 03           LDX MU           ;MULTI COLOR INDICATOR
00157  C090  F0 03              BEQ OFFF         ;NOT MULTICOLOR
00158  C092  4A                 LSR A            ;GET BIT PAIR
00159  C093  09 08              ORA #$08
00160  C095              OFFF                   ;GET BIT(S) TO TURN OFF
00161  C095  AA                 TAX
00162  C096  BD 00 C0           LDA TABLE,X      ;BIT TABLE
00163  C099  8D 48 03           STA OFF
00164  C09C  60                 RTS
00165  C09D              SETUP                  ;SET UP COLOR MEMORY
00166  C09D  A0 00              LDY #$00
00167  C09F  8C 47 03           STY ON           ;SET BIT PAIR TO 0
00168  C0A2  AD 3F 03           LDA C            ;GET COLOR
00169  C0A5  09 F0              ORA #$F0         ;TURN ON HI NYBBLE
00170  C0A7  CD 21 D0           CMP $D021        ;COMPARE TO BCKGRND
00171  C0AA  F0 6A              BEQ DNE          ;COLOR IS BCKGRND
00172  C0AC  A9 44              LDA #$44         ;PAGE $44
00173  C0AE  18                 CLC              ;PTR TO COLORS 1,2
00174  C0AF  65 FE              ADC $FE
00175  C0B1  85 FE              STA $FE
00176  C0B3  AD 44 03           LDA MU
00177  C0B6  D0 0B              BNE MLTI
00178  C0B8  20 17 C1           JSR GETCOL       ;HIRES COLOR HANDLER
00179  C0BB  20 20 C1           JSR PUTCOL
00180  C0BE  AD 48 03           LDA OFF
00181  C0C1  D0 50              BNE ONN
00182  C0C3              MLTI                   ;MULTI COLOR HANDLER
00183  C0C3  20 17 C1           JSR GETCOL
00184  C0C6  8A                 TXA              ;GET COLOR 1
00185  C0C7  29 F0              AND #$F0
00186  C0C9  8D 45 03           STA C1           ;SAVE COLOR 1
00187  C0CC  4A                 LSR A            ;MOVE COLOR 1 TO LOWER NYBLE
00188  C0CD  4A                 LSR A
00189  C0CE  4A                 LSR A
```

```
LINE# LOC   CODE       LINE
00190 C0CF  4A                  LSR A
00191 C0D0  CD 3F 03            CMP C              ;IS C = COLOR 1
00192 C0D3  F0 0A               BEQ C1ISIT         ;C AND COLOR 1 ARE SAME
00193 C0D5  09 F0               ORA #$F0
00194 C0D7  CD 21 D0            CMP $D021          ;HAS COLOR 1 BEEN USED?
00195 C0DA  D0 0A               BNE TRYC2          ;TRY COLOR 2
00196 C0DC  20 20 C1            JSR PUTCOL         ;STORE C IN COLOR 1
00197 C0DF             C1ISIT                     ;GET "ON" BIT FOR COLOR 1
00198 C0DF  AD 48 03            LDA OFF
00199 C0E2  29 55               AND #%01010101
00200 C0E4  D0 2D               BNE ONN
00201 C0E6             TRYC2
00202 C0E6  AD 46 03            LDA C2             ;COLOR 2
00203 C0E9  CD 3F 03            CMP C              ;IS COLOR SAME AS COLOR 2?
00204 C0EC  F0 0F               BEQ C2ISIT         ;COLORS MATCH
00205 C0EE  09 F0               ORA #$F0           ;HAS COLOR 2 BEEN USED?
00206 C0F0  CD 21 D0            CMP $D021
00207 C0F3  D0 0F               BNE TRYC3          ;YES, IT IS USED: FORCE COLOR 3
00208 C0F5  AD 3F 03            LDA C              ;STORE C IN COLOR 2
00209 C0F8  0D 45 03            ORA C1
00210 C0FB  91 FD               STA ($FD),Y
00211 C0FD             C2ISIT                     ;GET "ON" BIT FOR COLOR 2
00212 C0FD  AD 48 03            LDA OFF
00213 C100  29 AA               AND #%10101010
00214 C102  D0 0F               BNE ONN            ;FORCED BRANCH
00215 C104             TRYC3                      ;FORCE IT IN COLOR3
00216 C104  A9 94               LDA #$94           ;COLOR RAM
00217 C106  18                  CLC
00218 C107  65 FE               ADC $FE
00219 C109  85 FE               STA $FE
00220 C10B  AD 3F 03            LDA C
00221 C10E  91 FD               STA ($FD),Y
00222 C110  AD 48 03            LDA OFF
00223 C113             ONN                        ;PIXEL(S) TO BE TURNED ON
00224 C113  8D 47 03            STA ON
00225 C116             DNE
00226 C116  60                  RTS
00227 C117             GETCOL                     ;RETRIEVE COLOR
00228 C117  B1 FD               LDA ($FD),Y
00229 C119  AA                  TAX
00230 C11A  29 0F               AND #$0F
00231 C11C  8D 46 03            STA C2
00232 C11F  60                  RTS
00233 C120             PUTCOL                     ;STORE COLOR IN MEMORY
00234 C120  AD 3F 03            LDA C
00235 C123  0A                  ASL A              ;*2
00236 C124  0A                  ASL A              ;*4
00237 C125  0A                  ASL A              ;*8
00238 C126  0A                  ASL A              ;*16
00239 C127  0D 46 03            ORA C2
00240 C12A  91 FD               STA ($FD),Y
00241 C12C  60                  RTS
00241 C12D
00242 C12D             F1XFL                      ;FAC#1 TO X FLOAT
00243 C12D  A2 4E               LDX #XFLOT
00244 C12F  2C         .BYTE    $2C
00245 C130             F1YFL                      ;FAC#1 TO Y FLOAT
00246 C130  A2 53               LDX #YFLOT
00247 C132  2C         .BYTE    $2C
00248 C133             F1DXFL                     ;FAC#1 TO DX FLOAT
00249 C133  A2 58               LDX #DXFLOT
00250 C135  2C         .BYTE    $2C
00251 C136             F1DYFL                     ;FAC#1 TO DY FLOAT
00252 C136  A2 62               LDX #DYFLOT
00253 C138  2C         .BYTE    $2C
00254 C139             F1LFL                      ;FAC#1 TO L FLOAT
00255 C139  A2 67               LDX #LFLOT
00256 C13B  A0 03               LDY #VARHI
00257 C13D  4C D4 BB            JMP $BBD4
```

```
LINE# LOC   CODE       LINE
00258 C140             XFLF1                   ;X FLOAT TO FAC#1
00259 C140  A9 4E             LDA #XFLOT
00260 C142  2C         .BYTE  $2C
00261 C143             YFLF1                   ;Y FLOAT TO FAC#1
00262 C143  A9 53             LDA #YFLOT
00263 C145  2C         .BYTE  $2C
00264 C146             LFLF1                   ;L FLOAT TO FAC#1
00265 C146  A9 67             LDA #LFLOT
00266 C148  A0 03             LDY #VARHI
00267 C14A  4C A2 BB          JMP $BBA2
00268 C14D             F1L                     ;FAC#1 TO L FIXED
00269 C14D  20 39 C1          JSR F1LFL
00270 C150  20 F7 B7          JSR F1FIX
00271 C153  A6 14             LDX $14
00272 C155  A4 15             LDY $15
00273 C157  8E 49 03          STX LLO
00274 C15A  8C 4A 03          STY LHI
00275 C15D  60                RTS
00276 C15E             .LIB   POINT2
00277 C15E             PLOT
00278 C15E  20 FD AE          JSR EATCOM        ;EAT COMMA
00279 C161  20 EB B7          JSR GETXY         ;GET X,Y
00280 C164  8E 3E 03          STX YFIX
00281 C167  A6 14             LDX $14           ;
00282 C169  A4 15             LDY $15           ;GET X FIXED
00283 C16B  8E 3C 03          STX XLO
00284 C16E  8C 3D 03          STY XHI
00285 C171  C9 A4             CMP #$A4          ;'TO' TOKEN
00286 C173  F0 1F             BEQ LINE          ;LINE ROUTINE
00287 C175             POINT
00288 C175  20 F1 B7          JSR GETY          ;GET C
00289 C178  8E 3F 03          STX C
00290 C17B  20 F1 B7          JSR GETY          ;GET C
00291 C17E  8E 44 03          STX MU
00292 C181             PLOTIT
00293 C181  20 38 C0          JSR FIND          ;FIND PIXEL
00294 C184  20 9D C0          JSR SETUP         ;FIND PIXEL
00295 C187  AD 48 03          LDA OFF
00296 C18A  49 FF             EOR #$FF
00297 C18C  31 FB             AND ($FB),Y       ;TURN OFF PIXEL
00298 C18E  0D 47 03          ORA ON
00299 C191  91 FB             STA ($FB),Y       ;STORE COLOR
00300 C193  60                RTS
00301 C194             .END
00302 C194             .LIB   LINE2
00303 C194             ;
00304 C194             LINE
00305 C194             ;
00306 C194  20 73 00          JSR CHRGET          ;EAT 'TO'
00307 C197  20 8A AD          JSR BASF1           ;X2->FAC1
00308 C19A  20 0F BC          JSR F1F2            ;FAC1->FAC2
00309 C19D                    FIXF1 XLO,XHI       ;FIXED->FAC1
00314 C1A6  20 2D C1          JSR F1XFL           ;FAC1->XFLOAT
00315 C1A9  20 53 B8          JSR SUBT            ;SUBTRACT
00316 C1AC  20 33 C1          JSR F1DXFL          ;FAC1->DXFLOAT
00317 C1AF  46 66             LSR $66             ;ABS(FAC1)
00318 C1B1  20 4D C1          JSR F1L             ;FAC1->LFLOT
00319 C1B4                                     ;FAC1->FIXED
00320 C1B4  20 F1 B7          JSR GETY            ;EAT , & GET Y
00321 C1B7  8A                TXA
00322 C1B8  A8                TAY
00323 C1B9  20 A2 B3          JSR YF1             ;Y2->FAC1
00324 C1BC  20 0F BC          JSR F1F2            ;FAC1->FAC2
00325 C1BF  AC 3E 03          LDY YFIX
00326 C1C2  20 A2 B3          JSR YF1             ;Y->FAC1
00327 C1C5  20 30 C1          JSR F1YFL           ;FAC1->YFLOAT
00328 C1C8  20 53 B8          JSR SUBT            ;SUBTRACT
00329 C1CB  20 36 C1          JSR F1DYFL          ;FAC1->DYFLOAT
00330 C1CE  46 66             LSR $66             ;ABS(DY)
```

