

CREATING
**ARCADE
GAMES**
ON THE
VIC

A step-by-step guide to creating your **own** arcade game on the VIC,[®] plus four excellent, finished games which show the process from the first idea to the completed program.

Robert Camp



CREATING
**ARCADE
GAMES**
ON THE
VIC

Robert Camp

COMPUTE! Publications, Inc. 
One of the ABC Publishing Companies

Greensboro, North Carolina

VIC-20 is a trademark of Commodore Electronics, Ltd.

Copyright 1984, COMPUTE! Publications, Inc. All rights reserved.

Reproduction or translation of any part of this work beyond that permitted by Sections 107 and 108 of the United States Copyright Act without the permission of the copyright owner is unlawful.

Printed in the United States of America

ISBN 0-942386-25-6

10987654321

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is a subsidiary of American Broadcasting Companies, Inc., and is not associated with any manufacturer of personal computers. VIC and VIC-20 are trademarks of Commodore Electronics, Ltd.

Contents

Foreword	v
Chapter 1: The VIC Is a Game Machine	1
Chapter 2: Game Design	17
Chapter 3: Setting Up Your Screen	41
Chapter 4: Custom Characters	61
Chapter 5: Getting Your Figures Moving	77
Chapter 6: Collisions	99
Chapter 7: Sounds and Music	115
Chapter 8: Introductions, Instructions, and Farewells	133
Chapter 9: The Shape of the Game	145
Chapter 10: Missiles and "Moonraker"	157
Appendices	167
A. A Beginner's Guide to Typing In Programs	169
B. How to Type In Programs	171
C. Screen Location Table	173
D. Screen Color Memory Table	174
E. Screen Color Codes	175
F. Screen and Border Colors	176
G. ASCII Codes	177
H. Screen Codes	181
Index	183



Foreword

Most people who play arcade games take them for granted. But when you try to program such a game on your VIC, you quickly learn to appreciate the amount of work and creativity behind every game. Figures on the screen, movements or collisions, sounds all must be fit together into a complex, yet harmonious whole. It can easily seem impossible, especially if you're just starting.

Until now, most game books for personal computers have been simple listings of programs which could be typed in and run. Now you can learn the principles and techniques of writing your *own* games.

With this book — and your creativity and imagination — you'll quickly be developing and writing games. Each step of the process is outlined in clear and detailed language, including examples from the four complete games included in the book. The VIC-20's sound and graphics capabilities are unlocked in ways you knew were possible, but learning how to do the programming was another matter.

As with all COMPUTE! books, this guide will be useful not just once (when you write your first game), but again and again, as you refer back to it for more information and more complex game ideas and techniques.



Dedication

To my wife Susan, who put up with those late nights on the VIC.





1
**The VIC Is a
Game
Machine**



1

The VIC Is a Game Machine

I looked for a book to tell me how to plan a game, how to make figures move across the screen, and how to make things happen when they collided. I wanted to know how to work sound into the program, how to rack up scores, how to make shapes, and on and on. I needed a book that would tell me, a new VIC user, exactly what I needed to know for my specific purpose: game design.

After spending hundreds of quarters on video arcade games, I decided it would be a better investment to buy a computer. Then I could play games as often as I liked at home, without waiting for other people to finish playing, and without having to spend a lot of money before I finally got good enough to beat the machine.

But I had more than convenience in mind. I didn't just want a videogame machine. I wanted to write my own games, maybe a game so good that it too would end up in the arcade, among the others that I had enjoyed so many times.

After sitting down with my VIC-20 for about an hour, I realized two things. First, programming wasn't exactly easy. There were many things to think about, many things to remember, and the computer wasn't very forgiving of my mistakes. And the second thing I noticed was that the VIC owner's manual didn't help as much as I wanted it to.

The story has a happy ending. Eventually I learned how to program games. And in this book I've done my best to provide you with exactly the book I had wished for so many times. I'm assuming that you're reasonably familiar with the basics of BASIC — how to use PRINT and GOTO, FOR-NEXT loops and GOSUB-RETURN subroutines. If you aren't comfortable with those, you might want to scan your owner's manual, or at least keep it handy in case questions come up while you're creating games. What this book provides is the specific programming techniques you'll need for game creation.

The Design of the Book

The place to start when creating a game is to plan how you want the game to play. So in Chapter 2, you'll first design what will actually go on between the computer and the player. When you have a pretty clear idea of what you want to end up with, it's much easier to get the computer to give you exactly the right results.

Once you have your game in mind, the next few chapters will introduce all the BASIC programming tools you'll need. How to set up a screen. How to design custom character sets and make the characters move. How to set up relationships between different figures on the screen. The uses of sounds and how to produce them. And how to help the player understand what he's supposed to do.

With the basics covered, you can go on to spend some time refining certain techniques, polishing your programs, learning strategies to make your programs use less memory and run faster.

How to Use This Book

Each chapter will include different problems to solve and questions to answer. Sometimes you won't need to actually try the sample program — it may be something you've already learned. But many things can't be explained. They have to be shown. And since I won't be leaning over your shoulder while you sit at your computer, the only way I can show you is for you to type in example programs or work out problems.

If you get stuck on any problem, remember that there are usually several ways to solve it, and sometimes the sample solution is used because it's the easiest for a beginner to understand, not because it's necessarily the fastest or most useful way to solve the problem. If you come up with a solution that works, then your answer is as right as mine.

The only way to really learn a programming technique is to try it out. In fact, the most helpful thing you can do for yourself is to experiment. Once you've tried the example I give you, think of problems for yourself and try to solve them using the techniques you've just learned. After all, you're learning a language — and languages are only learned through practice.

Besides, the fun of programming isn't in reading a book — it's

in the programming, giving the computer instructions and seeing what happens. The point of this book is to help you have fun making games, so the more creating you do while reading the book, the more value it will have for you.

Be Patient

Your learning process won't be just like mine. You'll grasp some things faster than I did; other things may take more effort. One thing is sure — unless you're already an expert programmer, programming a game is going to take *time*. Not only the time to type the program lines in the first place, but also the time it takes to figure out why the game isn't working. Because unless you're a miracle worker, your program is going to have some bugs in it, some places where things aren't happening the way you planned.

So be patient with yourself. Even with the help of this book, programming takes a while to learn. Don't expect to learn everything in a few hours one night. And don't expect your first program to run smoothly. If you're like me, there'll be more than a few nights when you sit there staring at some program lines, trying to figure out why the program crashes whenever it gets there, only to discover that you typed a zero instead of the variable `O`; or that you forgot to have your countdown `FOR-NEXT` loop `STEP -1`, so it's only executing once; or that you put the wrong variable as the subscript of an array; or that your `GOSUB` refers to a line that you deleted in your last series of revisions.

That sort of thing happens to everybody. It's just one of the realities of working with a computer. Your VIC will always do exactly what you say — not what you really *meant* to say.

Look at Other People's Programs

In almost every computer magazine there are programs listed in full so readers can type them in and use them. At first those programs look like a lot of gibberish to a beginning programmer. But you've probably already noticed that some of those programs begin to make sense to you. You can see at a glance what's going on in certain subroutines; you can spot where loops begin and end, and what they're accomplishing.

You can learn quite a bit about programming just by typing in other people's programs. Besides getting typing practice, you can see how *they* solved particular problems.

You'll notice recurring patterns, techniques you can use. But you'll also notice places where they took the long way to solve a problem, where you know a shortcut. I remember how surprised I was to see widely published programs using some roundabout, slow methods that I wouldn't have used.

Then, when you have their program typed in and saved on tape or disk, experiment with it, just the way you'd experiment with the exercises in this book. If you don't like the colors they used, find where the program assigns the color values and change them. If you want a ghost turned into an octopus, find out where the program sets up the character set and redefine the characters. Sometimes your experiments may crash the program, but then you can figure out *why* and do better on the next try.

One of the biggest advantages, you see, is that you have a complete program that already works. It's easier to find out and learn from your mistakes when you know the error must be in the two or three lines you changed.

Write Down Your Ideas

While you're reading this book, especially after an hour or so of steadily working at the computer, you're going to start getting some ideas. That's the way creativity works — when your mind is working hard on something exciting, the ideas start to flow.

Often it will be an idea completely unrelated to the problem you're working on at the moment. It might be an idea for a special effect, or a game scenario, or a movement pattern — anything at all.

You should write it down. Because once that creative mood has passed, it's often as hard to remember a good idea as it is to remember a dream as it slips away from your conscious mind. And those ideas are valuable. Not all of them will be fantastic. Some of them may not even work.

But game design isn't just programming techniques. There are plenty of excellent programmers who still can't come up with a game worth spending a single quarter on. Why? They've mastered the skills, but they don't know what a player will have *fun* with. And yet there are other programmers who are sometimes clumsy and disorganized in their programming, but their ideas are so good that the players will stand in line to play their games.

In fact, some videogame manufacturers have recognized the difference between game *design* and game *programming*, and split the jobs. The game designer writes down, in English, an exact

description of every single thing that happens in the game — what happens when you push the joystick up, down, left, right; what the button does; what happens when object A and object B collide on the screen; how many points each goal should be worth; how many seconds or fractions of a second each activity should take; and so on.

When it's all there, on paper, the designed game is given to the programmers, who do their best to create a program that does exactly what the designer specified. Sometimes there have to be compromises, because some ideas just can't be executed within the available memory or within the available time. And the game is certainly as much the creation of the programmer as of the designer.

But it's a different kind of creativity. And even though I can try to stimulate you with this book, I can't teach you how to think up good game ideas the way I can teach you how to redefine a character. That's something you have to develop yourself.

So save every idea you think of. You never know what it might lead to. And it's your ideas more than your programming skill that will make the difference between a run-of-the-mill game and a great game that players can't wait to play again.

| It Isn't Done When It's Done

What happens when you've taken your game from the first idea to a working game? All the bugs are out of the program, and it works just the way you planned it. What now?

Well, if you were creating the game just for yourself, and you're satisfied with the way it plays, then congratulations — you're finished.

But if you want to offer the game to commercial software houses or to magazines that publish games, you still have a few things to do.

Test the Game

Play the game awhile. How fun is it for *you*? If you're already bored with it, it may need some work.

Better yet, invite the world's toughest critics to test the game for you. They live right in your neighborhood — any kids who play arcade games will do. Tell them you've got a game you're in the middle of programming, and you need players to test it. You probably won't lack for volunteers.

1 The VIC Is a Game Machine

Then watch them as they play. Don't ask questions; don't explain; don't ask for their opinion. Friends might tell you a game is better or worse than it really is — after all, they aren't game designers. Just watch them play, and ask yourself these questions:

1. Where in the game do they get puzzled, confused, annoyed? That's exactly where your directions need to be clearer.
2. Are they excited, or at least interested? If they aren't enjoying themselves, is it because the game is too easy or boring, or is it because the game is too hard and frustrating?
3. How hard is it for them to get the hang of player movement? How responsive are your controls?
4. Watch how the scoring goes — are they beating the game right away, or getting nowhere?
5. How long do they play? Is the game over instantly? When it's done, do they immediately want to play it again, or do they ask you if they have to play it again?

The surest test of a successful game is if they don't want to quit. The surest test of a game with problems is if they come to you within a few minutes and ask, "Got any other games?"

Improve It

Just because the program runs doesn't mean the game is perfect — it only means the program is perfect. After your own tests and the neighborhood-kid tests, what can you think of that would make the game more fun, or make it stay interesting longer?

If you can think of some things, analyze your program and see how easy it would be to make the changes. If it's feasible, go ahead and try the improvement. But make sure you save a couple of tape or disk copies of the program that ran correctly, in case your improvements go so far off track that it's easier to go back and start over.

Market It

This might be a little premature, since your game isn't written yet, but it doesn't hurt to know in advance what the market possibilities are.

A lot of commercial software companies are eager for games, and they're especially eager for games for the VIC. Also, quite a few magazine and book publishers have discovered that their readers enjoy typing in and playing games. Once your game is ready, there might well be a software distributor or a publisher who'll be glad to purchase the publication or software rights from you.

How do you submit the game? With the software houses, send a letter describing your game, in detail, from the player's point of view (that is, what happens during the game, how you win) and listing the memory requirements and machine specifications. You can send such a letter to a lot of different software houses. Then submit a copy, on tape or disk, to the first distributor that shows an interest. If that doesn't pan out, then send a copy of the program to the next interested distributor.

If a software company wants the game, they will offer you a cash advance and a royalty. If they offer a single flat payment, and no royalty, don't accept — if the game sells a million copies, you deserve a percentage of the money it earns.

Or you may prefer to submit your game to a magazine or book publisher. Submit it with complete documentation and two machine-readable copies of the game (the same tape, front and back, will do). Magazines will pay you a flat rate for one-time use; book publishers will usually pay you an advance and a royalty.

Some rules about submitting software:

1. Never send the actual software to two companies at once. Publishers and software distributors assume, when they go to the time and expense of paying someone to test and evaluate your game, that they are the only company being offered the game at that time. (The letter that you send to many companies at once is not a submission. It is an inquiry, to find out if they want to see the game in the first place.)

2. Give them time. You're anxious, of course, to hear whether they want to use the game or not, but they have dozens or hundreds of submissions to look at, and they have no way of knowing that yours is the greatest videogame of all time until they get around to testing it. But if you haven't heard anything in six to eight weeks, it doesn't hurt to send a letter asking if they've had a chance to look at it.

3. Be flexible. Especially if you're a new programmer, it's quite possible that the editors will see real possibilities in your game, but will want improvements. They may make suggestions to you — it won't hurt you to try to make improvements. If the game needs to be translated into machine language, they may ask you to allow another person to work on your game — it's certainly worth considering. They may want to buy just the story or ideas, and incorporate them into a different game — that's high praise for your creativity, and not at all a slur on your programming skills. Or they may want to use a compiled version of the game, so

it plays faster. If you're willing to compromise, your chances of selling the game are much better.

4. Be professional. Reputable companies deal fairly with people who submit games — after all, you're the lifeblood of their business. Don't assume that anyone's going to try to steal your game — they aren't. Don't assume that a long delay means they don't like the game — often a delay means that it's getting serious consideration. And your game will always get serious consideration. Publishers and software houses really want your game to be brilliant. Every time they open a submission, they hope that this is a blockbuster game like *Space Invaders*, *Asteroids*, *Pac-Man*, or *Donkey Kong*.

And who knows? Maybe it will be.

What You Should Have on Hand

In writing this book, I'm assuming that you have available some basic resources:

1. A VIC-20.
2. A Datassette or a disk drive. After all the work of programming, you certainly want to be able to save the game and play it again and again.
3. *Personal Computing on the VIC-20*. It came with your computer.
4. *The VIC-20 Programmer's Reference Guide* (published by Sams) or *VIC-20 User Guide* (published by Osborne). Available at most bookstores, and extremely valuable in problem-solving.
5. Plenty of cassettes or disks to save your programs. If you're using cassettes (as most of us VIC-users do), I suggest that you use a low-grade, low-noise tape. They're cheap and work better for computer purposes than high-quality *sound* cassettes. And you'll always want to have some extra cassettes on hand, so you can store different versions of the same program. All it will take is one time when you have to quit in the middle of a revision and you don't have a single blank cassette to use. You don't want to wipe out a working program to save a half-finished job — but you also don't want to have to start that job over again.
6. Graph paper. The best is the kind that emphasizes 8x8 groups of tiny squares.
7. A six-inch ruler. This is very helpful when you're typing in a program out of a book. By putting it under the line you're typing, you make sure you don't accidentally skip down into an-

other line with similar commands.

8. A pocket calculator. I know, you have a few bucks worth of computer right there in front of you. But many times it's a lot simpler to use a calculator, which is designed to do nothing but arithmetic, than to set up PRINT statements on the VIC in order to solve a problem.

9. A color TV. How are you going to design color games on a black-and-white monitor? I paid only \$130 for a used 19-inch TV with remote control — you don't have to break the bank to see your games in color.

Introducing the VIC-20

If you just got your VIC and aren't really familiar with it, then let's spend a page or two getting acquainted with the machine. If you're already familiar with the computer, then you can probably skim or skip the rest of this chapter and go right on to Chapter 2.

RAM

The more I get to know the VIC, the more I realize what a powerful computer it really is. Its RAM (Random Access Memory) starts at 5K (about 5000 bytes) and can be expanded to 32K (32,768 bytes). Every letter or symbol you see on the screen, every program instruction, every number you use takes up at least one byte of RAM. That means that the more RAM you have, the larger and more complex the programs you can run.

Even the unexpanded VIC is a powerful computer — on the original vacuum-tube computers at the beginning of the computer age, 5K would have taken 40,960 vacuum tubes. That little box you have in front of your TV would have looked like a miracle back then.

But it's only natural that the 5K VIC won't be able to handle as complex and detailed a game as a VIC with more memory, especially if the game is written in BASIC, which takes up more memory than a machine language program. Since we're working with the basics in this book, we'll deal exclusively with BASIC programs, and since I want this book to be valuable to the largest number of readers, almost everything we do here can be done in 5K.

That means that the sample programs here will be pretty simple. But you'll be learning all the *principles* of VIC game programming, so that if you have more than 5K on your computer, there's

1 The VIC Is a Game Machine

nothing to stop you from combining all the things you learn into really dazzling 32K games.