```
LINE# LOC   CODE        LINE
00331  C1D0                    COMPAR LFLOT,VARHI ;CMP FAC1->MEM
00336  C1D7  30 0B             BMI DNTMES
00337  C1D9  20 2B BC          JSR SGNF1          ;SGN(FAC1)
00338  C1DC  D0 03             BNE LINEOK
00339  C1DE  4C 75 C1          JMP POINT
00340  C1E1             LINEOK
00341  C1E1  20 4D C1          JSR F1L            ;FAC1->LFLOAT
00342  C1E4             DNTMES
00343  C1E4  20 46 C1          JSR LFLF1           ;LFLOT->FAC1
00344  C1E7                    F2DIV DXFLOT,VARHI  ;DIVIDE
00349  C1EE  20 33 C1          JSR F1DXFL         ;FAC1->DXFLOAT
00350  C1F1  20 46 C1          JSR LFLF1           ;LFLOT->FAC1
00351  C1F4                    F2DIV DYFLOT,VARHI  ;DIVIDE
00356  C1FB  20 36 C1          JSR F1DYFL         ;FAC1->DYFLOAT
00357  C1FE  20 F1 B7          JSR GETY            ;GET C
00358  C201  8E 3F 03          STX C
00359  C204  20 F1 B7          JSR GETY            ;GET MU
00360  C207  8E 44 03          STX MU
00361  C20A             NEXT
00362  C20A  20 81 C1          JSR PLOTIT
00363  C20D  20 40 C1          JSR XFLF1           ;XFLOT->FAC1
00364  C210                    F2ADD DXFLOT,VARHI
00369  C217  20 2B BC          JSR SGNF1
00370  C21A  30 37             BMI QUIT
00371  C21C  20 2D C1          JSR F1XFL          ;FAC1->XFLOAT
00372  C21F  20 F7 B7          JSR F1FIX
00373  C222  A5 14             LDA $14
00374  C224  A6 15             LDX $15
00375  C226  8D 3C 03          STA XLO
00376  C229  8E 3D 03          STX XHI
00377  C22C  20 43 C1          JSR YFLF1           ;YFLOT->FAC1
00378  C22F                    F2ADD DYFLOT,VARHI
00383  C236  20 2B BC          JSR SGNF1
00384  C239  30 18             BMI QUIT
00385  C23B  20 30 C1          JSR F1YFL          ;FAC1->YFLOAT
00386  C23E  20 F7 B7          JSR F1FIX
00387  C241  A5 14             LDA $14
00388  C243  8D 3E 03          STA YFIX
00389  C246  CE 49 03          DEC LLO
00390  C249  D0 BF             BNE NEXT
00391  C24B  CE 4A 03          DEC LHI
00392  C24E  10 BA             BPL NEXT
00393  C250  4C 81 C1          JMP PLOTIT
00394  C253             QUIT
00395  C253  60                RTS
00396  C254             .END
00397  C254             .LIB   PAINT2
00398  C254             PAINT               ;PAINT A SHAPE
00399  C254  20 FD AE          JSR EATCOM   ;EAT COMMA
00400  C257  20 EB B7          JSR GETXY    ;GET X AND Y
00401  C25A  8E 3E 03          STX YFIX     ;STORE Y
00402  C25D  A6 14             LDX $14
00403  C25F  A4 15             LDY $15
00404  C261  8E 3C 03          STX XLO      ;STORE X
00405  C264  8C 3D 03          STY XHI
00406  C267  20 F1 B7          JSR GETY     ;GET COLOR
00407  C26A  8E 3F 03          STX C        ;STORE COLOR
00408  C26D  8A                TXA
00409  C26E  09 F0             ORA #$F0
00410  C270  CD 21 D0          CMP $D021    ;TRYING TO FILL WITH BACKGROUND COLOR?
00411  C273  D0 04             BNE GOON     ;QUIT IF TRUE
00412  C275  20 F1 B7          JSR GETY     ;GRAB MU VARIABLE
00413  C278  60                RTS
00414  C279             GOON
00415  C279  20 F1 B7          JSR GETY     ;GET MULTI INDICATOR
00416  C27C  8A                TXA
00417  C27D  F0 08             BEQ START    ;HIRES
00418  C27F  AD 3C 03          LDA XLO      ;MULTI
00419  C282  29 FE             AND #$FE     ;KNOCK OFF LSB
```

```
LINE# LOC   CODE        LINE
00420  C284  8D 3C 03           STA XLO
00421  C287           START
00422  C287  8E 44 03           STX MU
00423  C28A  E8                 INX
00424  C28B  8E 40 03           STX XI          ;X STEP SIZE
00425  C28E  A0 00              LDY #$00        ;SET UP STKPTR
00426  C290  8C 43 03           STY STKPTR
00427  C293           LOOP                      ;INITIALIZE INTERESTING POINT WATCHERS
00428  C293  A9 00              LDA #$00
00429  C295  8D 42 03           STA UP
00430  C298  8D 41 03           STA DOWN
00431  C29B           FINDIT                    ;FIND LEFT EDGE OR LEFT BOUNDRY
00432  C29B  AD 3C 03           LDA XLO
00433  C29E  D0 05              BNE FLOOP
00434  C2A0  CD 3D 03           CMP XHI
00435  C2A3  F0 20              BEQ FILLIT      ;FOUND LEFT BOUNDRY
00436  C2A5           FLOOP                     ;STEP LEFT
00437  C2A5  38                 SEC
00438  C2A6  ED 40 03           SBC XI
00439  C2A9  8D 3C 03           STA XLO
00440  C2AC  B0 03              BCS L5
00441  C2AE  CE 3D 03           DEC XHI
00442  C2B1           L5
00443  C2B1  20 43 C3           JSR PEEK        ;LOOK FOR EDGE OF SHAPE
00444  C2B4  F0 E5              BEQ FINDIT      ;LOOK AGAIN
00445  C2B6  18                 CLC             ;FOUND EDGE OF SHAPE
00446  C2B7  AD 3C 03           LDA XLO         ;STEP RIGHT
00447  C2BA  6D 40 03           ADC XI
00448  C2BD  8D 3C 03           STA XLO
00449  C2C0  90 03              BCC FILLIT
00450  C2C2  EE 3D 03           INC XHI
00451  C2C5           FILLIT
00452  C2C5  EE 3E 03           INC YFIX        ;LOOK DOWN
00453  C2C8  20 43 C3           JSR PEEK
00454  C2CB  D0 0B              BNE L6          ;NOT INTERESTED
00455  C2CD  AD 41 03           LDA DOWN        ;CHECK LAST POINT
00456  C2D0  D0 0B              BNE L7          ;NOT INTERESTED
00457  C2D2  20 4E C3           JSR PUSH        ;SAVE INTERESTING POINT
00458  C2D5  A9 01              LDA #$01        ;LAST POINT WAS OFF
00459  C2D7  2C             .BYTE $2C
00460  C2D8           L6
00461  C2D8  A9 00              LDA #$00        ;LAST POINT WAS ON
00462  C2DA  8D 41 03           STA DOWN
00463  C2DD           L7
00464  C2DD  CE 3E 03           DEC YFIX        ;STEP DOWN
00465  C2E0  CE 3E 03           DEC YFIX
00466  C2E3  20 43 C3           JSR PEEK        ;LOOK UP
00467  C2E6  D0 0B              BNE L8          ;NOT INTERESTED
00468  C2E8  AD 42 03           LDA UP          ;CHECK LAST POINT
00469  C2EB  D0 0B              BNE L9          ;NOT INTERESTED
00470  C2ED  20 4E C3           JSR PUSH        ;SAVE INTERESTING POINT
00471  C2F0  A9 01              LDA #$01        ;LAST POINT WAS OFF
00472  C2F2  2C             .BYTE $2C
00473  C2F3           L8
00474  C2F3  A9 00              LDA #$00        ;LAST POINT WAS ON
00475  C2F5  8D 42 03           STA UP
00476  C2F8           L9
00477  C2F8  EE 3E 03           INC YFIX        ;STEP DOWN
00478  C2FB  20 81 C1           JSR PLOTIT      ;PLOT POINT
00479  C2FE  AD 3C 03           LDA XLO         ;STEP RIGHT
00480  C301  18                 CLC
00481  C302  6D 40 03           ADC XI
00482  C305  8D 3C 03           STA XLO
00483  C308  AD 3D 03           LDA XHI
00484  C30B  69 00              ADC #$00
00485  C30D  8D 3D 03           STA XHI
00486  C310  F0 29              BEQ L11         ;CHECK WRAP
00487  C312  AD 3C 03           LDA XLO
00488  C315  C9 40              CMP #$40
```

```
LOC   CODE      LINE
C317  90 22               BCC L11           ;DIDNT REACH RIGHT BOUNDRY YET
C319            L10
C319  AE 43 03            LDX STKPTR        ;OUT OF INTERESTING POINTS?
C31C  F0 2F               BEQ END           ;YES
C31E  CA                  DEX               ;POP INTERESTING POINT
C31F  BD 00 CB            LDA $CB00,X
C322  8D 3D 03            STA XHI
C325  BD 00 CA            LDA $CA00,X
C328  8D 3C 03            STA XLO
C32B  BD 00 C9            LDA $C900,X
C32E  8D 3E 03            STA YFIX
C331  CE 43 03            DEC STKPTR
C334  C9 C8               CMP #$C8          ;CHECK Y BOUNDRY
C336  B0 E1               BCS L10           ;POP ANOTHER
C338  4C 93 C2            JMP LOOP          ;HANDLE NEW INTERESTING POINT
C33B            L11
C33B  20 43 C3            JSR PEEK          ;RIGHT EDGE?
C33E  D0 D9               BNE L10           ;YES:  POP
C340  4C C5 C2            JMP FILLIT
C343            PEEK                        ;LOOK AT POINT
C343  20 38 C0            JSR FIND
C346  A0 00               LDY #$00
C348  B1 FB               LDA ($FB),Y
C34A  2D 48 03            AND OFF
C34D            END
C34D  60                  RTS
C34E            PUSH                        ;PUSH INTERESTING POINT
C34E  AE 43 03            LDX STKPTR
C351  AD 3E 03            LDA YFIX
C354  9D 00 C9            STA $C900,X
C357  AD 3C 03            LDA XLO
C35A  9D 00 CA            STA $CA00,X
C35D  AD 3D 03            LDA XHI
C360  9D 00 CB            STA $CB00,X
C363  EE 43 03            INC STKPTR
C366  60                  RTS
C367            .END
C367            .LIB      INSERT
C367            INSERT                      ;INSTALL INTERRUPT WEDGE
C367  20 F1 B7            JSR $B7F1          ;GET 1 BYTE
C36A  8E 4B 03            STX KEYEN          ;KEYBOARD SWITCH
C36D  20 F1 B7            JSR $B7F1          ;GET 1 BYTE
C370  8A                  TXA
C371  48                  PHA                ;MASTER SWITCH
C372  20 F1 B7            JSR $B7F1          ;GET 1 BYTE
C375  8E 4C 03            STX EN1            ;PLAYER 2 ENABLE
C378  20 F1 B7            JSR $B7F1          ;GET 1 BYTE
C37B  8E 5D 03            STX DIR1           ;MISSLE 1 DIRECTION
C37E  20 F1 B7            JSR $B7F1          ;GET 1 BYTE
C381  8E 5E 03            STX DIR2           ;MISSLE 2 DIRECTION
C384  20 F1 B7            JSR $B7F1          ;GET 1 BYTE
C387  8E 5F 03            STX DIR3           ;TARGET DIRECTION
C38A  20 F1 B7            JSR $B7F1          ;GET 1 BYTE
C38D  68                  PLA
C38E  F0 1D               BEQ WDGOFF         ;MASTER OFF
C390  A9 7F               LDA #%01111111     ;INTERRUPT MASK
C392  8D 0D DC            STA $DC0D          ;DISABLE INTERRUPT
C395  8E 07 DC            STX $DC07          ;TIMER B (VELOCITY)
C398  A9 C2               LDA #<WEDGE        ;NEW WEDGE VECTOR
C39A  8D 14 03            STA $0314          ;IRQ VECTOR
C39D  A9 C3               LDA #>WEDGE        ;NEW WEDGE VECTOR
C39F  8D 15 03            STA $0315          ;IRQ VECTOR
C3A2  A2 11               LDX #$11           ;START TIMER B
C3A4  A0 82               LDY #$82           ;ENABLE TIMER B INTERRUPT
C3A6            STORE
C3A6  8E 0F DC            STX $DC0F          ;TIMER B CONTROL
C3A9  8C 0D DC            STY $DC0D          ;ENABLE TIMER B INTERRUPT
C3AC  60                  RTS
C3AD            WDGOFF
```