And if you are working with only 5K, you'll soon find that while your graphics displays might not be as pretty as those in the arcade games, and your animation might not be as detailed, you can make games that play very well and stay fun and interesting for a long time.

Besides, limited memory just forces you to be more creative, finding ways to fit as many features into that 5K as you can. You don't have to rush out and get an expander right away. It might even be a good idea to work with the 5K for a while, learning methods of shortening your programs. It can help you develop good programming habits that you'll find useful even with larger memory. You'll know when you really *need* the expander — and by then the price will probably be half what it is right now.

The Screen

The VIC screen is 22 columns wide and 23 rows from top to bottom. That's fewer rows and columns than many other home computers, which is why each letter is bigger and takes up more space. In effect, this means that each display screen is "smaller" than with other computers — the screen displays fewer characters at a time.

But that isn't necessarily a disadvantage. A 22-by-23 screen means you have 506 squares to work with. That's a lot.

Besides, each square is really an 8-by-8 grid of 64 smaller squares. That's how letters are made — by putting one color in some of those squares, and another color in the rest of them. The pattern makes the letter shape. Since the VIC displays 506 blocks of 64 pixels, you really have 32,384 pixels at your command. That's high-resolution graphics!

With the VIC, you can design your own character shapes. And since each character block is bigger than those on most computers, you can change larger areas of the screen display with each single command. What you may lack in fine-line drawing you have gained in speed and memory space.

Colors

With a color TV or monitor, the VIC can produce 16 different screen colors and 8 character colors. The background, border, and characters can have different colors, and in multicolor mode the characters don't all have to be the same color, either, allowing you hundreds of possible color combinations.

Graphics Characters

You can redefine characters — but that does use up memory. Don't forget that the VIC provides 62 built-in graphics characters. You can save a lot of memory by using as many of those existing characters as you can before you start defining new ones. Many a great game can be designed without a single character redefinition.

Sound

With the VIC's four voices, you can get a variety of sounds and sound effects, from explosions and lasers to a Bach fugue. Sounds are not really extras in a game. They serve several vital purposes, and the VIC gives you plenty of sound resources to work with.

Languages

The VIC has two languages easily available, BASIC and machine language.

Machine language consists of eight-digit binary codes that give instructions to the CPU (central processing unit), which is the real brain of the machine. The binary number 10000101 (decimal 133) tells the CPU to store a certain value in a certain location.

Since machine language is the CPU's native tongue, so to speak, the computer understands it very quickly. That's why games written in machine language are very fast and run very smoothly.

The trouble is that numbers like 10000101 aren't too easy for human beings to work with. Even when they're translated into hexadecimal (base 16) numbers, they don't carry much meaning. So machine language programmers usually write their programs with an assembler program, which uses a series of three-letter *mnemonics* which stand for the machine language commands. *STA* is the mnemonic for 10000101. The assembler program then translates symbols like *STA* into machine language numbers. It's that translated program that becomes the finished game.

However, machine language programming is very tedious and very complex. Especially for beginners, BASIC is by far the easier language for programming. The commands are more like English, they're easier to remember, and it takes fewer commands to perform each operation.

What you get in ease of programming, however, you pay for in running time. While your BASIC program runs, BASIC has to

1 The VIC is a Game Machine

translate each command into machine language every single time it is used, and that takes up a lot of time. BASIC programs generally run slower.

But BASIC is only slow by comparison with machine language. It is a very fast and powerful language in its own right, and the more you work with it, the better you'll do at writing fast, effective games in BASIC.

Once you are familiar with programming in BASIC and have a good idea of what you can get the computer to do, it is much easier to progress to machine language. You'll probably begin by writing short machine language routines to include in your BASIC programs, so that you can get the speed of machine language in a few places where you really need it. But machine language is beyond the reach of this book. Once you learn how to make the VIC play good games, it's up to you to decide what language to use.

The 6502

The VIC has a brain made of silicon. The CPU (central processing unit) is the famous 6502 microprocessor. MOS Technology, the company that created the 6502, is now a subsidiary of Commodore, the company that made your VIC. The 6502 is now the CPU in the Apple, the Atari, and other microcomputers. Because Commodore owns the 6502 and many of the other special chips in the VIC, they can offer the VIC at a lower price than if they had to buy the 6502 from somebody else.

Program Translation

Just because Apple, Atari, and Commodore all use the same CPU doesn't mean that you can run programs written for those machines on your VIC.

For one thing, VIC BASIC is different in small but important ways from the BASIC used by the other machines.

Also, the CPU doesn't control *everything* going on in the computer. Each computer has its own way of communicating with cassette recorders, disk drives, keyboards, and the screen.

Each memory location can mean different things on different computers. When the 6502 stores a 1, for instance, at location 50 in RAM, that might tell one computer to turn off the TV screen, while another computer would take that as a command to erase memory and start over.

However, as long as a BASIC program sticks to generally accepted BASIC commands and doesn't PEEK and POKE into

specific memory locations, it's quite possible to translate a program written in one computer's BASIC into another computer's BASIC with some relatively simple changes.

Unfortunately for game designers, however, BASICs often differ the most in graphics and sound handling, which are vital to game programming. By the time you finish translating the game, you've practically created a whole new program. That's why few people bother to go through the translation process. It's often just as easy to start programming from scratch.

Which is exactly what we're going to do in Chapter 2.





2
Game
Design



2

Game Design

If you don't know where you're going, how will you know when you get there?

It's important to plan your game *before* you start programming it. If you have a clear idea of how the finished game should play, you'll also have a pretty good idea of what needs to happen in your program.

Some things are obvious. If you're going to use the joystick, you'll need to have a routine in your program that reads the joystick. If not, your program will have to read the keyboard to find out what the player wants to do.

Some things, however, are more subtle. What will happen if your player-figure touches the sides of the screen? Will it explode? Appear on the opposite side? Bounce off? Lose points? Slow down? Speed up? Get larger? Get smaller? Disappear? Cause five new enemy creatures to appear? Cause the enemy creatures to move faster or come closer?

There are hundreds of choices like that, and at some point you'll need to decide how your game program will handle them. If you make those choices beforehand, in the planning stage, then you can design a program that will do everything right — and will fit together the first time with a minimum of revision.

The Design of "Joust"

Let's begin by taking a closer look at one of the most popular arcade games. My own favorites are *Defender* and *Stargate*, but the same design principles apply to *Donkey Kong*, *Ms. Pac-Man*, or *Joust*. What is the computer actually doing during the game?

Words

Let's look at *Joust* in detail. When you put in a quarter, what happens? There are words on the screen, telling you what to do next. When you choose the one- or two-player option, there are more words. Remember, too, that there are instructions written on the outside of the game machine. All of these, combined, tell you how to operate the controls.

This is *documentation*. If it is written down on paper or on the machine, it is *printed* documentation. If it appears as part of the

video display during the game, it is *internal* documentation. Whichever form it takes, though, good documentation is vital to make a game fun to play. If there's nothing to help a player understand what he's expected to do, or if the instructions are inaccurate or incomplete, playing becomes frustrating instead of fun.

The Display

Then the words disappear from the main display area. What is on the screen? The *playfield* — the area where the game action will take place — consists of brown, rocklike islands. The bottom island rises from the base of the screen, with a lake of fire on either side. The other islands, though, float in midair. Several of them have white strips near the surface — these, the player will soon discover, are the platforms where new player-figures and new enemy knights will appear.

Besides the playfield, there are several other things on the screen. The score is constantly displayed, for either one or two players, so you can keep track of how many points you have to earn before the next bonus knight. Also, there is a little box that displays how many knights you have left; this lets you see at a glance how many more turns you'll have.

Movement and Animation

Now the action begins. Your player-figure, a knight mounted on an ostrich-legged bird, appears on the left side of the bottom island. When you hold the joystick to the left, the bird turns and starts to walk; keep holding it, and it runs in that direction. Then if you push the joystick right and hold it, the bird screeches to a halt, pauses, turns right, and then starts walking.

When you push the flap button, your mount's wings flap once. This makes your knight rise into the air a little, but almost at once he begins to sink. When you push the button repeatedly, however, the wings keep flapping and your knight rises higher and higher.

The realistic flapping of the wings, the motion of the legs, and the way the legs and wings seem to cause the figure to move are *animation*. The overall pattern of movement, the relationship between the player's joystick and button and the motion of figures on the screen, is *player movement*.

Collisions

Now other knights start appearing on the screen. They're computer-controlled characters that fly according to their own

patterns. As you move your knight around the screen, he starts bumping into things — islands and other knights.

If you collide with the bottom or edge of an island, or the top of the screen, you bounce off. If you were going fast when you hit, you go just as fast when you bounce off. Your bounce also takes into account the direction you were going when you collided — if you were going upward to the right when you hit the top, you go downward to the right after the bounce.

If you fall down onto the surface of an island, your mount extends its legs and starts to walk — regardless of how fast or how far you fell. Running into islands or the ceiling will never hurt you.

One oddity is that if you glide into contact with the surface of an island along an almost exactly horizontal path, the legs of your mount won't come down. You'll just bounce along without walking, belly-flopping your way across the screen.

When you collide with another knight, one of three things can happen. First, if your knight and the other one are on almost exactly the same level, you will simply rebound from each other with a loud noise.

If your knight is lower on the screen than the other knight, your knight will disappear, and a new one will come to life at one of the creation platforms.

If your knight is higher, it is the enemy that disappears. In his place, a large egg flies away from the mount, as the mount flaps quickly off the screen.

Your knight has a certain amount of time in which to go collide with the egg before a new enemy knight hatches out of the egg, and his mount flies out to join him. The egg trembles a bit before it hatches, and the enemy's mount does take a while to arrive and get in place, so there's plenty of warning — but if the new enemy knight gets mounted, he is more formidable than the one before. And the sooner you get the egg, the more points you get.

Intelligent Enemies

The enemy knights start out relatively slow and stupid, but even at the start they tend to notice where your knight is and home in on him. Also, the red and blue enemy knights are faster and harder to beat. They have a habit of maneuvering right under the edge of an island, or right along the top of the screen, so that it's hard for you to get above them. They also use the island and screen for rebounding — they fly quickly upward and then re-

bound downward so they come at you faster than you expect and from a direction you weren't prepared for. If you follow their strategies, you'll do better against them.

Other Hazards

From the third level onward, the sea of lava at the bottom of the screen is uncovered. If your knight falls in, he is consumed. If he comes too close while moving too slowly, the lava troll reaches up and seizes him, pulling him downward into the lava. The hand sometimes seizes the other knights, but, after holding them awhile, always lets them go. While they are being held, however, they are vulnerable to you.

Creation

What happens after one of your knights has disappeared? One of the platforms turns yellow, and your knight rises into view. Unlike the enemy knights, yours does not immediately fly away. If you don't move, you have a few seconds in which your knight cannot be harmed, to wait for enemy knights to leave you a clear section of sky in which to take off.

You'll also notice that usually your new knights arise from a platform in the area where the fewest enemy knights are flying. Likewise, when enemy knights are being created, they will not usually appear at a platform if your knight is hovering too close.

Increasing Difficulty

The longer you play, the harder it gets. The most noticeable change is that at higher levels there are more enemy knights. Also, the knights move faster and make more erratic, unpredictable movements. They also get smarter and evade you better.

If you take too long at any one level, a pterodactyl appears (though I prefer to think of it as a phoenix). The pterodactyl flies back and forth across the screen and doesn't home in on you very accurately, but if you're careless it will get you — and then it doesn't help you to be above it, for the pterodactyl is "indestructible."

Or is it? The message on the screen says, "Beware the indestructible (?) pterodactyl." That question mark leaves some doubt, and you eventually learn that if you meet the pterodactyl head-on, with your knight's lance exactly at the level of the monster's mouth, you can kill it. At later stages, the pterodactyl comes on the screen almost at once, and homes in on you faster and more accurately.

At higher levels, the platforms begin to disappear, which makes the game more of a free-for-all, with fewer places to hide, until finally you have no more hiding places at all.

Increasing Interest

If a game stays the same from beginning to end, except for getting harder, it soon becomes dull. *Joust* adds some new elements to increase interest. There is a Survival Wave. If you can keep the same knight alive throughout that wave, you get a substantial bonus. If you are playing with someone else, you have waves of competitive play and cooperative play, in which you get bonuses for hitting each other or for *not* hitting each other.

There is also an Egg Wave, in which, instead of enemy knights, all their eggs are strewn on the platforms. It is difficult but not impossible to get all the eggs before they hatch. But when the enemy knights hatch, they are significantly better fighters than they were in the round before.

The pterodactyl, the lava troll, and the disappearing islands also serve to change the game and add interest.

Incentives

To encourage you to play again, improving each time, *Joust* gives you a score for your achievements in the game. Each egg you get is worth 250 points more than the last one, starting with 250 points and peaking at a maximum of 1000 points per egg. You also get a bonus if you get the egg quickly.

If your knight is destroyed and a new one appears, the scoring starts over at 250. Likewise, the egg scoring starts over with each new wave.

You get 1000 points for killing a pterodactyl.

Every 20,000 points, you get a bonus knight. This means that the better you play, the more chances you have to play even longer.

If your score is higher than the lowest score on the vanity board (list of best players), you get to enter your initials at the end of the game.

Sounds

Each event has its own sound. A collision in which you win has a different sound from a collision in which you lose. Bumping into different surfaces makes different sounds. There are sounds for knights arising from platforms, for eggs hatching and birds com-

ing in, for the pterodactyl's arrival, and for collisions with eggs. There are different sounds for walking and for flying. There is a sound to tell you when you get a bonus knight.

Even while you're concentrating on your own knight, trying to keep him alive, the sounds tell you a lot about what's going on elsewhere on the screen.

Story

Everything we've analyzed so far is detail, the bits and pieces that come together to make up the whole effect. But much of the fun in *Joust* is the story. You get to be a knight on a fantastic steed, flying rapidly among floating islands, jousting with dangerous opponents and evading or attacking mythical monsters. It's a fascinating world, a fantasy tale that you are acting out as you play. Whether you get a hundred thousand points or a tenth of that, you still get the pleasure of doing something with the computer that you could never do in real life. It is a pleasure just to visit the world of *Joust*, and that sort of pleasure can also be part of *your* game.

Complex Programming for Simple Games

When you analyze a game, the way we have just analyzed *Joust*, you can see that the simple, smooth play that you enjoy so much is actually the result of careful, meticulous planning and programming. Every single effect, every single rule of the game was planned and programmed. What happens if A collides with B, or with C, or with D? How does the game get harder as it goes along? All these things go into making a game that plays well the first time — and keeps on being fun the hundredth time you play.

Your first game doesn't have to be a *Joust*. It takes machine language, for one thing, to keep so many different figures moving around on the screen at those speeds. But even in fairly simple, slow-moving games, you'll need to take into account every one of these things in order to create a fun, interesting game.

In fact, it wouldn't be a bad idea to go to an arcade and *study* some of the games. Stand behind a game wizard while he plays and write down the things that happen on the screen. How do the ghosts respond to the player in *Pac-Man*? How often do the girders bounce out during the elevator sequence in *Donkey Kong*? How are pickle movements different from frankfurter movements in *Burger Time*?

Once you become aware of how game creators have designed the games in the arcades, you'll be much better prepared to plan the same kinds of effects in your own games.

The Story of Your Game

Donkey Kong is the story of Mario trying to save his girlfriend from the gorilla. In *Missile Command* you are trying to save cities from an enemy attack. *Defender* pits you against alien invaders. In *Robotron*, you are trying to rescue little human figures from robot attackers. In *Asteroids*, you're in space, threading your way among asteroids while fighting off enemies. The stories range from almost abstract games, like *Qix* and *Tempest*, to fully developed stories like *Venture* and *Donkey Kong*.

Don't underestimate the importance of having a good story. For one thing, if your story is too close to the story of an existing game, you won't ever be able to release your game commercially — you'd get sued! But more important than just being different is the fact that a good story will give you ideas for game events that will make it more fun to play.

Let's say you wanted to design a game that has the same movement pattern as *Centipede* — a chain of linked segments moving back and forth across the screen, getting closer and closer to the player at the bottom. Naturally you can't just duplicate *Centipede* — there's no future in plagiarism. But what other story could explain that movement?

Your player could be a roller coaster repairman, and the moving segments could be a roller coaster. Of course, shooting roller coaster cars isn't exactly sociable behavior, so you may want to change the play action to having your player-figure patch holes in the roller coaster track or remove obstacles that are being put in the way by a terrorist. A roller coaster doesn't just get to the bottom of the screen and disappear, either — unlike *Centipede*, your game might have the roller coaster climb slowly to the top again, during which your player-figure has time to patch some holes.

You see how the game changes as you develop the story. And that can only improve your game. Even if an arcade game was your starting point, there's no reason why your game shouldn't be *better* than its inspiration. Look how much *Galaxians* improved on *Space Invaders*, only to have *Firebird* and *Galaga* improve even more.

Nowhere is this clearer than in *Tron*. What is the cone sequence except good old *Breakout*? Here, though, it has a story; with a character trying to get through the blocks, it takes on a whole new meaning. *Tron* also includes a traditional tank shoot-out, and the light cycles sequence is nothing but a variation on the old Atari VCS cartridge, *Surround*. Even the spiders aren't really new. But the story has changed and reshaped them, and the result is a game that *is* new.