```
LINE# LOC   CODE        LINE
00558  C3AD  A9 7F              LDA #$7F        ;INTERRUPT MASK
00559  C3AF  8D 0D DC           STA $DC0D       ;DISABLE INTERRUPTS
00560  C3B2  A9 31              LDA #$31        ;ORIGINAL IRQ
00561  C3B4  8D 14 03           STA $0314       ;IRQ VECTOR
00562  C3B7  A9 EA              LDA #$EA        ;ORIGINAL IRQ
00563  C3B9  8D 15 03           STA $0315       ;IRQ VECTOR
00564  C3BC  A2 08              LDX #$08        ;START TIMER A
00565  C3BE  A0 81              LDY #$81        ;ENABLE TIMER A INTERRUPT
00566  C3C0  D0 E4              BNE STORE
00567  C3C2              .END
00568  C3C2              .LIB   SPRITE
00569  C3C2              WEDGE                  ;SPRITE
00570  C3C2  20 A8 C4           JSR GETJOY      ;GET PLAYER MOVEMENTS
00571  C3C5  AE 5D 03           LDX DIR1        ;MISSLE 1 DIRECTION
00572  C3C8  F0 53              BEQ MP1         ;MISSLE 1 DISABLED
00573  C3CA  AD 15 D0           LDA $D015       ;SPRITE ENABLE BYTE
00574  C3CD  29 04              AND #$04        ;SPRITE 4 BIT
00575  C3CF  D0 2D              BNE MM1         ;ALREADY IN MOTION
00576  C3D1  AD 60 03           LDA JOY1        ;CHECK FIREBUTTON
00577  C3D4  29 10              AND #$10
00578  C3D6  D0 45              BNE MP1         ;NOT FIRED
00579  C3D8  AD 15 D0           LDA $D015       ;TURN ON MISSLE 1
00580  C3DB  09 04              ORA #$04
00581  C3DD  8D 15 D0           STA $D015
00582  C3E0  AD 10 D0           LDA $D010       ;POSITION MISSLE 1 AT PLAYER 1
00583  C3E3  29 FB              AND #$FB
00584  C3E5  8D 10 D0           STA $D010
00585  C3E8  29 01              AND #$01
00586  C3EA  0A                 ASL A
00587  C3EB  0A                 ASL A
00588  C3EC  0D 10 D0           ORA $D010
00589  C3EF  8D 10 D0           STA $D010
00590  C3F2  AD 00 D0           LDA $D000
00591  C3F5  8D 04 D0           STA $D004
00592  C3F8  AD 01 D0           LDA $D001
00593  C3FB  8D 05 D0           STA $D005
00594  C3FE              MM1                    ;MOVE MISSLE 1
00595  C3FE  8A                 TXA             ;DIRECTION
00596  C3FF  A2 04              LDX #$04
00597  C401  A0 05              LDY #$05
00598  C403  20 1A C5           JSR MOVE        ;MOVE MISSLE 1
00599  C406  AD 10 D0           LDA $D010       ;X WRAP ?
00600  C409  29 04              AND #$04
00601  C40B  0D 04 D0           ORA $D004
00602  C40E  F0 05              BEQ M1OFF       ;WRAP OCCURED
00603  C410  AD 05 D0           LDA $D005       ;Y WRAP ?
00604  C413  D0 08              BNE MP1         ;NO WRAP OCCURED
00605  C415              M1OFF                  ;TURN OFF MISSLE 1
00606  C415  AD 15 D0           LDA $D015       ;SPRITE ENABLE BYTE
00607  C418  29 FB              AND #$FB
00608  C41A  8D 15 D0           STA $D015
00609  C41D              MP1                    ;MOVE PLAYER 1
00610  C41D  AD 60 03           LDA JOY1        ;GET MOVEMENT
00611  C420  A2 00              LDX #$00
00612  C422  A0 07              LDY #$07
00613  C424  20 1A C5           JSR MOVE        ;MOVE
00614  C427  AD 4C 03           LDA EN1         ;PLAYER 2 ENABLED ?
00615  C42A  F0 62              BEQ TARGET      ;NOT ENABLED
00616  C42C  AE 5E 03           LDX DIR2        ;MISSLE 2 DIRECTION
00617  C42F  F0 53              BEQ MP2         ;MISSLE 2 DISABLED
00618  C431  AD 15 D0           LDA $D015       ;SPRITE ENABLE BYTE
00619  C434  29 08              AND #$08
00620  C436  D0 2D              BNE MM2         ;ALREADY IN MOTION
00621  C438  AD 61 03           LDA JOY2        ;CHECK FIREBUTTON
00622  C43B  29 10              AND #$10
00623  C43D  D0 45              BNE MP2         ;NOT FIRED
00624  C43F  AD 15 D0           LDA $D015       ;TURN ON MISSLE 2
00625  C442  09 08              ORA #$08
00626  C444  8D 15 D0           STA $D015
```