Once you start thinking of new stories for games, you'll find it's hard to stop. You can fill up notebooks full of game ideas and plans in a very short time. And eventually there'll be one that you like so much that you can't let it go. You keep thinking about it, refining it, coming up with new variations. That's the one that you should program — the game you care about is the one that you'll have the patience and insight to program well.

Display: What Does the Player See?

On one level, all that goes on in a videogame is that a bunch of dots on the video screen are lighting up in various colors, and sounds are coming from a speaker.

But the dots, or pixels (short for "picture cells"), aren't just lighting up at random. They form patterns that the human brain recognizes as meaningful.

If the pixels form a yellow circle that has a wedge cut out of it, we recognize *Pac-Man*.

Mix different colored dots in set patterns, and we recognize the hardworking hero Mario from *Donkey Kong*, or the magnificent flying mounts from *Joust*.

Characters

On your VIC, a whole set of shapes already exists. Every character on the screen is a pattern of pixels that you recognize as letters or numbers or other symbols. None of them looks like a spaceship or a gorilla, but that's all right — you can design your own characters to replace some or all of the regular characters. When your program PRINTs the letter *A*, for instance, what appears on the screen will be the pattern you designed. Or several new characters can be put together to make part of a larger picture on the screen.

Keep Memory in Mind as You Design

Every picture your program displays will require one or more characters, and each character you redefine will use up memory, first when you actually create the character pattern, and again when you put that pattern on the screen. Since each character takes eight bytes of data to define it, and a whole character set contains 128 characters, you *could* use up 1024 bytes — a whole K — just in the data for a character set. In a 5K VIC, that makes a real difference in the amount of program space you have left.

So when you're planning the graphics display for your game, you need to remember that every different shape you show uses up memory. That's why it's a good idea to *repeat* shapes whenever you can. If you are going to have ten aliens on the screen, you can save a lot of memory by making them all look alike.

Animation

The pictures are just pictures until they move. It is when the wings of your steed start to flap that *Joust* comes alive. And half the fun of *Donkey Kong Jr.* is making Junior climb up or slide down the ropes and chains.

It is possible to make your animation as smooth and realistic as the best cartoon movie. But you also face the same problems that a cartoon animator faces. Each slight movement of a character requires a new picture of that character. Each direction a character faces, each action the character performs requires another drawing. And that takes memory.

So game designers compromise. They simplify the animation. The gorilla in *Donkey Kong* stamps his feet and grimaces — but that requires only two drawings per leg (stamping and not stamping), two drawings for the face (grimacing and not grimacing), and a single drawing for the body. When Kong is rolling barrels, there are new drawings to show him doing that. But since he rolls barrels to only one side, the programmer had to create the shape for only that side. And there's no attempt to make the gorilla walk realistically — he just bounces, without moving his legs. It's far from a classic Disney animated film, but it's good enough to make a great game.

Remember that anything on the playfield that changes during a game requires the same sort of memory considerations. If a trap door opens or a cup of milk empties, you'll need new characters for every new shape you show.

Player Movement

When players press a key or move a joystick, something needs to happen on the screen. And here is where the “feel” of a game comes in. Pushing on the joystick only causes a movement on the screen when your game program checks the joystick and makes changes according to what it finds there. If your program checks the joystick only once every second, the player-figure can’t move any more often than once every second. And that’s going to feel awfully sluggish to players.

As long as you’re programming in BASIC, you’ll want to design your program so that it checks the joystick and allows movement as often as possible. (That isn’t so important in machine language games, however — everything happens so fast that you often need to insert timer routines to slow down the action.)

So when you’re planning your program, you’ll probably build it around a central loop, a series of commands that repeat over and over. At the heart of that loop will be a joystick-reading routine. If the player isn’t pushing the stick, then you can skip on to other things. But if some action is requested, your program needs to be able to perform it as quickly as possible after the player asks for it.

This means that you want to have as few things happening on the main loop as possible. If the computer-controlled enemy-figures move as often as the player, then they will all have to be moved every time you go through that loop. And in BASIC, that will make your program crawl.

The solution is to compromise. Either have fewer computer-controlled opponents, have them move less often, or give them simple, regular actions. The back-and-forth movement of the aliens in *Space Invaders* is a good example of this. They don’t aim when they shoot; they don’t have to figure out a path when they move. They just do the same thing, over and over. Nothing could be simpler — or faster.

And as you design your program, move as much as possible off the main loop. Using IF/THEN GOSUB or ON/GOSUB statements, you can have the program decide during the main loop whether a particular routine is necessary. If it is, the program can jump to the subroutine and then come back; if it isn’t, the program can ignore it and go on, without wasting much time.

Eventually you may even start looking for books on machine language, so that your game programs can perform some of these functions much faster than BASIC allows. Still, don’t despair if

your first version of a program runs more slowly than you want. You'll soon learn tricks to streamline your program and make it go faster. As long as you don't expect the impossible, you'll be able to get your game to play smoothly and well.

Relationships on the Screen

The screen display may be dazzling, but the computer can't see it. It can't just glance and see whether a player-figure has bumped into a wall, or moved off the screen, or collided with an opponent.

The computer has to remember where everything is, and then check to see if there has been a collision. In our roller coaster variation on *Centipede*, for instance, the computer won't be able to see that the roller coaster is passing over a gap in its track. What it *will* do, though, is remember that there is a gap in the track at column X, row Y. That information can be stored in an array variable, GAP. GAP(3,0) gives the column number of the third gap; GAP(3,1) gives the row number. Each time the roller coaster moves, then, the program checks all the gap location variables, and if the roller coaster location is right above a gap location, the program can make the roller coaster fall through the gap.

You have to program that sort of thing into your game, so that the computer can do it over and over again. It sounds tedious, but the computer doesn't mind. In fact, that's one of the greatest things about videogames. Imagine playing *Joust* on a game board, having to calculate all the tiny movements and collisions yourself. The game would be slow — it *wouldn't* feel like you were flying. But the computer does all the drudgery of figuring out where you should be on the screen and whether you've bumped into anything, so that you can simply fly.

Intelligent Opponents

It's one thing to have the aliens in *Space Invaders* move mindlessly back and forth across the screen, or a centipede mindlessly dropping downward according to a set of rules. It's something else again to have the computer operate seemingly intelligent opponents that maneuver to try to get the better of you.

In a primitive form, like *Berserk*, it's a simple homing instinct — the robots tend to move up if your player-figure moves up.

In *Joust*, however, the programming is much more complex. The opponents have a general homing pattern — but they also have built-in strategy. If you're near, they'll maneuver to gain alti-

tude on you; they'll hover under the lip of a floating island; they'll fly upward and rebound down on you. The algorithms to control that kind of behavior are pretty complex, both as mathematics and as programming — it verges on artificial intelligence.

But don't worry. Most arcade games aren't nearly so complex. In *Donkey Kong*, for instance, the flames do have a tendency to home in on Mario, but they also follow certain basic patterns. When you learn them, you realize that the flames are only slightly more complicated than the robots in *Berserk*.

There are three levels of intelligence in computer-controlled figures: the mindless pattern, the homing pattern, and artificial intelligence.

Mindless Patterns

Here, the figures move in a set pattern, regardless of what the player does. Examples are the aliens in *Space Invaders* and the birds in *Donkey Kong Jr.* Once you see the pattern, you can plan on it and work around it. However, the computer usually makes up for its lack of subtlety by having lots of opponent-figures for you to cope with. One alien moving back and forth across the screen would be a picnic; a few dozen, and it's suddenly a challenge.

The Homing Pattern

In this system, the computer-controlled figures change their pattern according to where the player-figure is. The response is pretty simple, as in *Berserk* or *Pac-Man*. The monkeys in *Kangaroo* seem more intelligent, but they are pretty much the same. They move up and down the tree according to one of three set paths. They respond to the player's position by stopping on the same level as the kangaroo. Then they walk left a certain distance, depending on the left-to-right position of the kangaroo, and throw fruit. Programming this behavior pattern requires some careful planning and attention to detail, but no math more complicated than simple arithmetic.

Artificial Intelligence

With this kind of opponent, the computer *anticipates* what you will do and plans strategies accordingly. Sports simulations often require this, as the computer lines up a football team in a pattern that it thinks will cope with whatever you have planned. The complex knight movements in *Joust* require some anticipation. But games that rely heavily on artificial intelligence algorithms are

relatively rare. Most games rely on mindless or homing patterns, which are much easier to program and still make excellent game opponents.

Give the Poor Human a Break

One thing to keep in mind, though, is that even in BASIC the computer is usually faster than the human player's reflexes. In a homing pattern, the computer can always recognize, instantly, any change in the player-figure's movement and respond without any delay. A human being, however, takes a moment to recognize the change and respond to it. It's an easy matter to program an opponent to home in on the player and destroy him every single time. But that wouldn't be much fun to play.

So you need to build weaknesses into your computer-controlled characters. Make them slow or dumb — or both. They can get smarter as the game goes on, but no one enjoys playing a game of sudden death, in which there's no chance of survival at all.

Other Complications

Besides the computer-controlled opponents, you'll probably want to have other hazards. Most games have them.

In *Pac-Man*, for instance, the maze itself is a complication because players have to keep turning and dodging, and can get trapped in long corridors with ghosts at both ends.

In *Joust*, the lake of fire and the demon hand make the game a bit trickier, while the islands make it impossible to have many long, smooth flights. The eggs are one more thing to worry about while you try to battle the other knights.

Donkey Kong makes you cope with ladders and elevators and gaps between girders. *Kangaroo* makes you climb or leap from level to level. *Rally-X* not only has a maze, but also puts random rocks in the road and keeps you worrying about running out of gas.

The basic principle of complication is to make sure the player has to think of several things at once. Not only do you have to dodge the pickles, hot dogs, and fried eggs in *Burger Time*, but you also have to make hamburgers and Egg McMuffins. The more things going on at once, the busier the player has to be.

But, again, don't make it impossible. The complications should come at a rate a human being can cope with. It isn't fun to find out that a computer is quicker than you are. What's fun is feeling like you're a match for the computer, at least for a while.

Entrances and Exits

Figures appear or disappear often during a game. But a simple vanishing and reappearing act isn't very satisfying to the player. Besides, things are often happening so fast that players need a bigger effect to help them know what's going on.

For instance, when you eat a ghost in *Pac-Man*, it doesn't just vanish and then reappear somewhere else. We can see the eyes of the now-invisible ghost as it rushes from the site of its untimely demise to the box in the middle of the screen. There, the ghost gets back its shape — but it still wanders around inside the box for a moment or two before reemerging. This gives players some time without that ghost in the way, and helps them keep track of when it will come out again.

Even more important than when an opponent disappears and returns, however, is when the player's own player-figure is wiped out. It can range from the big explosion in *Asteroids* and other shoot-outs to Mario's head-over-heels tumble in *Donkey Kong*, but something needs to happen to let players know that they've been beaten, at least in this round.

However, as a matter of psychology, it helps if the player-figure's doom isn't *too* unpleasant. Abusive comments, for instance, don't make anybody feel like playing the game again. That's why so many arcade games have a cute ending — funny sounds and an animated sequence that doesn't suggest death or terrible pain.

Remember, if you've done well at creating a good player-figure and an enthralling game, players will be taking events in the game pretty personally. Your computer should be a good sport about winning, and not gloat.

Maintaining Interest

One of the elements in games like *Monopoly* or *Poker* or *Chess* or *Go* that has made them classics is that they remain a challenge, no matter how often you play. You may not come up with a *Monopoly* your first time, but you certainly don't want to create a game that people will play once and then discard. So what is it about a game that brings people back to play again and again?

Unpredictability

No game is completely unpredictable — the rules are designed, in fact, so that it will play pretty much the same every time. What

good would baseball be if you never knew what order to run the bases from one game to the next, or where the bases would be, or the size of the ball? Yet every baseball game is different because there are, within the framework of the rules, some things that can never be predicted. The speed and spin of the pitch, the force and direction of the bat, the angle and speed of the ball off the bat, the arrangement of runners on the bases and players in the outfield and the infield — these are never twice the same, even though the rules never vary.

You'll notice that all those variables depend on what human beings do. The trouble with computer games is that the computer is absolutely predictable — it will always obey your program commands. The computer has only two ways of being unpredictable. One is to generate a random number and use it in your program. The other is to let the player's input change the way the program acts.

Space Invaders has very little unpredictability. The aliens will tend to shoot more where the player is, and if you wipe out columns of aliens at the edges, it will take longer for them to reach the edge and drop down to the next level. That's about it.

Missile Command uses the random method of being unpredictable. While the enemy always aims at the same targets, the missiles never start at the same place and in the same pattern at the top of the screen.

Dig-Dug depends on the player for its unpredictability. The player, in effect, draws the playfield by carving a maze through the rock. Even though the placement of the creatures at the beginning of each level is always the same, and their movement starting times never vary, the levels can be as different as the player wants to make them, because everything they do depends on where the player has cut the maze and where the player happens to be.

Increasing Difficulty

Another way of keeping interest high is to make the game increasingly difficult. If you're a tennis player, you know that it's most fun to play with an opponent who's just a little better than you. And most arcade games use the same principle — no matter how good you are, at some point the computer is going to be just a little better.

How do you make a game harder from level to level? Speed, accuracy, and complication are the keys.

Speed. At low levels, opponents move slowly and are easy to catch or evade. Just by changing the timing, however, opponents

can be made to move faster, bit by bit, until they zip around the screen. Players can also be allowed to move faster, so they have to make decisions more quickly. And if opponents throw rocks or shoot bullets, those projectiles can move faster so they're harder to dodge.

Accuracy. At low levels, opponents can be programmed to miss. In *Asteroids*, for instance, the big enemy ship fires somewhere in the space around you — you usually have to practically try to get hit for it to harm you. But the small ship, which comes on later, fires much more accurately, predicting your future course so that it's much harder for you to dodge in time.

Complication. Keeping track of four enemy knights is hard enough at the beginning of *Joust*, when you're just starting out. But in later levels, you have dozens to worry about, and the eggs and pterodactyls, too.

Most games use a combination of these three, gradually increasing the speed, accuracy, and complications from level to level. It's easy to do if you design your programs with some key variables, so that you only have to change their values at the beginning of a level to have the game get faster and harder to play.

There's a problem, though, with having games get increasingly more difficult as you go from level to level. You have to get through a lot of boring stuff that's easy to do before you can get to the part of the game that's challenging.

The solution is to let players choose their starting levels at the beginning of the game. However, this might cause their total score to drop, and those who care about high scores will be frustrated. The designers of *Tempest* found one solution. If you decide to start at a higher level, you get a progressively larger bonus when you successfully complete that starting level. It's impossible for someone who starts at level 1 to get as high a score as someone who starts at level 11, even if both players end up at level 15, because the bonus for starting at a high level is so large. In the arcade, this also serves the purpose of encouraging players to play harder, shorter games, so the quarters flow faster, but even at home it's nice not to be penalized for starting out at a high level.

New Scenery

One thing that keeps players going with many games is the desire to find out what awaits them at the next level. The changes can be major ones, like the new screens introduced at each new level of *Donkey Kong*, or they can be minor, like the succession of bonus

vegetables in *Dig-Dug*. In *Galaga*, it's fun to see what new creatures will appear, and in what pattern they'll fly, in each bonus round. Later versions of *Pac-Man* vary the mazes; *Tempest* has new and increasingly complex geometric patterns at each level; *Venture* has new rooms, new treasures, and new monster guardians.

Even working in limited memory, it's possible to have small surprises hidden away for the player who makes it to higher and higher levels. It turns videogaming into exploration, and that adds a lot to the fun.

Incentives

Beating the machine and finding out what happens next are certainly incentives to keep playing longer, better, and more often. But you need to make sure that your game encourages players to accomplish the tasks the game sets out for them.

Negative incentives are usually the first things people think of. If you don't shoot the rockets out of the sky in *Missile Command*, your cities will be blown up and the game will end. If you don't punch out the monkeys and keep climbing in *Kangaroo*, you'll be decked by a piece of fruit or a gorilla.

There need to be positive incentives, too, to keep the player doing the right things. In *Monopoly*, for instance, you are encouraged to buy properties early in the game because, if you do, you get more rent later. You are encouraged to try for monopolies because your rent is doubled and you can make property improvements. You are discouraged from mortgaging property because you get no rent. That same kind of reward structure will help the player of your game to do the things your game requires.

Scoring

Scoring is the easiest way to let players know how they are doing. Each time players accomplish a goal, there should be a reward in points. It doesn't matter if the players know exactly how the points are awarded, but they should be given consistently, and harder accomplishments should be rewarded with more points.

In *Pac-Man*, for instance, you get points for each dot you eat. You also get points for each ghost you collide with after eating a power pill. And each ghost you collide with on the same power pill is worth progressively more points. The bonus for eating fruit and keys increases with each screen, too, rewarding you more for surviving farther and farther into the game.

But how many points? That's a tricky question, and it's really up to you. The general rule is that easy things get few points and hard things get lots of points. But no one accomplishment should get so many points that it completely overbalances the game.

Point inflation. Also, you should keep "point inflation" in mind. In *Asteroids*, you can't get less than ten points for anything. The score always ends with zero. You could get rid of that last zero, and it wouldn't affect anything, except that 1000 points would only be 100 points, and so on. Your score would be just as accurate.

So why does *Asteroids* multiply your "real" score by ten? Because it feels a lot better to get 10,000 points than to get 1,000 points. The game inflates your score so that you feel like you've accomplished more. After *Asteroids*, a game like *Super Breakout*, which rarely lets you get above 4,000 points, feels rather tame, in part because the score is lower.

But point inflation has limits. It's hard for players to visualize a billion points, so if scores get into the billions and trillions, and most of the score is meaningless zeros, players will tend to drop off the zeros anyway, and talk about getting 15 or 30 instead of 15,000,000,000 or 30,000,000,000.

Vanity board. Part of the fun of scoring is to see if you can get on the vanity board. Vanity boards began as a record of the high score. Gradually, games began including the top three or top five or top ten scores, and allowed the players to enter their initials or even their names. Players quickly learned to use these for graffiti, getting exactly the right scores so they could spell out messages of varying degrees of cleverness. However, the vanity board was erased when the machine was turned off. Now games often have methods of storing part or all of a vanity board even when the power is off. Most such games have split vanity boards. Half the high scores are permanent and do not disappear; the other half revert to zero when the game is off.

There are other variations, too. Some machines have default values when they power up, so that your score won't show on the vanity board until you get a minimum of, say, 10,000 points. Others come up with phony initials and plausible but low default scores, so that it looks like other players have left their initials and scores even when no one has played the machine that day.

It's often a good idea, if your game has scoring at all, to have a high score display at the end of a game, so that players can keep track of the top score earned since the program started running.

You can easily keep the top three or five scores, with initials, or the top score for each player. And it's not hard to include a permanent vanity board as part of the program, which is automatically saved each time the game ends.

However, vanity boards use up memory, and when things get tight in your game program, the vanity board is expendable; it just isn't a vital part of game play, the way scoring can be.

Bonus Turns

Games like *Pac-Man* and *Donkey Kong* never allow more than one bonus turn, but *Dig-Dug*, *Asteroids*, *Galaxians*, *Joust*, and others reward players with bonus turns at regular intervals. Limiting the number of bonus turns helps the arcade owners because it means people can't play as long on a single quarter — but people playing your game at home on their VIC don't need that restriction.

Of course, it can be carried to extremes, too. Endless games of *Asteroids* that end up with scores in the millions are the result of poor planning. The program should have been designed so that at the higher levels a dozen little enemy ships come on the screen at once. Then the game would have stayed challenging.

Story Rewards

There are rewards built into the story of the game. In *Donkey Kong*, Mario saves the girl; in *Donkey Kong Jr.*, Junior saves Papa. In *Kangaroo*, the mother saves her child. In these games, you'll notice that the objective is often achieved — the girl, Papa, and the baby kangaroo are saved several times during the game.

In games like *Defender*, *Missile Command*, and *Asteroids*, the win condition can never be achieved — there are always new waves of enemies, and never any time when the computer says, "Congratulations. You saved the earth." That makes these games more frustrating than the games that allow the main objective of the story to be achieved.

Sounds

Sound isn't just decoration. It isn't just music to attract you to the game when you hear it out on the street — though it does that very well.

Feedback

Have you ever played an arcade game with the sound turned off? Arcade owners sometimes get so tired of the sounds from popular

games that they turn the sound off entirely. When you play a silent machine for the first time, it throws your timing off completely. You rarely realize how much you depend on sound until it's gone.

Whenever players do something, your program should provide a sound that lets them know that the computer got the message. The walking and jumping sounds in *Donkey Kong* are a real help — sometimes it takes a moment to realize that Mario isn't doing what you told him, especially if you're glancing elsewhere on the screen at the time, and the sound can make a real difference in your timing.

News

Sounds also tell you about things happening elsewhere on the screen. An explosion, a collision, an alarm going off, a sound for having won a bonus round, a tune that tells you that the enemy is vulnerable or to warn you that a new enemy is on the screen — all these things help players keep track of what's going on while they're concentrating on the player-figure.

Mood

And don't underestimate the importance of music for simply setting the mood for the game. Background music is something chess never had. Videogame makers learned it from the movies.

Mysterious music can increase the suspense in a horror movie — and in a videogame. Busy, tense music makes a chase scene more exciting in an action film — and in a videogame. And bright, cheerful music can be part of the reward for success.

However, unless you know how to put music into interrupts — which can be done only in machine language — you'll have to keep in mind that every use of sound takes up memory and slows down the game. Most of the time, you'll probably use sound only where it's really necessary — to give information to the player.

Documentation

This is often one of the last things you'll add to a game program, but it's the *first* thing a player will see. By the time you finish programming, you know your game intimately. You know everything that will happen at every level, and you are probably already pretty good at beating the game. But other players won't know anything at all, and you need to tell them enough that they can have fun. If touching a balloon will blow them up, they

should know that; if they have to pick up a pot of gold in order to win, they should certainly be informed.

Your documentation, whether written on paper or on the screen, should tell the player several things:

1. *How to use the controllers.* What does the joystick do in this game? What does the fire button do? What will pushing the function keys do (if anything)? Players must know everything that will happen when they push keys or buttons or move sticks or paddles.

2. *Game options.* Beginning or advanced game? One or two players? Nighttime or daytime screen? Shields, hyperspace, flip-over, or no defense? All the possible options need to be explained.

3. *Game objectives.* What are players supposed to accomplish? How can they win? What gives them points? What gives them a bonus?

4. *Game hazards.* What is going to attack the players? What dangers can they run into? Anything that can cause players to lose a player-figure or lose the game should be mentioned. If players are trying to land a spaceship, they should know that touching the walls of the landing slip, landing too fast, and not having permission from Landing Control can all cause the ship to crash.

5. *Game rules.* What can and cannot be done? Computer games tend to be self-enforcing — if you try to do something illegal, the computer just won't do it. So the game rules documentation usually turns into a *tips* or *hints* section, where you let players know the way something works. If you have a football player trying to catch a pass, it helps to tell players what conditions have to be met for the pass to be caught.

Start Where You Are

Now you have your brilliant, fantastic videogame designed, and you discover that there isn't enough computer memory in the world, let alone in your VIC, to actually carry it out. Or you find out that some of the things you want to do would make the game run too slowly. Or you discover that some of the things you want to do are still out of your reach as a programmer.

That happens to all game designers, no matter how good they are. As you do the actual programming, you'll find out that some things in your plan just can't work the way you meant them to. That's fine. Every videogame you've ever played is the result of many compromises between the ideal game and the limitations of

2 Game Design

the machine and the programmer. Also, new ideas will occur to you while you're programming. The plan you make before the programming begins is to help you, not to limit you — to point out needs, not to eliminate alternatives. Once the creativity starts flowing, it can only help the game.



3

Setting Up Your Screen



3 Setting Up Your Screen

The video display in your game is very important, because that is where players have to “live” during the time they are playing your game. If it is comfortable, showing the players all the information they need to have, with attractive design and interesting graphics, they’ll be glad to come back to that world again and again. If it is cluttered and confusing and unattractive, the game will be hard to play and they aren’t likely to come back for more.

Changing Colors

So let’s start getting control of the screen. There are 128 possible combinations of playfield and border colors. Instead of describing them, I’ll show them to you. Just type in this simple program:

```
10 FOR X = 0 TO 255:POKE 36879,X:NEXT
```

What does this program do? Location 36879 controls the background and border colors. This program changes the number stored at location 36879 by POKing 256 different numbers there — all the numbers from 0 to 255 — so each of the possible combinations is shown twice.

But this isn’t very helpful. It all happens way too fast. So here’s a program that shows you the color combinations, but more slowly. And the program will tell you what number is being POKEd into 36879, so that if you like the colors, you can use them in your game.

Anything between braces — { } — is the name of a special VIC character. In line 10, you type the word PRINT, then a space, then a quotation mark. But then you don’t type { CLR }. Instead, you press the SHIFT key and the CLR/HOME key. In line 30, you don’t type { 2 DOWN } { 8 RIGHT }. Instead, you type the CURSOR DOWN key three times and the CURSOR RIGHT key eight times. See Appendix B, “How to Type In Programs” for a full explanation of all this.

```
10 PRINT "{CLR}"
20 FOR X=0 TO 255:POKE 36879,X
30 PRINT "{3 DOWN}{8 RIGHT}"X
40 FOR T=0 TO 1000:NEXT:NEXT
```

3 Setting Up Your Screen

Now the screen changes more slowly, so you can see what's happening. Also, line 30 shows you the number being POKEd into 36879.

Notice that when a value from 96 to 103 is POKEd in, the numbers on the screen turn into blue boxes. And from 104 to 111, the screen seems completely blank. The numbers are still there, but they're invisible because they're the same color as the rest of the screen. If you ever PRINT or POKE a character onto the screen, and it doesn't show up, the first thing to do is POKE a different value into 36879 — chances are the character is invisible because it's the same color as the screen.

Invisible objects. That's an important thing to remember — anything that is the same color as the background disappears on the screen. That means that you could put invisible objects on the screen, and then make them suddenly appear by changing their color or the background color. Magic — with a single POKE.

Here's a program to show you how that works:

```
10 PRINT "{CLR}"
20 POKE 36879,110+104*(PEEK(36879)=110):GOSUB 30:GOTO 20
30 FOR I=0 TO 19:PRINT CHR$(32){DOWN}{RIGHT}";:NEXT:RETURN
```

Line 20 acts like an on-off switch. It POKEs 36879 with 110 — unless 36879 already contains 110, in which case it POKEs 36879 with 6. But where does the 6 come from in that line?

On-off switch. It's a good thing to remember, with the VIC and many other computers, that *false* has a value of zero and *true* has a value of negative one. So the expression $(PEEK(36879) = 110)$ has a value of 0 if 36879 does not contain 110, and a value of -1 if 36879 *does* contain 110.

104 times 0 is 0 — so if 36879 holds any number other than 110, the program POKEs it with 110 + 0, or 110.

104 times -1, however, is -104. So if 36879 *does* contain 110, line 20 POKEs it with 110-104, or 6.

We could have done it another way, with two lines:

```
20 IF PEEK(36879) = 110 THEN POKE 36879,6:GOTO 30
25 POKE 36879, 110
```

Remember that in programming, there's usually more than one way to do the job.

Default color. What color does the screen usually display? Type RUN/STOP and RESTORE, to get the screen back to normal,

and then type PRINT PEEK(36879). The number 27 appears on the screen. That is the normal, or *default* value of 36879 — the number that the VIC puts into that memory location when you first turn it on.

Hide and Seek

Why did POKEing 36879 change the screen color? Can one POKE really do that much?

No. That POKE didn't actually change the colors. Color changes are complex operations, since the VIC has to change its entire signal to the television. That signal to the TV is handled by the Video Interface Chip, which does all the communication between the computer and the TV. The Video Interface Chip, or VIC, is so important that they named the computer for it.

The VIC chip updates the screen display 60 times a second. It does this automatically — you don't have to worry about it. The important thing is that the VIC chip looks at certain locations in memory in order to find out what the TV display should look like. 36879 is the location where the VIC looks — every 1/60 second — to find out what color the background and border should be. By putting a different number there, you are telling the VIC chip what colors you want — and the VIC chip takes care of the rest.

Screen Memory

That principle applies to everything about screen graphics. The VIC chip looks at a lot of different locations to find out what you want the screen to look like. By changing what is in those locations, you can get the VIC chip to display exactly what you want — as long as you follow the rules.

Besides checking 36879 to get the background and border colors, the VIC chip also scans an area called *screen memory* to find out which characters to display on the screen, and another area, called *color memory*, to find out what foreground and background color each of those characters should have, and still another area, the *character set*, to find out what the character actually looks like. It checks some other things, too, that we won't worry about right now. It does all this 60 times a second.

And by changing what the VIC finds when it scans through these areas of memory, you control what gets displayed on the television screen.

PEEKs and POKEs

If you're already familiar with the rules about using PEEK and POKE, you can skip this section.

POKE. This command puts a new number into a memory location. POKE is always followed by two numbers. The first number is the address that you want to change. The second number is the new value that you want to store there. There is always a comma between the two numbers:

POKE 36879,55

Legal POKEs. POKE always has to specify an address that actually exists in the computer. That means you can't POKE to a negative address number or to an address higher than 65535. Also, you can POKE into an address only a number from 0 to 255. Negative numbers or numbers above 255 will give you an error message and stop your program.

POKE uses only integers — whole numbers, with no fractions. However, using a fraction will not stop your program. POKE automatically chops off the fraction, both in the address and the number you are POKEing. If you told the computer

POKE 36879.5,55.3

the computer would act as if you had said

POKE 36879,55

What can you POKE? The numbers you POKE don't always have to be constants. They can be variables that contain the values you want:

POKE ADDR,NUM

POKE C(I),N(I)

In fact, you can even use *expressions*, like these:

POKE ADDR + X,INT((X*100)/256)

POKE SC + INT(I/256)*256,CHR\$(N)

Everything to the left of the comma is calculated to decide the address, and everything to the right of the comma decides what value to POKE.

PEEK. You use this function to find out what is stored at a certain location, without changing it at all. PEEK is not a command — it can't stand alone. You can't just say PEEK 55 — the computer won't know what to do. You have to have a command that tells the computer to do something with the number it finds when it PEEKs at a certain address. The address you are PEEKing

is always in parentheses right after the word PEEK:

X = PEEK(400)

You can use PEEK with many different commands:

PRINT PEEK(756)

IF PEEK(4096) = PEEK(500) THEN N = PEEK(9090)

FOR X = 0 TO PEEK(5999)

And PEEK works with expressions, too.

A = PEEK(INT(N/256)*256 + X + 3)

Whatever number results from all that calculating becomes the address where PEEK looks to retrieve a value.

PEEKing and POKEing in Screen Memory

POKE is the command you will use to directly change one spot on the screen. You will use the PEEK function to find out what is stored in a particular location in screen memory. Since the computer can't actually see what is on the TV screen, this is the only way your program can find out what is being displayed at any given spot.

Let's see how that works with a sample program. The value of ADDR will be 7680 (if you are using a VIC with 8K or more of expansion memory, use 4096). The variable X will be added to ADDR to select different locations on the screen:

```
10 ADDR=7680
```

If you have memory expansion, line 10 should be: 10
ADDR = 4096.

```
20 POKE 36879,90
```

```
30 FOR X=0 TO 505
```

```
40 POKE ADDR+X, INT(RND(X)*256):NEXT
```

```
50 FOR X=0 TO 505
```

```
60 POKE ADDR+X,32:NEXT:GOTO 30
```

The program is simple, and yet it has a pretty thorough effect on the screen, doesn't it?

Lines 30 and 40 put random characters on the screen. Lines 50 and 60 fill the screen with blank characters — the same character you get when you press the space bar.

You might notice that wherever there was writing on the screen before you typed RUN, the characters are blue, and inverse characters have a blue background. This is because we are doing nothing to change color memory. We'll get to that later.

Getting the Screen under Control

As you can see, changing the screen display is easy. The hard part is changing it exactly the way you want, to get exactly the effect you want — but we're getting there.

Codes. Notice that the screen doesn't display the *numbers* that resulted from the expression in line 30. Instead, it treats those numbers as codes for certain characters (letters, numbers, and symbols), and it prints the characters on the screen. These are not the same as the ASCII codes that you use with the CHR\$ function. You get very different results from these two statements:

```
PRINT CHR$(94)
POKE 7680,94
```

The CHR\$ function uses the ASCII code. Almost every computer understands the ASCII character codes — that's why BASIC uses them, so that your program can easily be transported to another computer. But your VIC's operating system has a harder time with them.

So you have two systems of code. But we'll go over character sets in more detail in the next chapter.

```
10 POKE 36879,155:ADDR=7680:S$="THIS IS A TEST"
20 PRINT "{CLR}"S$
30 FOR X=1 TO 14
40 A=PEEK(ADDR+X-1)
50 A$=MID$(S$,X,1)
60 PRINT A;TAB(5);A$;TAB(8);ASC(A$)
70 NEXT X
```

This example program puts a test message on the screen. Then it PEEKs at the first 14 addresses in screen memory, where that test message appears, and PRINTS the screen code number stored at that address in memory, then the character, and then the ASCII value of that character. By comparing the numbers, you can see how different they are.

But it isn't just a random difference. You'll notice that the screen code is always exactly 64 less than the ASCII code, except for the blank character, which is 32 in both lists.

Let's add some lines to the program:

```
80 FOR X=1 TO 14
90 A$=MID$(S$,X,1)
100 A=ASC(A$):IF A>32 THEN A=A-64
110 POKE ADDR+329+X,A
120 NEXT X
```

Line 100 takes the test message apart, one letter at a time, so that in line 110 the program can convert it to its ASCII numeric value (ASC(A\$)). Then, if the ASCII value is greater than 32, the program subtracts 64 to get the screen code. This formula doesn't work with all the characters, though — it's a bit more complex than that. The character code tables in Appendices G and H show the ASCII codes and screen codes for every character.

Organization of Screen Memory

Computer memory is one long chain of addresses, one right after the other. One section of this chain is used as screen memory. The 506 bytes from 7680 to 8185 in the unexpanded 5K VIC or with 3K expander, or from 4096 to 4601 in VICs with more memory added, are used as a map of the TV screen.

To convert that memory into 23 rows of 22 characters each, the VIC chip reads screen memory as if it were cut after every 22 addresses. At every twenty-third address, the VIC chip begins a new row on the screen. (See the Screen Location Table in Appendix C.)