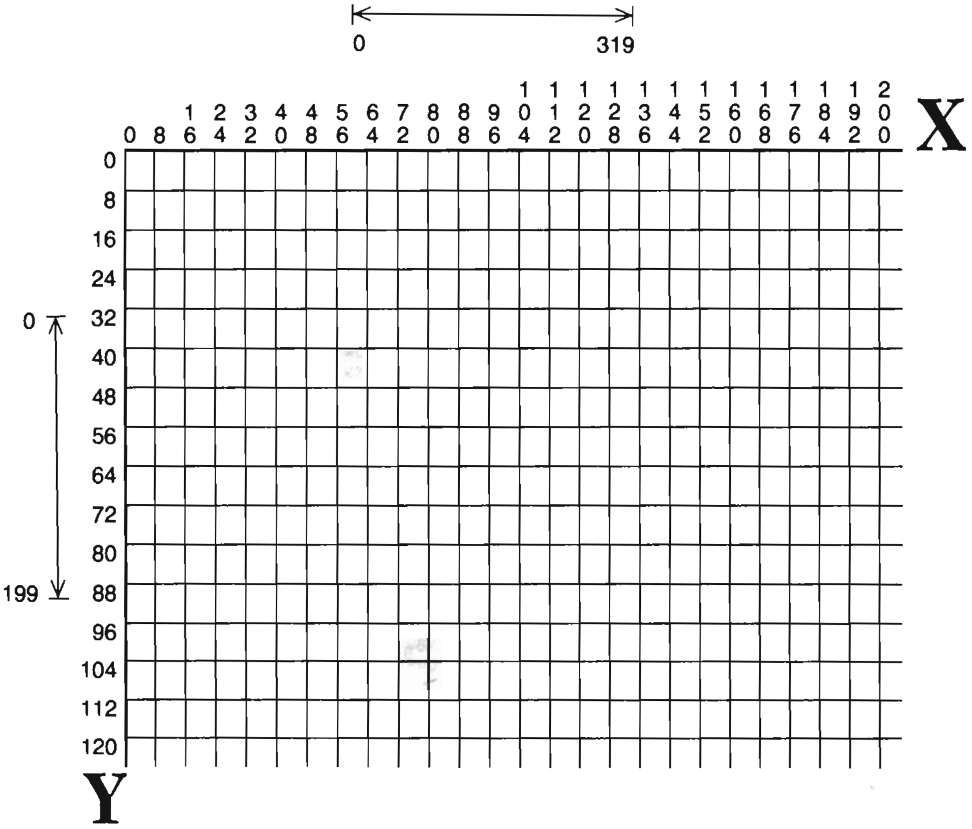
```
LINE# LOC   CODE       LINE
00627 C447  AD 10 D0          LDA $D010       ;PLACE MISSLE 2 AT PLAYER 2
00628 C44A  29 F7             AND #$F7
00629 C44C  8D 10 D0          STA $D010
00630 C44F  29 02             AND #$02
00631 C451  0A                ASL A
00632 C452  0A                ASL A
00633 C453  0D 10 D0          ORA $D010
00634 C456  8D 10 D0          STA $D010
00635 C459  AD 02 D0          LDA $D002
00636 C45C  8D 06 D0          STA $D006
00637 C45F  AD 03 D0          LDA $D003
00638 C462  8D 07 D0          STA $D007
00639 C465             MM2                    ;MOVE MISSLE 2
00640 C465  8A                TXA              ;DIRECTION
00641 C466  A2 06             LDX #$06
00642 C468  A0 04             LDY #$04
00643 C46A  20 1A C5          JSR MOVE         ;MOVE MISSLE 2
00644 C46D  AD 10 D0          LDA $D010        ;X WRAP ?
00645 C470  29 08             AND #$08
00646 C472  0D 06 D0          ORA $D006
00647 C475  F0 05             BEQ M2OFF        ;X WRAP
00648 C477  AD 07 D0          LDA $D007        ;Y WRAP ?
00649 C47A  D0 08             BNE MP2          ;NO Y WRAP
00650 C47C             M2OFF                  ;TURN OFF MISSLE 2
00651 C47C  AD 15 D0          LDA $D015        ;SPRITE ENABLE BYTE
00652 C47F  29 F7             AND #$F7
00653 C481  8D 15 D0          STA $D015
00654 C484             MP2                    ;MOVE PLAYER 2
00655 C484  AD 61 03          LDA JOY2          ;GET DIRECTION
00656 C487  A2 02             LDX #$02
00657 C489  A0 06             LDY #$06
00658 C48B  20 1A C5          JSR MOVE          ;MOVE PLAYER 2
00659 C48E             TARGET                 ;TARGET SPRITE
00660 C48E  AE 5F 03          LDX DIR3          ;TARGET DIRECTION
00661 C491  F0 08             BEQ CHECK         ;DISABLED
00662 C493  8A                TXA               ;DIRECTION
00663 C494  A2 08             LDX #$08
00664 C496  A0 03             LDY #$03
00665 C498  20 1A C5          JSR MOVE          ;MOVE TARGET
00666 C49B             CHECK
00667 C49B  AD 0D DC          LDA $DC0D         ;OLD IRQ FLAG ?
00668 C49E  29 01             AND #$01
00669 C4A0  F0 03             BEQ QUT           ;NOT TIME FOR OLD IRQ
00670 C4A2  4C 31 EA          JMP $EA31         ;OLD IRQ
00671 C4A5             QUT
00672 C4A5  4C BC FE          JMP $FEBC         ;END INTERRUPT
00672 C4A8
00673 C4A8             .LIB   GETJOY
00674 C4A8             GETJOY
00675 C4A8  AD 4B 03          LDA KEYEN       ;KEYBOARD OR JOYSTICK
00676 C4AB  D0 0D             BNE GETKEY      ;KEYBOARD
00677 C4AD  AD 01 DC          LDA $DC01       ;GET JOY1
00678 C4B0  8D 60 03          STA JOY1
00679 C4B3  AD 00 DC          LDA $DC00       ;GET JOY2
00680 C4B6  8D 61 03          STA JOY2
00681 C4B9  60                RTS
00682 C4BA             GETKEY                 ;KEYBOARD SCAN
00683 C4BA  A9 00             LDA #$00
00684 C4BC  AA                TAX             ;LEFT SHIFT BIT# (7-BIT)
00685 C4BD  A8                TAY             ;JOY1 INDEX
00686 C4BE  8D 00 DC          STA $DC00
00687 C4C1  AD 01 DC          LDA $DC01
00688 C4C4  A9 FD             LDA #$FD        ;ROW SELECT  (253)
00689 C4C6  20 F6 C4          JSR CHKFIR      ;CHECK BIT 7 OF ROW $FD
00690 C4C9  A9 FB             LDA #$FB        ;ROW SELECT  (251)
00691 C4CB  20 D9 C4          JSR CHKPLY      ;CHECK LEFT PLAYER KEYS
00692 C4CE  A0 01             LDY #$01        ;JOY2 INDEX
00693 C4D0  A2 03             LDX #$03        ;RIGHT SHIFT BIT# (7-BIT)
00694 C4D2  A9 BF             LDA #$BF        ;ROW SELECT  (191)