In reading screen memory, the VIC chip starts in the upper-left-hand corner of the screen, moves across the top row to the right, and then jumps down to the leftmost character in the second row and moves across that row to the right. When it reaches the lower-right-hand corner of the screen, it jumps back up to the top-left corner and starts over.

The upper-left-hand corner is the *lowest* address of screen memory (7680 or 4096). The lower-right-hand corner is the *highest* address of screen memory (8185 or 4601). It may seem confusing that the top of the screen is "lower" in memory than the bottom of the screen, but that just means that the lowest-numbered address is at the top of the screen, and the highest-numbered address is at the bottom. As long as you remember that the VIC reads from left to right and from top to bottom, just as we do, and the memory address numbers follow the same order, it shouldn't be too confusing. Appendix C shows the address of each screen location.

Setting Up a Screen

Let's say we want to draw a cross (+) in the center of the screen. The middle of the screen is character 11 in row 11 (counting from 0). To find x , the exact address in memory, multiply the row

3 Setting Up Your Screen

number (11) by 22, the total number of memory addresses per row. The answer is 242. Then add 11, because we want the twelfth character (the first character is character 0). The answer is 253.

```
10 ADDR=7680:PRINT "{WHT}{CLR}"
20 POKE 36879,42
30 ROW=11:COLUMN=11:GOSUB 150
140 END
150 POKE ADDR+COLUMN+ROW*22,102:RETURN
```

If you type that in and RUN it, a small checkered square will appear about the middle of the screen. (RUN this program each time we add a new line to it, so you can see how each statement affects the screen display.)

To put a character just to the left of that one, subtract 1 from the value of COLUMN. Subtracting 1 always moves you to the left, except when you are already at the left margin.

```
40 COLUMN=COLUMN-1:GOSUB 150
```

Now, to add a character to the right, add 2 to the value COLUMN:

```
50 COLUMN=COLUMN+2:GOSUB 150
```

Moving up a row is almost as easy. Remember that each row is 22 characters long. So the spot directly above our original character on the screen is 22 addresses earlier in memory. However, in this example program, the subroutine at 150 already takes care of multiplying by 22. So to move up, we only have to subtract 1 from ROW. We also want to get COLUMN back to its starting value.

```
60 ROW=ROW-1:COLUMN=COLUMN-1:GOSUB 150
```

Now down a row:

```
70 ROW=ROW+2:GOSUB 150
```

Borders

Now let's put a border around the screen. First, we'll fill the first row:

```
80 ROW=0:FOR COLUMN=0 TO 21:GOSUB 150:NEXT COLUMN
```

Now the left-hand margin:

```
90 COLUMN=0:FOR ROW=0 TO 22:GOSUB 150:NEXT
```

The bottom row:

```
100 ROW=22:FOR COLUMN=0 TO 21:GOSUB 150:NEXT
```

And the right-hand margin:

```
110 COLUMN=21:FOR ROW=22 TO 0 STEP -1:GOSUB 150:NEXT
```

Now, to keep the READY message from spoiling the screen, we'll make it so the program never ends.

```
120 GOTO 30
```

You'll have to press RUN/STOP to end the program.

Random Screen Displays

The screen you just created will be the same every time you run that program. Now let's design a screen that will be different every time — stars and planets in space.

Program 3-1. Starfield

```
10 ADDR=7680:POKE 36879,8:PRINT "{BLK}{CLR}"
20 Q=80*RND(9)+20:Q1=6*RND(9)+2
30 FOR I=0 TO Q:X=505*RND(9):N=46:GOSUB 150:NEXT
40 FOR I=0 TO Q1:X=505*RND(9):N=81:GOSUB 150:NEXT
50 FOR I=0 TO 999:NEXT:PRINT "{CLR}":GOTO 20
150 POKE ADDR+X,N:RETURN
```

This program creates a starfield, with distant (small) and near (large) stars. It waits about a second and then draws another, completely different one.

There are three random elements: how many near stars, how many far stars, and where the stars are placed.

Line 20 is where the program decides how many far stars (Q) and near stars (Q1) there will be. In lines 30 and 40, these random values are used to decide how many stars of each size will be placed on the screen.

Lines 30 and 40 also generate a random number, X, which represents the address on the screen where a particular star will be placed. So the number of near stars, the number of far stars, and their placement on the screen are all random.

Controlling the random numbers. How do these random numbers work? The RND(*n*) function generates a random fraction between 0 and 1. The number in parentheses (the *argument*) doesn't affect the range of the numbers. Your program then multiplies the random fraction to get a random number in the range that you want.

Integers. The number that results will almost always be a fraction, so to get a whole number you need to use the INT function. A = INT(500*RND(5)) will generate a random number between 0 and 499. A = INT(5*RND(5)) will generate a random number between 0 and 4.

Minimums. You establish minimums by adding to the

3 Setting Up Your Screen

random number. $A = \text{INT}(5 * \text{RND}(5) + 3)$ will generate a random number between 3 and 7. A simple program like this will let you experiment with random numbers:

```
500 A=INT(5*RND(5)+3):PRINT A,:GOTO 500
```

Hold down the CTRL key to make it print more slowly. You can type RUN/STOP to stop the program at any time. You might want to change the 5 and 55 to any value you like and see what happens.

Random numbers are vital to most games, because they are the easiest way of making sure that the game never plays the same twice. But you need to make sure that you control the values resulting from the RND function by setting the right minimum and maximum.

Regular patterns. As a general rule, the more regular the pattern of your playfield, the simpler the program needed to generate it. A simple checkerboard requires only a few lines:

```
10 ADDR=7680:POKE 36879,238:PRINT "{CLR}":N=160
20 FOR X=0 TO 20 STEP 2:FOR Y=0 TO 22 STEP 2:GOSUB
  150:NEXT:NEXT
140 END
150 POKE ADDR+X+22*Y,N:POKE ADDR+X+1+(Y+1)*22,N:RE
  TURN
```

Variety. Just because you use a regular pattern in your playfield doesn't mean it has to be dull. You can make simple changes in a screen setup routine that will make a big difference on the screen. Add these three lines to the program you just typed:

```
30 READ N:IF N=0 THEN RESTORE:N=160
40 GOTO 20
200 DATA 102,81,78,86,77,127,255,79,80,0
```

Now the program reads a "table" of information in the DATA statement in line 200. Each new number causes a new character to be printed to the screen. You can experiment by adding new numbers to the table. Just make sure the only zero in the list comes at the very end of the list.

You'll notice that the program keeps cycling through the characters, even though each character is listed only once at line 200. The key to this is in line 30. The program READs the value of N from the table. Each time it READs one number, BASIC remembers where it left off in the table, and on the next READ it will bring back the next value in order. When it brings back a zero, the IF-THEN statement notices and the statement RESTORE is

executed. RESTORE simply puts the pointer back to the beginning of the list. Then the value of *N* is set to 160 again, and the whole loop starts over.

Also, many of the patterns completely conceal the fact that the screen is divided into little rectangles. Just because the VIC uses characters for its graphics doesn't mean you have to have square-looking screens.

And see how much variety comes from changing the screen color. Change the screen color POKE in line 10 to POKE 36879,8 or POKE 36879,233 and see how much difference it makes.

Seeing What's on the Screen

Now that you have a checkerboard of characters on the screen, how can you fill in the spaces between them? All you need is a line that PEEKs at every location on the screen, and wherever it finds a spot that meets certain conditions, it POKEs a character there.

Here's a variation on the checkerboard program that shows how PEEK can be used. Look at it carefully, because this technique will be vital later, for figuring out whether your player-figure has bumped into something on the screen.

Program 3-2. Fill In

```

10 ADDR=7680:POKE 36879,233:PRINT "{CLR}":N=160:P=
  81
20 FOR X=0 TO 20 STEP 2:FOR Y=0 TO 22 STEP 2:GOSUB
  150:NEXT:NEXT
30 FOR X=0 TO 505:IF PEEK(ADDR+X)<>N THEN POKE ADDR
  R+X,P
40 NEXT:READ N,P:IF N=0 THEN RESTORE:N=160
50 GOTO 20
150 POKE ADDR+X+22*Y,N:POKE ADDR+X+1+(Y+1)*22,N:RE
  TURN
200 DATA 86,78,77,127,255,95,105,79,80,88,74,113,1
  12,32,0,81

```

You'll notice that in line 30, the program scans the entire screen, from position 0 to position 505, but the *P* character is POKEd only if that spot on the screen does *not* contain the current *N* character.

Multiple READs. In line 40, the program READs two numbers, *N* and *P*. That means that in the DATA table, the numbers for *N* alternate with the numbers for *P*. If you alter the table, you need to make sure that your list comes out even, and that there is a zero in the second-from-last position.

3 Setting Up Your Screen

Combining characters. Also, you can see how the alternating characters combine to create shapes that never existed before. That's how you'll make larger pictures on the screen — combining several shapes to make one complete design. The graphics characters built into the VIC's character set — screen codes 64-127 and 192-255, and ASCII codes 96-127 and 160-255 — are designed so that many of them fit together exactly, so you can make continuous drawings.

Screen Displays with PRINT

Just because you can POKE to screen memory doesn't mean you always want to. Sometimes PRINT statements can be even more effective, since you can print entire strings all at once. Setting up the strings takes more programming — and more typing — but once strings are set up they can be PRINTed instantly anywhere on the screen. This program demonstrates how quickly PRINTing a string can make a change on the screen. Think of it as a wiring diagram, and imagine that you're controlling a spark trying to escape from the opposite corner of the screen.

Program 3-3. Wire

```
10 DIM A$(6),S$(8):POKE 36879,1:PRINT "{CLR}{WHT}"
   :GOSUB 500
20 PRINT "{HOME}{DOWN}":FOR I=0 TO 8:PRINT A$(0):N
   EXT I
30 PRINT "{HOME}":GOSUB 150:PRINT A$(N+3):PRINT S$
   (8)+A$(6-N)
40 PRINT "{HOME}{DOWN}"+S$(S)+A$(0):GOSUB 160:GOSU
   B 170:PRINT "{HOME}{DOWN}"+S$(S)+A$(N)
50 FOR I=0 TO 2000:NEXT:GOTO 30
150 N=INT(2*RND(9)):RETURN
160 N=INT(3*RND(9)):RETURN
170 S=INT(7*RND(9)+1):RETURN
500 FOR X=1 TO 20:A$(0)=A$(0)+CHR$(221):NEXT
510 FOR I=1 TO 5 STEP 4:FOR X=1 TO 10:A$(I)=A$(I)+
   CHR$(106)+CHR$(107):NEXT:NEXT
520 FOR I=2 TO 6 STEP 4:A$(I)=CHR$(221):FOR X=1 TO
   9
525 A$(I)=A$(I)+CHR$(106)+CHR$(107):NEXT:A$(I)=A$(
   I)+CHR$(221):NEXT
530 FOR I=0 TO 2:A$(I)=A$(I)+CHR$(32)+CHR$(32):NEX
   T
540 FOR X=1 TO 20:A$(0)=A$(0)+CHR$(221):NEXT
550 FOR I=1 TO 3 STEP 2:FOR X=1 TO 10:A$(I)=A$(I)+
   CHR$(117)+CHR$(105):NEXT:NEXT
```



```
560 FOR I=2 TO 4 STEP 2:A$(I)=A$(I)+CHR$(221):FOR
    X=1 TO 9
565 A$(I)=A$(I)+CHR$(117)+CHR$(105):NEXT: A$(I)=A$(
    I)+CHR$(221):NEXT
570 S$(0)="{ 2 DOWN}":FOR I=1 TO 8:S$(I)=S$(I-1)+"
    { 2 DOWN}":NEXT:RETURN
```

As the screen changes, the spark will be forced to follow different tracks. Right now, the changes are randomly generated, using the RND(n) function, but in a game you might allow the player to control the center interruption, moving it up or down and left or right with the joystick, to counter the randomly shifting top and bottom strips. You might want to save this program — we'll be adding the spark and the player controls later.

Moving the Cursor with PRINT

If you're using PRINT statements to create your screen display, you need to have some way of starting the PRINT exactly where you want it. You do this by including the HOME and cursor control characters in the strings you are printing. This is easy to do with the VIC. Once you have typed the quotation mark that starts a string, pressing the HOME key or the cursor control keys will cause their character to be included in the string, instead of causing the cursor to move. Then, when the string is PRINTed, the cursor will move just as if you were pressing the cursor control keys.

The easiest way to make sure the cursor always ends up in the right place is to PRINT the HOME character, either by making it the first character in a string or by PRINTing CHR\$(19). This will move the cursor to the upper-left-hand corner of the screen. Then you can enter as many CURSOR DOWN and CURSOR RIGHT commands as you need to get the cursor to the exact position where you want the next string to start PRINTing.

An alternative to using HOME and CURSOR RIGHT characters is to use the TAB function. TAB will move the cursor to an absolute column position. That means that no matter where you are on the line, PRINT TAB(10) will move the cursor to column 10 on that line, even if it means moving backward. In effect, TAB(n) finds the left edge of the line and counts n columns to the right to find the new cursor position. If the n in TAB(n) is a number greater than 22, TAB will move to the next line and keep counting. A combination of HOME and TAB(n) can locate the cursor anywhere on the screen.

3 Setting Up Your Screen

In contrast with TAB, the SPC(*n*) function starts counting from the current cursor position and moves *n* spaces to the right from there. This is a relative rather than an absolute cursor positioning command, and it is usually not as useful for game programming.

Color Memory

Color memory is a map of screen locations, just like screen memory, only instead of interpreting the numbers stored there as *characters*, the VIC chip interprets the numbers in color memory as *color codes*.

Color memory is a perfect shadow of screen memory. For every location in screen memory, there is a matching location in color memory that controls the color of whatever character is displayed on the screen. That means that the tenth byte of color memory controls the color of the tenth character in screen memory.

Because of this arrangement, you can individually control the color of every single character on the screen. By changing a particular character to the background color, you make that character vanish, turning into a blank space. Or you can turn a single character into eight distinctly different characters by giving them different colors.

Your computer already allows this with PRINT statements — all you have to do is enter a COLOR key as part of the string to be PRINTed. The COLOR keys, on the top row of your keyboard, number 1 to 8, can be entered into a string by pressing CTRL and the COLOR key at the same time. Every character PRINTed to the screen from then on will be that color, until another COLOR key is pressed.

It is usually better, though, to POKE colors directly into color memory. It's easy to remember the numbers to POKE: just look at your VIC keyboard, see which number is on the COLOR key you want, and *subtract 1*. 0 = BLK, 1 = WHT, 2 = RED, and so on.

PEEKing and POKEing Color Memory

Color memory is a peculiar area of memory, in that you can't POKE any number higher than 15 into it. The numbers from 0 to 15 use the lowest four bits of a byte; if you POKE a number higher than 15 into color memory, the upper four bits of that number are chopped off. So if you POKEd 255 into a location in color memory, only the number 15 would be stored there.

However, if you PEEK at locations in color memory, you can easily get numbers higher than 15. That's because the VIC is putting garbage in those upper four bits. Whenever you PEEK into color memory, you should AND the value you get with 15, to wipe out the garbage in the upper four bits:

COLOR = PEEK(LOCATION) AND 15

After this program line, the variable COLOR would contain the color code stored in that particular LOCATION in color memory.

Finding Screen and Color Memory

Relocatable programs. So far we've been using the numbers 7680 and 4096 as *absolute* locations for screen memory. But when you write a program, you want to be able to RUN it no matter where screen memory is. So instead of using the number 7680 or 4096, we'll write a program line that finds out where screen memory is.

Fortunately, your VIC always stores the address of the start of screen memory at location 648. It isn't the whole address, however — it's just the "high byte," or the first byte of the two-byte address in hexadecimal notation. If you don't know how to use hexadecimal notation, don't worry. A simple program line will convert it into a decimal number that you — and your program — can understand.

SC = PEEK(648)*256

From then on, the variable SC (for S**Screen**) will contain the address of the upper-left-hand corner of the screen.

The same system works for locating color memory, except that the address of color memory must be found using a more complicated formula:

CM = 37888 + 256*(PEEK(648)AND2)

Or, you can find out both addresses in the same program line:

SC = PEEK(648):CM = 37888 + 256*(SC AND 2):SC = SC*256

If you include this line at the beginning of your program, SC will always be the address of the upper-left-hand corner of screen memory, and CM will always be the address of the upper-left-hand corner of color memory.

Playing with Color Memory

Here's a short program that will show you how color memory affects the screen — and which color code produces each color:

900 SC=PEEK(648):CM=37888+256*(SC AND 2):SC=SC*256

3 Setting Up Your Screen

```
920 FOR I=0 TO 505:N=INT(RND(9)*8):POKE SC+I,N+48:
    POKE CM+I,N
930 NEXT:GOTO 920
```

Line 900 establishes the values of CM and SC. Line 920 first generates a random number in the range 0 to 7. It POKES the screen code for that number character (N + 48) into screen memory (SC + I), and then POKES the color code into the corresponding location in color memory (CM + I). The same variable, I, leads us to the corresponding locations in color and screen memory.

You'll notice that there are some blanks. These are the color code that is identical to the background color. Whatever character is being displayed there is invisible because there's no contrast between the background color and the character color.

To see this working with graphics characters, try adding these lines to "Fill In" (Program 3-2).

```
10 SC=PEEK(648):CM=37888+256*(SC AND 2):ADDR=SC*25
   6
15 POKE 36879,233:PRINT "{CLR}":N=160:P=81
25 GOSUB 300
45 GOSUB 300
300 Q=INT(RND(9)*8):IF Q=PEEK(36879)AND 7 THEN 300
310 IF Q=PEEK(CM)AND 7 THEN 300
320 FOR I=0 TO 505:POKE CM+I,Q:NEXT I:RETURN
```

Now the background color will change with each screen change. The IF statement in line 300 protects us against having characters the same color as the background. Line 310 protects us against having the character color the same twice in a row.

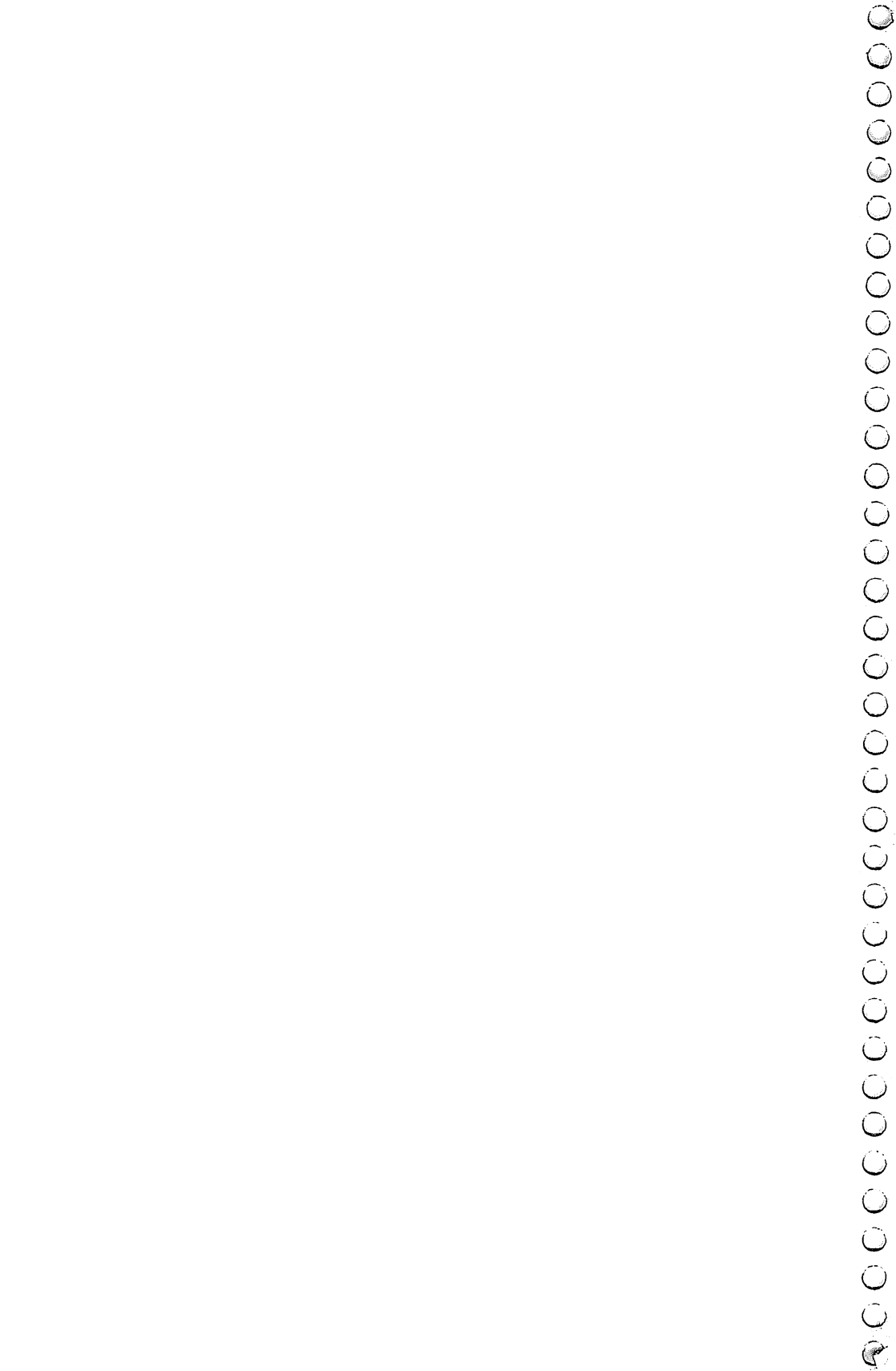
Or start again with the original version of Fill In, and add these lines:

```
10 SC=PEEK(648):CM=37888+256*(SC AND 2):ADDR=SC*25
   6
15 POKE 36879,233:PRINT "{CLR}":N=160:P=81:GOSUB 3
   00
300 FOR I=0 TO 505:Q=INT(RND(9)*8):POKE CM+I,Q:NEX
   T I:RETURN
```

Now the program assigns each location in color memory its own random color value. What was once a completely regular, symmetrical screen now looks completely unpredictable — and characters that used to fit together to make combined patterns now are clearly separate.

If any of the material in this chapter is still unclear to you, it

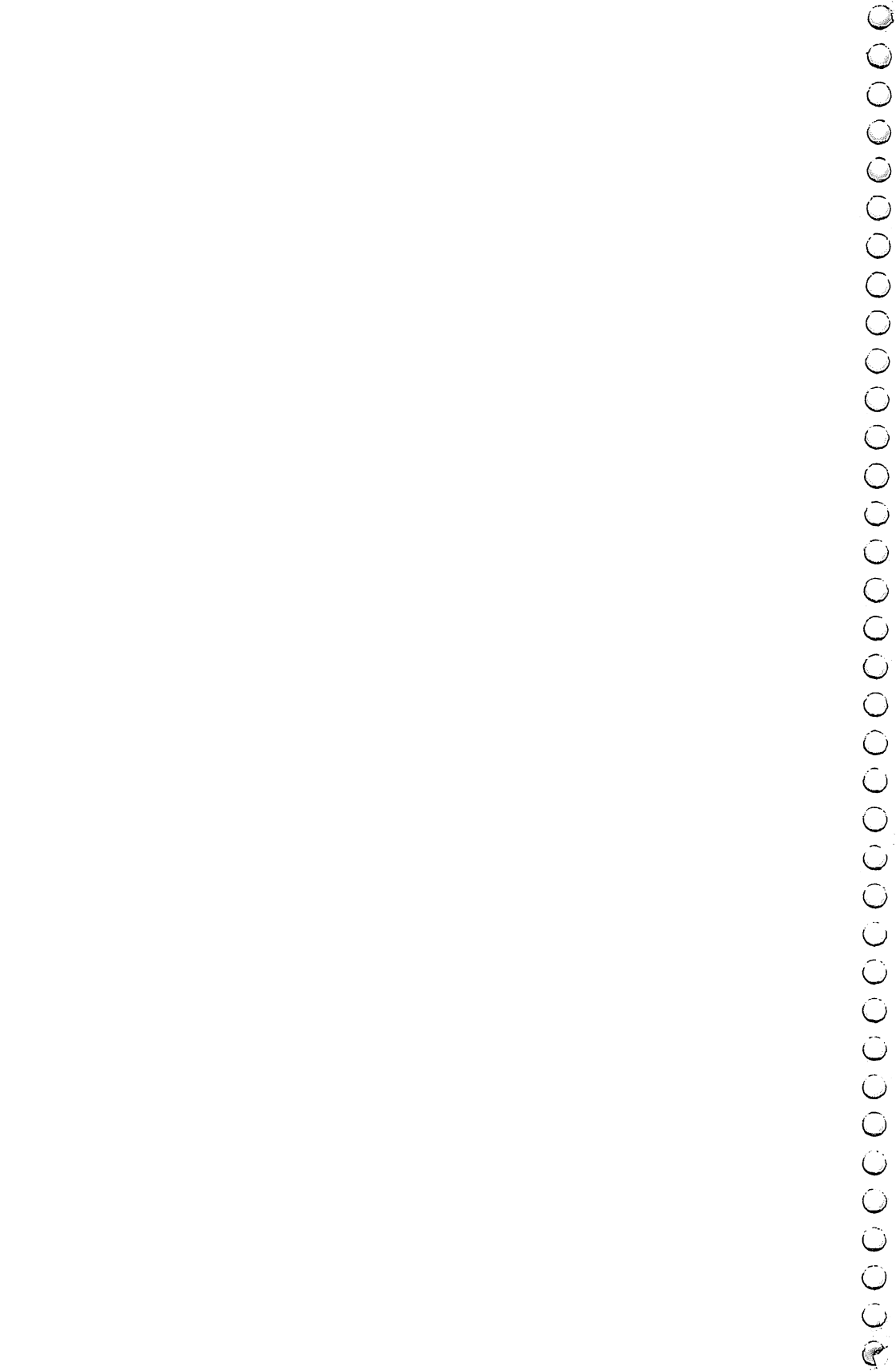
might be a good idea to go back and review it, studying the example programs to see how they work. Without a clear understanding of screen and color memory and how to use them, it will be harder to make use of the more advanced techniques presented in the rest of the book.





4

Custom Characters



4 Custom Characters

The VIC comes with a complete character set, consisting of all the numbers, letters, symbols, and graphics shapes that the VIC can produce. These standard VIC characters offer plenty of variety, and they can be combined to create many interesting shapes and patterns, as we've already seen in Chapter 3.

But there will be times when you need to draw a shape or a figure that the character set just can't produce. Perhaps you need a human figure, someone to live in the game world you've created, to experience the problems and dangers your game will present. The standard VIC graphics characters would produce a stick-figure at best, and you want something better.

That is when you throw out the VIC character set and create your own.

Creating a custom character set uses up more memory, leaving less room for your program, and it can take quite some time to design the characters and add them to the program. But when you have exactly the right figure or pattern that you want, you and the people who play your game will agree that it was worth the effort.

Graphing a Character

Once you've decided what kind of figures or patterns you want to draw, you need to translate those shapes into a form your computer can understand. And your computer understands nothing but numbers.

To see how shapes can become numbers, look at Figure 4-1. This is the character matrix, a grid eight dots wide and eight dots high. The matrix is not quite square because the dots are slightly wider than they are tall.

Each character on the screen is created in this matrix. Dots, or *pixels*, are either *on* (displaying the character color) or *off* (transparent, so the background color shows through). Figure 4-2 shows the pattern of *on* and *off* pixels that make up the @ character on the VIC. If you compare Figure 4-2 to the @ character on

4 Custom Characters

the TV screen, you'll probably be able to pick out the individual pixels, depending on how sharp your eyes, and the TV image, are.

Figure 4-1. The Character Matrix

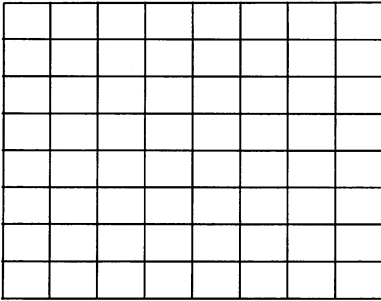
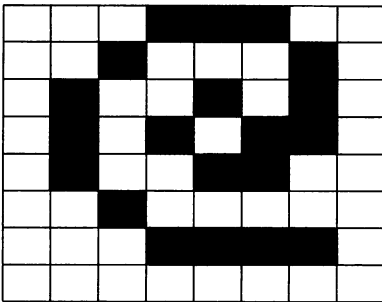


Figure 4-2. The Pattern of the @ Character



Each row of the character is eight pixels across. These are matched with the eight bits of a byte of memory. A 1 in the byte becomes an *on* pixel in the character. This means that to produce the pattern `_X_ _XX_ _` we need the binary byte 01001100, which is the decimal number 76. Each row of a character is stored in a single byte; eight bytes in memory contain the entire character pattern, in order, from the top row to the bottom row.

Translating Patterns to Numbers

The rightmost pixel has the lowest binary value, 1. If it is *on*, there will be a 1 in that position; if it is *off*, there will be a 0. Each pixel to the left has a value twice that of the one before. The second pixel from the right, then, has a value of 2: if there is a 1 in that posi-

tion, it adds 2 to the number; if there is a 0, it adds nothing. The third pixel has a value of 4; the next is 8; then 16, 32, 64, and 128.

Figure 4-3 charts these values for you, using the pattern of the character @. The top row of the character has three pixels *on*, forming the binary number 00011100. The values of these three bits, as you can see from Figure 4-3, are 16, 8, and 4. Add them together and you get 28. That number must be stored in character memory to create that row of pixels on the screen.

Figure 4-3. Translating @ to Numbers

		Bit Values								
		128	64	32	16	8	4	2	1	Total
Row	0				■	■	■			28
	1			■					■	34
	2		■			■				74
	3		■		■		■		■	86
	4		■			■		■		76
	5			■						32
	6				■	■	■	■		30
	7									0

The second row has only two pixels *on*, the third and seventh from the left, which form the binary number 00100010. The values of this are 32 and 2, making a total of 34 for the second row. Check the rest of Figure 4-2 to make sure the totals are correct.

You'll have noticed that the last row is completely empty and has a value of zero. The reason for this is that the VIC produces characters in the way outlined in Figure 4-3, but the 8 x 8 areas of characters on the screen actually touch the adjoining characters. This is what makes the screen a continuous field of dots. If you want your characters to make one continuous pattern, draw them right up to the edge of the matrix. If you want them to be clearly separate, however, be sure to include spaces in your character design, or one character will "bleed" into the next. If you leave space at the bottom of the character, and on the left and/or right sides, everything will work out fine. If *all* your characters have their left column blank, you can fill in the right column and there'll still be a separation. Note that this is what has been done with the @ character representation in Figure 4-3.

Graphing Your Own Character

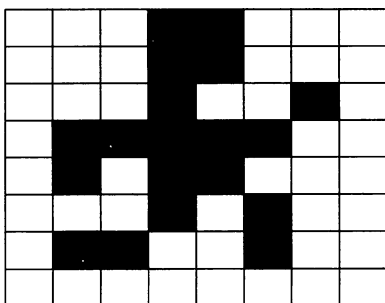
To draw your own characters, you can either divide graph paper into 8×8 squares, photocopy Figure 4-5, or simply draw your own 8×8 matrix. With this grid, create the image or images you'll need for your game by filling in the appropriate boxes.

If you need to draw something larger than one matrix — a large animal or space station, for instance — simply combine two or more matrixes. The VIC will treat them as two separate characters, but when you put them together on the screen, they'll look like a single drawing. However, the screen is not that large, and smaller characters will give the impression of more space in the playing area, especially when you want those characters to move about the screen.

You will probably need several custom characters for your game. Will the figures be throwing or shooting things at targets, or at each other? What will the projectiles look like? What will be the result when a figure is hit with one? All of these will need custom characters.

After you have drawn your custom character(s), you might have something similar to Figure 4-4.

Figure 4-4. Running Figure 1



This running figure has 19 pixels *on*, while the other 45 are *off*. To compute the value of each row of pixels, use the Character Work Grid provided by Figure 4-5.

Figure 4-5. Character Work Grid

		Bit Values								
		128	64	32	16	8	4	2	1	Total
Row	0									
	1									
	2									
	3									
	4									
	5									
	6									
	7									

Figure 4-6 shows the running character laid out on the work grid, with pixel values totaled on the right.

Figure 4-6. Running Figure 1 on the Work Grid

		Bit Values								
		132	64	32	16	8	4	2	1	Total
Row	0				■	■	■	■		24
	1				■	■	■	■		24
	2				■	■	■	■	■	18
	3	■	■	■	■	■	■	■	■	124
	4	■	■	■	■	■	■	■	■	88
	5				■	■	■	■		20
	6	■	■	■	■	■	■	■		100
	7									0

When you're finished, you should have eight numbers for each character. These numbers will be placed in DATA statements within your program so that the computer can READ them and put them in character memory.

The two figures we have drawn so far, for example, would have these DATA statements.

@ DATA 28,34,74,86,76,32,30,0
 Running Figure 1 DATA 24,24,18,124,88,20,100,0

Memory Locations

In Chapter 3, we worked with screen and color memory. These are areas of Random Access Memory, or RAM; we could not only PEEK to see what was stored in each location, but we could also POKE them to change what was stored there. The character set, however, is located in Read Only Memory, or ROM. You can PEEK there, but you can't POKE. The character set is permanently stored in your VIC.

The VIC's character memory begins at location 32768, with the top row of the @ character. The second row of that character is stored at location 32769, the third row at location 32770, and so on. A formula for finding the location of a particular row for a given character is:

Location Number = beginning of character memory (32768)
+ row + (8*screen code).

Note that the first row must be numbered 0 for this formula to work. The screen codes for each character can be found in the *Programmer's Reference Guide* or the *VIC-20 User's Guide*.

How can you create custom characters, if the character set cannot be changed? The memory location that instructs the VIC where to find the characters is in a RAM location. This can be changed to point to several sections of memory. In fact, changing where the VIC looks for character memory changes what is on the screen at the time. By making the character memory pointer point to the RAM of your VIC, you can begin to program your character set.

The location from which the VIC gets its character information is 36869. Its value is normally 240, or 242, but it can be changed. Although you can begin any custom character set in a RAM location ranging from 4096 to 7168, the best place to start is at 7168. To move the pointer to this RAM location, use the command POKE 36869,255. Now the computer will look at location 7168 for character data, instead of at the location in ROM, 32768.