```

```
LINE# LOC   CODE        LINE
00695 C4D4  20 F6 C4           JSR CHKFIR      ;CHECK BIT 4 OF ROW $BF
00696 C4D7  A9 DF              LDA #$DF        ;ROW SELECT  (223)
00697 C4D9              CHKPLY                 ;CHECK FOR PLAYER MOVEMENT
00698 C4D9  8D 00 DC           STA $DC00       ;LATCH ROW SELECT
00699 C4DC  AD 01 DC           LDA $DC01       ;GET COLUMN
00700 C4DF  09 C9              ORA #%11001001  ;DISREGARD COLUMNS
00701 C4E1  A2 07              LDX #$07        ;CHECK 8 DIRECTIONS
00702 C4E3              CMP1
00703 C4E3  DD 0A C5           CMP TBL1,X      ;CHECK FOR VALID KEY COMBINATIONS
00704 C4E6  F0 04              BEQ POK         ;FOUND VALID KEYS
00705 C4E8  CA                 DEX
00706 C4E9  10 F8              BPL CMP1        ;TRY ANOTHER COMBINATION
00707 C4EB  60                 RTS
00708 C4EC              POK                    ;STORE VALID MOVEMENT
00709 C4EC  BD 12 C5           LDA TBL2,X      ;GET KEY TO JOY CONVERSION
00710 C4EF  39 60 03           AND JOY1,Y      ;DONT DISTURB FIRE INDICATOR
00711 C4F2  99 60 03           STA JOY1,Y
00712 C4F5  60                 RTS
00713 C4F6              CHKFIR
00714 C4F6  8D 00 DC           STA $DC00       ;LATCH KEYBOARD ROW
00715 C4F9  AD 01 DC           LDA $DC01       ;GET COLUMN
00716 C4FC  3D 00 C0           AND TABLE,X     ;CHECK FOR SHIFT KEY
00717 C4FF  D0 03              BNE SKIPP       ;NOT PRESSED
00718 C501  A9 0F              LDA #$0F        ;SET FIRE TO TRUE AND INIT MOVEMENT TO FALSE
00719 C503  2C              .BYTE $2C
00720 C504              SKIPP
00721 C504  A9 1F              LDA #$1F        ;SET FIRE TO FALSE AND INIT MOVEMENT TO FALSE
00722 C506  99 60 03           STA JOY1,Y
00723 C509  60                 RTS
00724 C50A              TBL1                   ;VALID COLUMN COMBINATIONS
00725 C50A  FD          .BYTE 253,239,251,223
00725 C50B  EF
00725 C50C  FB
00725 C50D  DF
00726 C50E  F9          .BYTE 249,235,221,207
00726 C50F  EB
00726 C510  DD
00726 C511  CF
00727 C512              TBL2                   ;KEY TO JOY CONVERSIONS
00728 C512  1E          .BYTE $1E,$1D,$1B,$17
00728 C513  1D
00728 C514  1B
00728 C515  17
00729 C516  1E          .BYTE $1E,$1D,$1E,$1D
00729 C517  1D
00729 C518  1E
00729 C519  1D
00730 C51A              .END
00731 C51A              .LIB  MOVE
00732 C51A              MOVE
00733 C51A  4A                 LSR A           ;CHECK FOR UP
00734 C51B  B0 03              BCS DWN         ;NOT UP
00735 C51D  DE 01 D0           DEC $D001,X     ;SPRITE Y POSITION
00736 C520              DWN
00737 C520  4A                 LSR A           ;CHECK FOR DOWN
00738 C521  B0 03              BCS LEFT        ;NOT DOWN
00739 C523  FE 01 D0           INC $D001,X     ;SPRITE Y POSITION
00740 C526              LEFT
00741 C526  4A                 LSR A           ;CHECK FOR LEFT
00742 C527  B0 12              BCS RIGHT       ;NOT LEFT
00743 C529  BD 00 D0           LDA $D000,X     ;SPRITE X POSITION LO
00744 C52C  D0 09              BNE DECX        ;NO X LO UNDERFLOW
00745 C52E  AD 10 D0           LDA $D010       ;SWITCH SCREEN SECTIONS
00746 C531  59 00 C0           EOR TABLE,Y     ;GET BIT
00747 C534  8D 10 D0           STA $D010
00748 C537              DECX
00749 C537  DE 00 D0           DEC $D000,X     ;ADJUST X LO
00750 C53A  60                 RTS
00751 C53B              RIGHT
```

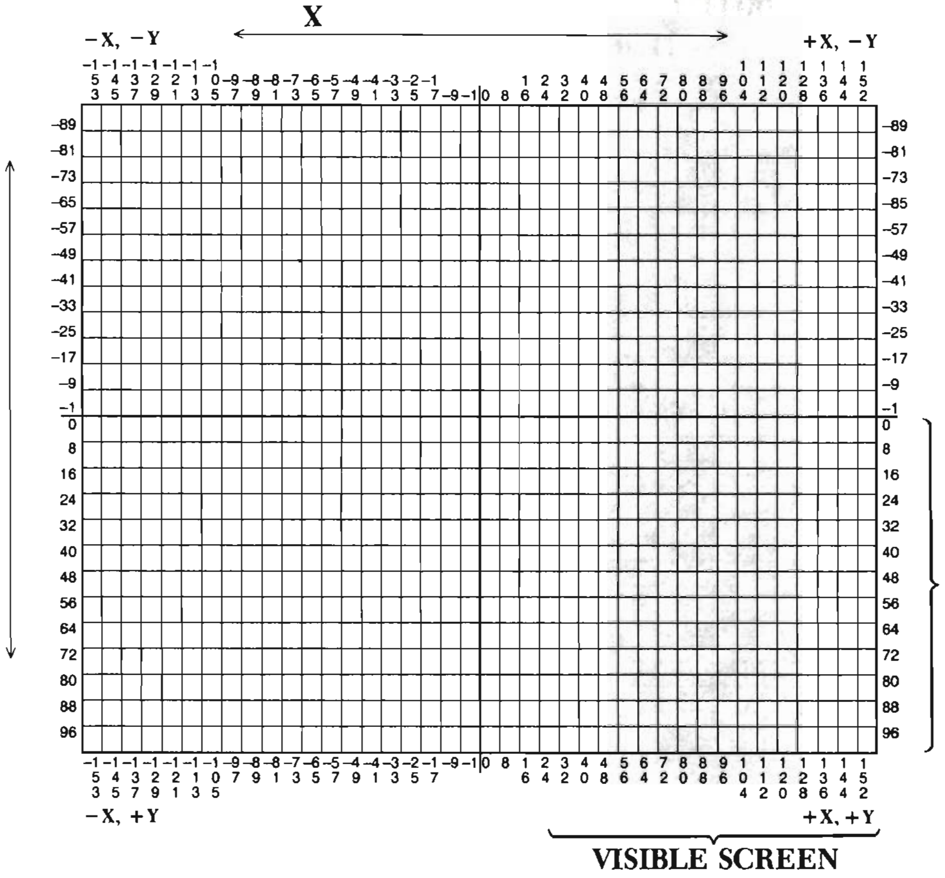
```
00752  C53B  4A                    LSR A            ;CHECK FOR RIGHT
00753  C53C  B0 0E                 BCS RETURN       ;NOT RIGHT