Doing this, however, presents problems. First of all, creating your own character set by telling the VIC to get the data from RAM makes the standard character set unavailable. This may not be a problem if your game uses *only* custom characters. Then you can simply create the necessary figures and PRINT or POKE them into your game program. The following program does just that.

Program 4-1. Creating Characters

```

10 POKE 36869,255
20 POKE 52,29:POKE 56,29:CLR
30 FOR C=7168 TO 7183:READ D:POKE C,D:NEXT
40 DATA 24,24,18,124,88,20,100,0
50 DATA 24,24,18,62,88,20,98,0
60 PRINT "{CLR}A@A@A@"
70 GOTO 70

```

Program Explanation

Line Function

- 10 POKE 36869,255 tells the computer to go to location 7168 to get the data for characters, rather than looking in ROM location 32768.
- 20 Locations 52 and 56 tell the VIC to change the pointers to the top of the available RAM memory. The normal value in the unexpanded VIC is 30. A value of 29 takes 256 bytes of memory from BASIC, the smallest possible block. Ordinarily, this would be enough for a small custom character set.
- 30-50 Tell the VIC to replace the A and @ characters in the standard set with the new figures by READING the DATA statements.
- 60-70 PRINT the new custom characters on the screen and keep them there.

A program such as this produces *only* the custom figures you instruct it to. You do not have any of the letters, numbers, symbols, or graphic characters in the VIC's standard set. If you eliminate line 70, for instance, and press any key but the A or @, only garbage will show on the screen.

If you need any other characters from the VIC's standard set, you'll have to copy the letters, numbers, or symbols you intend to use into the new character memory in RAM. When you move your character set to location 7168, the topmost section of BASIC, you must also protect it from that program's actions. Fortunately, both problems can be corrected easily.

The following program copies the first 64 characters of the VIC to RAM location 7168 and then protects them from BASIC.

Program 4-2. Copying Characters

```

10 PRINT "{CLR}":POKE 36869,255
20 POKE 52,28:POKE 56,28: CLR

```

4 Custom Characters

```
30 FOR I=7168 TO 7679:POKE I,PEEK(I+25600):NEXT  
40 END
```

Program Explanation

Line Function

- 10 Again, POKE 36869,255 tells the computer to go to location 7168 to get the data for characters, rather than looking in ROM location 32768. (To send the computer back to 32768 for character data, just POKE 36869,240, its normal value.)
- 20 As in the first program example, locations 52 and 56 change the pointers to the top address of available RAM memory. A value of 28 takes 512 bytes of memory from BASIC, just enough for the data for the 64 characters you are moving to RAM. This line also protects the character memory from BASIC.
- 30 This line may look complicated, but actually it's not. The VIC is told to look at the data numbers for the first 64 characters in ROM (32768 to 33280), look at what's there (PEEK(I + 25600)), and then POKE those same numbers into locations beginning at 7168 and ending at 7679. In other words, the first 64 characters have been copied from their ROM locations to your new RAM location. The number 25600 is 32768 minus 7168. Another way of doing this would be:

```
30 FOR X = 0 to 511:POKE 7168 + X, PEEK(32768 + X): NEXT
```

After you run Program 4-2, you'll notice several things. First of all, the flashing cursor is lost. Second, only the first 64 characters are available. If you copied all 512 characters in the VIC, two kilobytes of RAM would be used up. Since you will rarely need more than a dozen custom characters, 64 is usually enough to give you custom characters *and* all the letters and numerals.

Try pressing random keys and see what happens. You'll see that the characters with screen codes from 0 to 63 (refer to the screen code tables in your reference guide) are available. These are the characters from the @ character to the ? character. If you press SHIFT and then another key, you'll often get garbage that means nothing to you. Continue to press keys and then sit back and watch. Some of the characters are changing before your eyes.

What is happening here is that the VIC thinks character memory is 2K, or 256 characters, even if you use only 64. When you press keys that call for characters with screen codes greater

than 63, the VIC reads their patterns from *screen* memory, so that the numbers stored there become patterns of the graphics characters. Your screen display is creating itself.

Now that you have the character set moved to RAM, available for altering, and protected from BASIC, you can begin to place your own custom characters in this set.

Replacing Characters

Glance through the screen code table in either your *Programmer's Reference Guide* or your *VIC-20 User's Guide* for a moment. Remembering that the first 64 characters are available, decide which of these you'll need for your game. How many of the letters will you need for screen titles and other game indicators? If you have the title "Lunar Lander" on the screen, as well as "Time" and "Fuel," you would not replace any of those letter characters with your new custom characters.

You should make up a data sheet, using the blank provided by Figure 4-7, to simplify the process of deciding which characters to save and which to change to custom characters.

In the left columns, list the first 64 characters, then their screen code values. You can find these values in the screen code tables in your reference guide. The starting location can be found by multiplying the screen code value by 8 and adding that to 7168, the starting location. This gives you the starting location of each character's eight bytes of data. The next column can be filled in with the name of the new character, such as "Running figure," which will replace the old character of that row. If you wanted to replace the @ sign with this new custom character, then write the name in the first row. Next to this, list the eight data numbers for the new character. In this example, the data numbers would read 24,24,18,124,88,20,100,0.

All this will take time, but it will be a valuable reference later in your game designing and programming.

Now you're ready to actually replace certain characters with your new custom characters. What you need to do is POKE your new numbers where the old values are. The data sheet will make this job easier, for it details the starting locations of each character, as well as the new values.

As stated earlier, the most convenient way to replace the old characters is with DATA and READ statements. You can put all the numbers into the DATA statements and then have the VIC READ

4
Custom
Characters

Figure 4-7. Character Data Sheet

Letter	Screen Code (POKE) Value	Starting Location	New Name	DATA Statement

them. The computer always READs the data in the order that it's listed, so make sure the numbers are in the proper order, and that there are eight numbers for each character. The DATA statements can be anywhere in the program, and for this reason many programmers put them at the end of the program, out of harm's way.

The standard format is:

FOR C(new character) =X to X +7:READ D:POKE C,D:NEXT

where X is the starting location of the new character from the data sheet.

For example, if you wanted to replace the @ character with the running figure, you would list it like this:

```
40 FOR C = 7168 to 7175:READ D:POKE C,D:NEXT
50 DATA 24,24,18,124,88,20,100,0
```

Add these lines to Program 4-2, replacing the END statement with this line 40. After running this program and typing an @, you should see the running figure printed on the screen. If you don't see the custom character, check the DATA statements for errors.

Using this same format, you can replace characters with your own set of custom characters. Make sure you insert your data numbers in the proper locations for all the replacements.

If your new characters are replacing old ones which are right after each other in the screen code table, you can place more than one in a READ statement. Figure 4-8 is a character data sheet for five characters which can replace sequential characters in the standard set.

Notice that two of the figures are split between two 8 x 8 squares. One is split between two squares horizontally, the other is split between two vertically. When both 8 x 8 squares are printed on the screen, the figure will appear complete.

To POKE all five figures, and READ all seven DATA statements that make up those figures, you could write:

Program 4-3. Characters in Sequence

```
40 FOR C=7432 TO 7487:READ D:POKE C,D:NEXT
50 DATA 24,24,18,124,88,20,100,0
60 DATA 1,1,1,3,6,1,3,0
70 DATA 128,128,32,224,128,64,32,0
80 DATA 0,56,56,16,124,16,124,68
90 DATA 56,56,16,124,16,124,68,68
100 DATA 0,0,0,56,56,84,124,16
110 DATA 56,40,40,0,0,0,0,0
```

4
Custom
Characters

Figure 4-8. Sequential Characters

Letter	Screen Code (POKE) Value	Starting Location	New Name	DATA Statement
!	33	7432	Running Figure #1	24,24,18,124,88,20,100,0
“	34	7440	Running Figure #2 Lft	1,1,1,3,6,1,3,0
#	35	7448	Running Figure #2 Rt	128,128,32,224,128,64,32,0
\$	36	7456	Squatting Figure	0,56,56,16,124,16,124,68
%	37	7464	Rising Figure	56,56,16,124,16,124,68,68
&	38	7472	Jumping Figure Top	0,0,0,56,56,84,124,16
'	39	7480	Jumping Figure Bot.	56,40,40,0,0,0,0,0

The standard characters from the ! (POKE 33) to the ' (POKE 39) were replaced with this one FOR-NEXT statement. Replace lines 40 and 50 in Program 4-2 and add the additional lines. After running this program, you should see the figures printed on the screen as you press the appropriate keys. Lines 100 and 110 include the values for the figure which would normally be two 8 x 8 squares high. However, when you press the keys for & and ' it will print a jumbled character, for the screen will show the two sections beside each other instead of one above the other. The next chapter on movement and animation will show you how to correctly display this figure on the screen.

If you choose to replace characters that are not next to each other in the screen code table, you will have to write separate lines for each character replaced. As you can see, replacing characters that *are* sequential will make it easier to write the program and will reduce the amount of memory used.

As you begin typing in the lines for your own custom characters, remember several things.

Each character needs eight numbers in the DATA statement to define it, even if one or more of the values are zero. Each character also needs eight bytes of memory to hold those numbers. If you don't have all eight bytes, the VIC will go on and read the next location, and the next, until it has an eight-byte pattern. Of course, this will not create your expected figures.

Be sure to list the DATA statements in the same sequence as the FOR commands to replace the characters. The VIC will read the first data numbers when it finds the first READ command, the second group of data numbers when it finds the second READ command, and so on.

When you are finished replacing characters, RUN the program to see if it works properly. If some of the custom characters turn out different from what you had expected, check the DATA statements. Are they the correct values for the pattern of pixels turned *on*? Were they typed in correctly? Also recheck the FOR C = X TO Y statements to make sure the memory locations are correct for the characters being replaced. Remember that if there is an error, it is in the program, not in the computer. By examining your program, you will always be able to solve the problem.

To find out how much memory is still available after building your custom characters, type in:

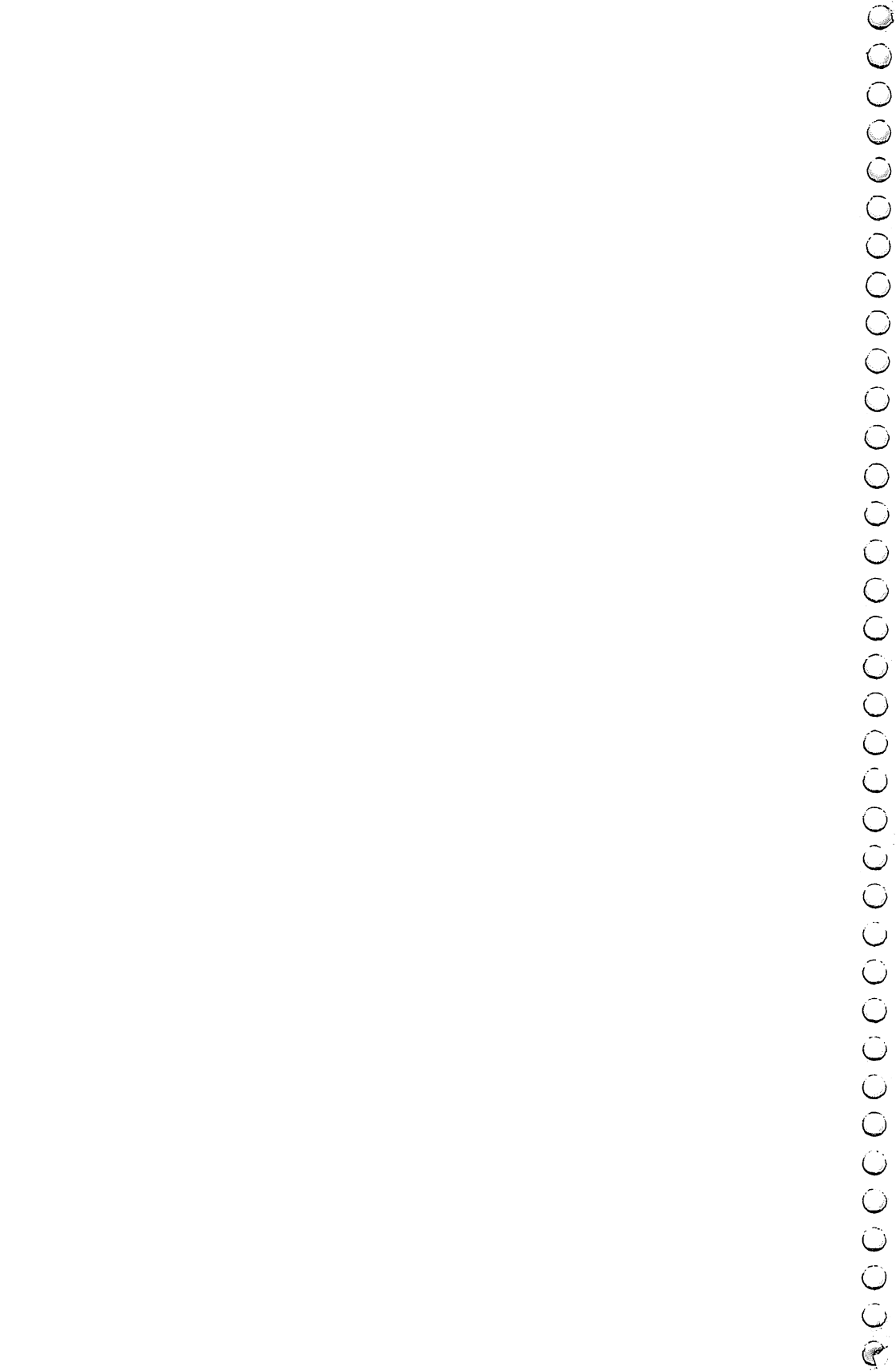
```
PRINT FRE(0)
```

The number displayed is the number of bytes available for the rest of your program.





5
**Getting Your
Figures
Moving**



5 Getting Your Figures Moving

Movement on the television screen is a series of still pictures. They seem to move because each still picture is only slightly changed from the one before, and each new picture comes on the screen so quickly that the eye can't see the jump from one to the next.

Speed

Movies change their image 24 times per second — any slower, and you begin to see flickering.

But the raster scan in your television refreshes the screen image 60 times per second. That's almost four times faster than the minimum for smooth animation. If you're programming in machine language, that leaves you plenty of room to perform all your calculations and still change the screen image often enough to have smooth movement. In fact, most machine language games have to insert delay loops or timing routines to slow them down so you can see what's happening on the screen.

That's why, in arcade games, no matter how fast the game goes, it can always go faster when you get to the next level. Even when the computer is controlling dozens of figures on the screen at the same time, it is probably marking time to stay slow enough for you to play it.

Unfortunately, BASIC calculations are slower (but a lot easier to program) than machine language calculations. It's sometimes hard to perform all the calculations you need fast enough for animation to be smooth. You need to make sure that your program performs as few calculations as possible between movements on the screen, or your game will run slower and slower and slower.

Movement and Animation

There are really two separate problems in programming movement on the screen. One is getting a figure to move from one

5 Getting Your Figures Moving

screen location to another. The other is making the figure's movements seem natural. Even though in many cases *movement* and *animation* are pretty much the same thing, in this book I'd like to use them to mean different things.

Movement is getting figures from one place to another on the screen.

Animation is making the player seem natural in its movements.

For instance, in *Donkey Kong*, *movement* is getting Mario from one place to another on the screen — up and down ladders, along ramps, jumping, and so on. *Animation* is the way Mario's legs move, the way he seems to face the direction he's moving in, the way he moves the hammer up and down — all the things that make his actions seem human.

Three Steps to Movement

Let's assume that the figure we want to move is already on the screen. We know he's at location 20 in screen memory. To make the figure move, we need to carry out three steps:

1. Calculate where the new location should be.
2. Erase the figure at the old location.
3. POKE or PRINT the figure at the new location.

Perform those steps over and over again, and you have movement.

Here's a program that will show you movement at its simplest:

Program 5-1. Movement Demonstration

```
10 POKE 36879,45:PRINT "{CLR}"
20 FOR I=7680 TO 7680+505
30 POKE I-1,32:POKE I,90
40 NEXT:POKE I-1,32:GOTO 20
```

Even in BASIC, this program is so fast you can hardly follow the movement, and your eyes probably fool you into seeing several characters in a row instead of just one. So let's add a delay loop to slow it down:

```
40 FOR X=0 TO 84:NEXT:NEXT:POKE I-1,32:GOTO 20
```

The movement is still jerky, but that's because we are moving from one character position to the next, with nothing in between. If you want smooth movement, all you have to do is create custom characters that, in combination, represent the half-moved figure. But that requires more memory and more logic to process the

movement, and by the time you execute it, things have slowed down enough that most of the gain in smoothness has been eliminated. In machine language, of course, the speed problem is eliminated, and smooth movement is relatively easy. For our purposes right now, however, the direct character movement is good enough.

Reading the Keyboard

Movement alone isn't enough, of course. The movement has to be controlled. And the player-figure must be controlled by the player. That means you must have some way of getting the player's instructions and making the figure on the screen respond to them.

Normal BASIC INPUT and GET statements won't work too well — they're too slow, for one thing. You'll want to get the player's instructions directly from the hardware — either the keyboard or a joystick. Right now let's use the keyboard.