00754  C53E  FE 00 D0              INC $D000,X      ;ADJUST X LO
00755  C541  D0 09                 BNE RETURN       ;NOT OVERFLOW
00756  C543  AD 10 D0              LDA $D010        ;ADJUST X HI
00757  C546  59 00 C0              EOR TABLE,Y      ;GET BIT
00758  C549  8D 10 D0              STA $D010
00759  C54C                 RETURN
00760  C54C  60                    RTS
00761  C54D                 .END
00762  C54D                 .END
```

```
ERRORS = 00000


SYMBOL TABLE

SYMBOL VALUE
 BASF1   AD8A    C       033F    C1      0345    C1ISIT  C0DF
 C2      0346    C2ISIT  C0FD    CHECK   C49B    CHKFIR  C4F6
 CHKPLY  C4D9    CHRGET  0073    CMP1    C4E3    COMPAR  FFFF
 DECX    C537    DIR1    035D    DIR2    035E    DIR3    035F
 DNE     C116    DNTMES  C1E4    DOWN    0341    DWN     C520
 DXFLOT  0058    DYFLOT  0062    EATCOM  AEFD    EN1     034C
 END     C34D    F1DXFL  C133    F1DYFL  C136    F1F2    BC0F
 F1FIX   B7F7    F1L     C14D    F1LFL   C139    F1XFL   C12D
 F1YFL   C130    F2ADD   FFFF    F2DIV   FFFF    FILL    C027
 FILLIT  C2C5    FILMOR  C02D    FIND    C038    FINDIT  C29B
 FIXF1   FFFF    FLOOP   C2A5    GETCOL  C117    GETJOY  C4A8
 GETKEY  C4BA    GETXY   B7EB    GETY    B7F1    GOON    C279
 INSERT  C367    JOY1    0360    JOY2    0361    KEYEN   034B
 L10     C319    L11     C33B    L5      C2B1    L6      C2D8
 L7      C2DD    L8      C2F3    L9      C2F8    LEFT    C526
 LFLF1   C146    LFLOT   0067    LHI     034A    LINE    C194
 LINEOK  C1E1    LLO     0349    LOOP    C293    M1OFF   C415
 M2OFF   C47C    MLTI    C0C3    MM1     C3FE    MM2     C465
 MOVE    C51A    MP1     C41D    MP2     C484    MU      0344
 NEXT    C20A    OFF     0348    OFFF    C095    ON      0347
 ONN     C113    PAINT   C254    PEEK    C343    PLOT    C15E
 PLOTIT  C181    POINT   C175    POK     C4EC    PUSH    C34E
 PUTCOL  C120    QUIT    C253    QUT     C4A5    RETURN  C54C
 RIGHT   C53B    SCREEN  C00C    SETUP   C09D    SGNF1   BC2B
 SKIP    C086    SKIPP   C504    SKP     C050    START   C287
 STKPTR  0343    STORE   C3A6    SUBT    B853    TABLE   C000
 TARGET  C48E    TBL1    C50A    TBL2    C512    TRYC2   C0E6
 TRYC3   C104    UP      0342    VARHI   0003    WDGOFF  C3AD
 WEDGE   C3C2    XFLF1   C140    XFLOT   004E    XHI     033D
 XI      0340    XLO     033C    YF1     B3A2    YFIX    033E
 YFLF1   C143    YFLOT   0053

END OF ASSEMBLY
```

APPENDIX E: DESIGN GRIDS



"X,Y PIXEL POINTS"


X
−X, −Y
+X, −Y
−X, +Y
+X, +Y
-153 -145 -137 -129 -121 -113 -105 -97 -89 -81 -73 -65 -57 -49 -41 -33 -25 -17 -9 -1 0 8 16 24 32 40 48 56 64 72 80 88 96 104 112 120 128 136 144 152
-89 -81 -73 -65 -57 -49 -41 -33 -25 -17 -9 -1 0 8 16 24 32 40 48 56 64 72 80 88 96
-89 -81 -73 -85 -57 -49 -41 -33 -25 -17 -9 -1 0 8 16 24 32 40 48 56 64 72 80 88 96
VISIBLE SCREEN

SPRITE DESIGN GRID (TOP)

	A								B								C							
ROW #	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								
	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1

DATA STATEMENTS

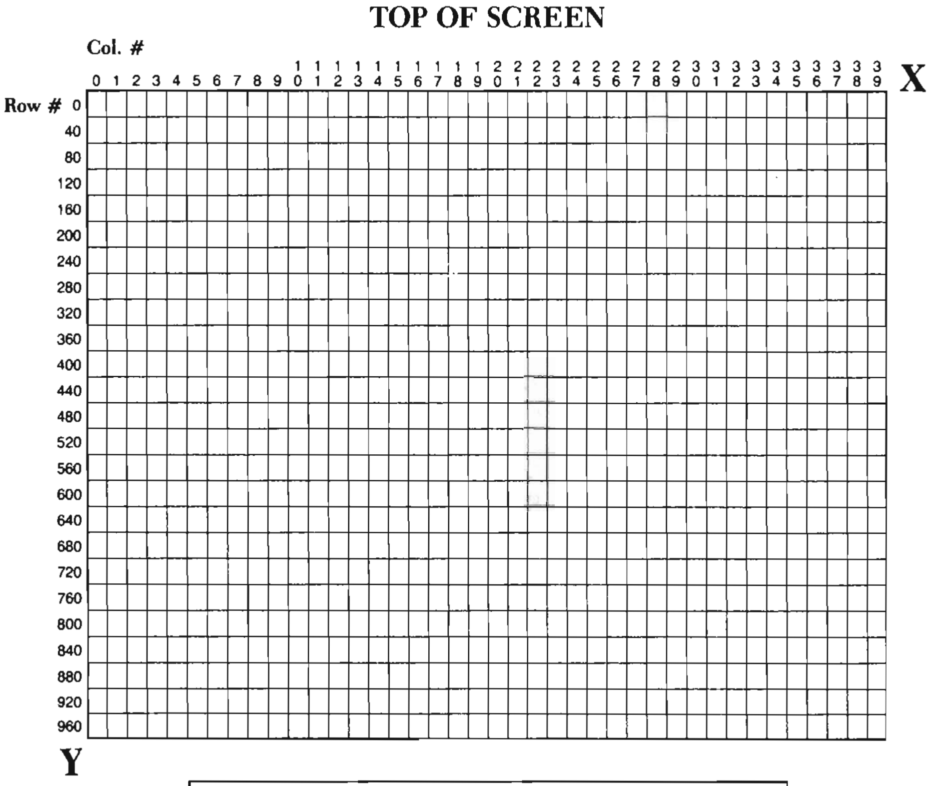
BASIC LINE #	DATA	SUM OF A	SUM OF B	SUM OF C
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			
	DATA			

APPENDIX F: COLOR CHARTS

There are sixteen available colors on your Commodore 64. These colors, and their corresponding color codes, are:

COLOR	CODE
Black	0
White	1
Red	2
Cyan	3
Purple	4
Green	5
Blue	6
Yellow	7
Orange	8
Brown	9
Red	10
Gray 1	11
Gray 2	12
Green 2	13
Blue 2	14
Gray 3	15

The 1,000 screen color blocks are diagrammed on the grid below.



COLOR BLOCK # = COL. # + ROW #

MEM. LOC. = 17408 + COLOR BLOCK #

COLOR MEMORY

APPENDIX G: TOOL KIT REFERENCE CARD

Under "How To Use" you will find that all variables are set equal to "#". See the back of this card for the value ranges allowed for each variable. Note also that most of the tools listed below require a DIM statement of:
"DIM P%(99,2),R%(99,2),L%(99,1), C(2,2),T(2,2),W(2,2)".

Tool	Description	How To Use
10	TURN ON GRAPHICS	MU=#: GOSUB 10
20	RETURN TO TEXT	GOSUB 20
30	CLEAR HIRES/MULTI	C=#: GOSUB 30
40	PLOT A POINT	X=#: Y=#: C=#: GOSUB 40
50	PLOT A LINE	X1=#: Y1=#: X2=# Y2=#: C=#: GOSUB 50
60	PAINT A SHAPE	X=#: Y=#: C=#: GOSUB 62 *or* PP=#: C=#: GOSUB 60
70	CLIP A SHAPE	(SEE TOOL 90)
80	DRAW A SHAPE	C=#: GOSUB 90 (SEE TOOL 800)
90	DRAW A SHAPE	ND=#: NL=$: C=#: MU=#. SEE TOOL 800.
100	APPLY TRANSFORMS	(SEE TOOL 90)
110	CLEAR C MATRIX	GOSUB 110
120	CLEAR T MATRIX	GOSUB 120
130	COMBINE MATRICES	(SEE TOOLS 140,150,160)
140	TRANSLATE A SHAPE	XT=#: YT=#: GOSUB 140
150	SCALE A SHAPE	XS=#: YS=#: GOSUB 150
160	ROTATE A SHAPE	RO=#: GOSUB 160
170	ZAP!	(DON'T USE WITHIN PROGRAM) *Type RUN 172*
180	TURN ON SPRITE SP	SP=#: GOSUB 180
190	TURN OFF SRITE SP	SP=#: GOSUB 190
200	X EXPAND SPRITE SP	SP=#: GOSUB 200
210	X UNEXPAND SPRITE SP	SP=#: GOSUB 210
220	Y EXPAND SPRITE SP	SP=#: GOSUB 220
230	Y UNEXPAND SPRITE SP	SP=#: GOSUB 230
240	SP PRIORITY OVER SHAPES	SP=#: GOSUB 240
250	SHAPE PRIORITY OVER SP	SP=#: GOSUB 250
260	SET SPRITE TO COLOR C	SP=#: C=#: GOSUB 260
270	PLACE SPRITE AT X,Y	X=#: Y=#: SP=#: GOSUB 270
280	MOVE SP FROM X1,Y1 TO X2,Y2	X1=#: Y1=#: X2=#: Y2=#: SP=#: SD=#: GOSUB 280
290	HOOK UP ACTION SPRITES	KB=#: P1=#: P2=#: M1=#: M2=#: T1=#: VE=#: GOSUB 290
300	COLLISION DETECTION	GOSUB 300
310	RESET COLLSION REGISTER	GOSUB 310
320	SUSPEND GAME	GOSUB 320
330	RESTART GAME	GOSUB 330
340	CRASH SOUND ON	GOSUB 340
350	SOUND OFF	GOSUB 350

7/11/87

360	COLLISION PUNISHMENT	SP=#: GOSUB 360
800	RETRIEVE A SHAPE	SE$=#: GOSUB 800
810	RETRIEVE A SPRITE	SE$=#: SP=#: GOSUB 810

Variable List

The following variables are commonly needed by this book's subroutine tools:

Variable	*Description*	*Value Range*
MU	Multicolor Indicator	0 = Hi-Res, 1 = Multicolor
C	Color	0 To 15
X	X Coordinate	0 To 319
Y	Y Coordinate	0 To 199
X1	X Coordinate Endpoint 1	0 To 319
Y1	Y Coordinate Endpoint 1	0 To 199
X2	X Coordinate Endpoint 2	0 To 319
Y2	Y Coordinate Endpoint 2	0 To 199
PP	Paint point coordinates' position in P% array	0 Based
XT	Translate Along X	———
YT	Translate Along Y	———
YS	Scale Along X	———
YS	Scale Along Y	———
RO	Rotation Degrees	———
SP	Sprite Number	0 To 7
KB	Keyboard Enable	0 = Joysticks, 1 = Keyboard
P1	Player 1 Enable	0 = Disable, 1 = Enable
P2	Player 2 Enable	0 = Disable, 1 = Enable
M1	Missile 1 Direction	0 = Disable, 1 = Up, 2 = Down, 4 = Left, 8 = Right, 5 = \, 6 = /, 9 =/, 10 = \.
M2	Missile 2 Direction	0 = Disable, 1 = Up, 2 = Down, 4 = Left, 8 = Right, 5 = \, 6 = /, 9 =/, 10 = \.
T1	Target Direction	0 = Disable, 1 = Up, 2 = Down, 4 = Left, 8 = Right, 5 = \, 6 = /, 9 =/, 10 = \.
VE	Game Speed	0 = Fastest, 65 = Slowest
SE$	Search String	———

COMMODORE 64 COLOR GRAPHICS: AN ADVANCED GUIDE

Designed by Ellis/Ross/Tani, Los Angeles.

Commodore 64 Color Graphics: An Advanced Guide is a step-by-step guide to creating advanced animated graphics on your personal computer.

The easy to follow yet comprehensive programs can take any beginning programmer into the world of *high speed* graphics. You'll be amazed how rapidly you will be able to draw and paint your graphic displays. Machine language data turns time-consuming graphics tasks (such as plotting lines and painting shapes) into high speed magic.

Sequential instructions lead you through advanced graphics techniques. You'll learn how to quickly reposition shapes on the viewing screen, change their size, and duplicate their shape around a central point. Tools to perform these tricks are provided as well as helpful design ideas and art concepts to enrich your graphics compositions.

The book's final object is to produce an action arcade game using advanced sprite techniques. You are provided tools to allow you to connect sprites to joysticks, detect sprite collisions, produce sound, and keep score. When you are ready to create your own arcade game you'll have a head start with your new game construction tools.

As a special bonus, appendices include all the tools you'll need to save an actual picture (not your program) on disk or tape and print your pictures on paper with your VIC1525 printer.

The Reston Computer Group
Reston Publishing Company, Inc.
11480 Sunset Hills Road
Reston, VA 22090

$14.95
U.S.A

ISBN 0-8359-0787-2