Every time a key is pressed on the VIC, the operating system stores a number in location 197. This is the key code, and it has no relation to ASCII or screen code. Location 653 is similar, except it holds the code for the SHIFT, Commodore, and CTRL keys. Table 5-1 shows the key code for every key on the VIC that can be read from these locations.

Stop and go. The simplest player instruction of all is move or don't move. Using the movement program already in the computer, let's add a simple keyboard read line. If any key is pressed, movement will occur. If no key is pressed, movement will stop.

```
25 IF PEEK(653)=0 AND PEEK(197)=64 THEN 25
```

If no key is pressed, line 25 will loop forever; if any key is pressed, the program will go on from this line. (Pressing RUN/STOP will stop the program, as usual, even though it also generates a key code.)

You'll notice that pressing SHIFT and Commodore still shifts the character set, and turns the diamond shape into a Z. To keep the computer from switching character sets, POKE 657,128. You might add this command to the end of line 10. After your program is through, you'll probably want to POKE 657,0 to allow switching again.

Control matrix. When your program uses the keyboard for movement control, you usually want more than a simple stop-and-go instruction. You want separate keys for left, right, up, and down, and perhaps keys to perform other functions, the way the

5 Getting Your Figures Moving

joystick button is used in games to cause a figure to jump or shoot or disappear.

But you can't use just any keys. You want players to be able to put their fingers on the right keys and then forget about having to find them again, just like pressing buttons in the arcade. So you wouldn't want to use 1 for left, * for right, X for up, and f2 for down. There's no sense to that arrangement.

You might want to use the L, R, U, and D keys — but they're widely spaced, and only touch typists can remember which is which by reflex. You might want to use the cursor keys and have them perform their normal functions — but this means that keys change value depending on whether SHIFT is pressed or not, which can ruin the player's concentration in the middle of playing.

The most common solution is to set up a control matrix, an arrangement of keys in which the key that moves up is on top, down is on the bottom, left is to the left, and right is to the right:



Here is a single program line that will read the keyboard using the matrix on the right (@ is up, : is left, = is right, and / is down). After this line is executed, the variable LR will equal 1 for right and -1 for left, and the variable UD will equal 1 for down and -1 for up.

```
A = PEEK(197):LR = (A = 45) - (A = 46):UD = (A = 53) - (A = 30)
```

Whenever an expression like $A = 45$ is evaluated, it returns a value of either 0 or -1. *False* equals 0, and *true* equals -1. So if $A = 45$, then LR will equal -1, or true (-1) minus false (0). If $A = 46$, however, LR will equal 1, or false (0) minus true (-1). Remember that subtracting a negative number is the same as adding. If that sounds too much like math to you, just remember it this way:

Horizontal = (left?) - (right?) Vertical = (up?) - (down?)

Inside the parentheses you put the expression:

Keypress = Directionvalue

So the whole formula works like this:

$$\begin{aligned} & \text{Keypress} = \text{PEEK}(197) \\ \text{Horizontal} & = (\text{Keypress} = \text{Leftvalue}) - (\text{Keypress} = \text{Rightvalue}) \\ \text{Vertical} & = (\text{Keypress} = \text{Upvalue}) - (\text{Keypress} = \text{Downvalue}) \end{aligned}$$

Notice that if none of the four movement keys is pressed, the value of both LR and UD will be 0.

Allowing diagonals. However, there is a drawback to the control matrix. Your computer will store only one key code at a time at location 197. If two keys are pressed at the same time, the key with the higher code value is the one that will show up. So you can't press the key that means *up* and the key that means *right* at the same time and get a diagonal movement upward to the right. You only get *either* right *or* up.

In most games that's good enough. Diagonal movements don't matter much. Or if you assume most players will use joysticks, you can let the keyboard users have a slightly inferior game.

But there *is* a solution. Use SHIFT, Commodore, and/or CTRL for horizontal movement, and any two other keys for vertical movement (or vice versa). Since SHIFT, Commodore, and CTRL are read from location 653 and the rest of the keys are read from 197, your program can read vertical and horizontal movement separately, and get diagonals when two keys are pressed at once.

There is one drawback to this method. SHIFT, Commodore, and CTRL can be pressed in combination (see Table 5-1), and then a different value is stored at 653. But this is easy to cope with. Here's a line that reads location 653 and makes the variable LR equal 1 if SHIFT is pressed, -1 if Commodore is pressed, and 0 if both are pressed *or* if neither is pressed.

$$\text{LR} = \text{PEEK}(653) : \text{LR} = (\text{LR} = 1) - (\text{LR} = 2)$$

Or if you want the movement to be left if Commodore is pressed, regardless of whether SHIFT is pressed at the same time or not, use this line:

$$\text{LR} = \text{PEEK}(653) \text{AND } 3 : \text{LR} = (\text{LR} > 1) - (\text{LR} = 1)$$

Notice that this time we ANDed the value at 653 with 3; this was just to make sure that if the CTRL key was accidentally pressed, it wouldn't force a leftward movement, too, since the value at 653 will always be 4 or greater if CTRL is pressed.

It's just as easy to read the values at location 197. Let's read f5

Table 5-1. Key Codes

Values Stored at Location 197

Code	Key pressed	Code	Key pressed
0	1	33	Z
1	3	34	C
2	5	35	B
3	7	36	M
4	9	37	.
5	+	39	f1
6	£	41	S
7	DEL	42	F
8	{left arrow }	43	H
9	W	44	K
10	R	45	:
11	Y	46	=
12	I	47	f3
13	P	48	Q
14	*	49	E
15	RETURN	50	T
17	A	51	U
18	D	52	O
19	G	53	@
20	J	54	{up arrow }
21	L	55	f5
22	;	56	2
23	{cursor left-right }	57	4
24	RUN-STOP	58	6
26	X	59	8
27	V	60	0
28	N	61	-
29	/	62	CLR/HOME
30	/	63	f7
31	{cursor up-down }	64	{No key pressed }
32	{space bar }		

The following key codes cannot occur: 16, 25, 38, 40.

Values Stored at Location 653

Code	Key(s) pressed
0	(No key pressed)
1	SHIFT
2	Commodore
3	SHIFT and Commodore
4	CTRL
5	SHIFT and CTRL
6	Commodore and CTRL
7	SHIFT, Commodore, and CTRL

and f7 as up-and-down controls.

```
UD = PEEK(197):UD = (UD = 55) - (UD = 63)
```

Couldn't be easier. After this program line is executed, UD will equal 1 if the f7 key is pressed and -1 if the f5 key is pressed.

And since UD and LR are read from separate locations, a player can be moved diagonally — two directions at once — instead of horizontally alone or vertically alone.

Horizontal and Vertical Movement

Except for erasing the figure in its old location, you've already done everything that is involved in horizontal and vertical movements, back when we were setting up the screen. To move horizontally, you change the screen location by adding or subtracting 1. To move vertically, you add or subtract 22. Subtract to move up or left; add to move down or right.

The two key-reading routines we used give us values of 1 or -1. We can use this number directly for left-right movement. For vertical movement, however, we want to add or subtract a value of 22 or -22. So the vertical movement lines should be changed. Here's what it should look like in the diagonal-movement routine:

```
UD = PEEK(197):UD = 22*( UD = 55) - (UD = 63) )
```

If the value is negative, it will be -22 (22* -1); if positive, it will be 22 (22*1); and if an invalid key was pressed, the value will be 0 (22*0).

Staying on the screen. There's one other concern. We don't want the figure to move off the screen. There are two possible options: we can make the figure stop at the edge of the screen or keep moving and simply reappear at the opposite edge.

Both methods begin the same way. First, check to see if the player wants to move the figure. Then calculate the new address where the figure will be POKEd. If the address is beyond the edge of the screen, either don't move at all or change the new address so the figure will appear on the opposite side. Only then do you erase the figure at the old address and POKE it in at the new address.

A player-controlled movement routine. This simple program allows you to move a character around the screen using the @: = / matrix we already discussed. When the figure reaches the edge of the screen, it will stop:

Program 5-2. Movement with POKE

```
10 POKE 36879,47:PRINT "{CLR}":POKE 657,128
15 SC=PEEK(648):CM=37888+256*(SC AND, 2):SC=SC*256:
   OL=1:FG=90:POKE SC+OL,FG
20 A=PEEK(197):LR=(A=45)-(A=46):UD=22*((A=53)-(A=3
   0))
25 IF LR=0 AND UD=0 THEN 20
30 NL=OL+LR+UD:ON-(LR<>0)-2*(UD<>0)GOSUB 100,200
35 IF NL=OL THEN 20
40 POKE SC+OL,32:POKE SC+NL,FG:OL=NL:GOTO 20
100 EH=INT(NL/22):EH=NL-22*EH
110 IF (EH=0 AND LR=1) OR (EH=21 AND LR=-1)THEN NL
   =OL
120 RETURN
200 IF NL>505 OR NL<0 THEN NL=OL
210 RETURN
```

Here's how this program works:

Line	Function
10	Set the screen and border colors, clear the screen, and disable the character set shift.
15	Establish variable values: SC = starting address of screen memory; OL = old location (initialized at 1); FG = figure value — the screen code for the character to be moved. Then POKE FG into SC plus OL, to make the figure appear on the screen in the starting position.
20	Start the main movement loop. Set A to the key code currently in location 197. Set LR for the horizontal instruction, UD for the vertical instruction.
25	If both movement instructions (UD and LR) are 0, go back and look for a new key value.
30	Set NL to the new location for the figure (OL + LR + UD). Then GOSUB to one of the two edge-test subroutines, at 100 for the left/right edge test and 200 for the top/bottom edge test.
35	If NL has come back from the subroutines with the same value as OL, go back to the start of the movement loop without moving the figure.
40	Erase the figure at the old location (OL) and POKE it into the new location (NL). Then set OL to equal NL and go back to the start of the movement loop.
100	Left/right edge-test subroutine. EH is set to a number from 0 to 21, representing which horizontal position on the line the figure will be at if the movement is actually executed.

- 110 If the figure will be in the leftmost position ($EH = 0$) and got there by moving right ($LR = 1$), that means the figure *crossed* the right-hand edge of the screen, and the move won't be allowed. Likewise, if the figure will be in the rightmost position ($EH = 21$) and got there by moving left ($LR = -1$) that means the figure crossed the left-hand edge of the screen. The movement is cancelled in either case by setting NL to equal OL .
- 120 RETURN to line 35.
- 200-220 Top/bottom edge test subroutine. This test is easier. If the new location will be off the top of the screen ($NL < 0$) or off the bottom of the screen ($NL > 505$), then the movement is not allowed.

Wrapping around. With this next program, we'll change several things. First, instead of POKEing the figure on the screen, we'll PRINT it there. Second, instead of reading the control matrix, we'll read SHIFT and Commodore for left/right movement, and f5 and f7 for up/down movement.

Program 5-3. PRINT Movement Routine

```

5 DIM V$(22):V$(0)="{HOME}":FOR I=1 TO 22:V$(I)=V$(
  I-1)+"{DOWN}":NEXT
10 POKE 36879,42 :PRINT "{CLR}{BLK}":POKE 657,128
15 FG$="Z":NV=11:NH=11:GOTO 40
20 A=PEEK(197):B=PEEK(653)AND3
25 H=(B>1)-(B=1):V=(A=55)-(A=63):IF V=0 AND H=0 TH
  EN 20
30 NV=OV+V+22*((OV=21 AND V=1)-(OV=0 AND V=-1))
35 NH=OH+H+21*((OH=20 AND H=1)-(OH=0 AND H=-1))
40 PRINT V$(OV);TAB(OH);" ";V$(NV);TAB(NH);FG$
45 OH=NH:OV=NV:GOTO 20

```

Here's how this program works:

- | Line | Function |
|------|--|
| 5 | Set up the string array V(n)$, in which each value of V(n)$ PRINTs HOME and the same number of CURSOR-DOWN characters as the subscript. In other words, V(10)$ PRINTs HOME and ten CURSOR-DOWN characters; V(0)$ PRINTs HOME and no CURSOR-DOWN characters. |
| 10 | The same as line 10 in Program 5-2, POKE Movement. |
| 15 | Set the string $FG$$ to the character that will be the figure PRINTed on the screen. Set NV to the row and NH to |

5 Getting Your Figures Moving

- the column where the figure should first appear. Then skip the next few lines and execute the PRINT instructions at line 40.
- 20 The beginning of the movement loop. Set *A* to the value of the key pressed; set *B* to the value from SHIFT, Commodore, and CONTROL.
- 25 Set *H* to -1 (left) if Commodore (or SHIFT and Commodore) was pressed ($B > 1$), or 1 (right) if SHIFT was pressed ($B = 1$). Set *V* to -1 (up) if f5 was pressed, or to 1 (down) if f7 was pressed. If SHIFT and Commodore are not pressed, then set *H* to 0; if f5 and f7 are not pressed, then set *V* to 0.
- 30 Set *NV* (New Vertical position) to *OV* (Old Vertical position) plus *V* (Vertical movement). If *OV* is set at the last valid row, 21, and *V* calls for a movement down (+1), then subtract 22; if *OV* is at 0 and *V* calls for a movement up (-1), then add 22.
- 35 Set *NH* (New Horizontal position) to *OH* (Old Horizontal position) plus *H* (Horizontal movement). If *OH* is set at the last valid column, 20, and *H* calls for a movement right (+1), then subtract 21; if *OH* is set at 0 and *H* calls for a movement left (-1), then add 21.
- 40 Skip down the number of rows from HOME to the row where the figure is currently located (PRINT V\$(OV)), move right the number of columns to the current figure location (TAB(OH)), and erase the figure by printing a blank (" "). Then repeat the process, only using the new location values (*NV* and *NH*) so you can PRINT the figure (FG\$) at the new position.
- 45 Set *OH* and *OV* to the current figure position (*NH* and *NV*), and go back to 20 to get new instructions from the player.

Dangers of using PRINT. Using PRINT in games is very effective, since it works much faster than POKE, especially when you're moving large, multicharacter figures. However, the PRINT statement can have strange effects on the screen. As far as the computer knows, you want it to format text just the way it usually does. That text formatting can really get in your way.

If a character is PRINTed in the last column of the line, the lines below it on the screen are all moved down by one row to make room for the next character. That can completely destroy your screen display, unless you're working on a blank screen.

And if a character is PRINTed in the last column of the last row, or if a CURSOR-DOWN is PRINTed while the cursor is on the last row, the whole screen will scroll — and there goes the top row (or rows) of your screen display.

That's why this program treats the screen as if it were only 21 characters wide (column 0 to column 20) and 22 lines high (row 0 to row 21). If you keep the playfield and the border the same color, the player probably won't notice that the playfield is one column narrower and one row shorter than normal.

In-line logic. If you aren't an experienced programmer, the logic in lines 30 and 35 may look complicated to you. Since this kind of statement saves you from having a lot of IF statements and extra lines, it might be worthwhile to look more closely at what is going on. Let's examine line 35:

```
35 NH = OH + H + 21*( (OH = 20 AND H = 1) - (OH = 0 AND  
H = -1) )
```

The first part is pretty clear-cut. We are setting NH to equal OH, the old horizontal position, plus *H*, the player's horizontal movement instruction. *H* might be zero, of course, in which case this whole line will do nothing but add zero to OH, so that NH and OH are the same.

The rest of the line causes the wraparound. Look at the last 17 characters of the line: $-(OH = 0 \text{ AND } H = -1)$. If the figure was already in column 0 ($OH = 0$) and the movement called for is *left* ($H = -1$), then both conditions are *true* and the value within the parentheses is -1 . But we are *subtracting* that value, so that, in effect, if the result is *true* we will add 1. Now look back at the rest of that expression.

If the second half is true, then the first half ($OH = 20 \text{ AND } H = 1$) must be false. It will then equal 0. So the whole expression within parentheses evaluates as $0 - (-1)$, or $0 + 1$.

That number is multiplied by 21, with a result of 21, which is added to $OH + H$. Since we know that OH is zero and *H* is -1 , that means that NH is set to $0 - 1 + 21$, or 20. Now the PRINT instruction in line 40 will TAB to column 20, the last legal column.

What the player sees is that he or she instructed the figure to move left, and the figure jumped from the leftmost column to the rightmost position on the screen. That's wraparound.

Now you can see that if the first condition is true, the second will be false, and we'll be setting NH to the value $20 + 1 - 21$, or 0, and the figure will jump from the rightmost position to the leftmost position.

5 Getting Your Figures Moving

And if neither condition is true, then $21*(0 - 0)$ gives us 0, and NH will simply equal OH + H — the normal movement will take place.

PRINT for speed. PRINT is the fastest way to get characters on and off the screen in BASIC, provided you link all the PRINT instructions so they follow a single PRINT statement. The speed difference between Program 5-2 and Program 5-3 isn't so obvious, because we're putting only a single character on the screen. But what happens when you have a multicharacter figure, one that consists of two or more characters next to each other?

Here's a program that moves a four-character figure around the screen. If you have already entered and saved Program 5-3, all you need to do is change lines 15, 30, 35, and 40 so they are the same as in this program.

Program 5-4. Multicharacter PRINT Movement

```
5 DIM V$(22):V$(0)="{HOME}":FOR I=1 TO 22:V$(I)=V$(I-1)+"{DOWN}":NEXT I
10 POKE 36879,42:PRINT"{CLR}{BLK}":POKE 657,128
15 FG$="{&+&}{&-&}{2 LEFT}{DOWN}{&ε&}{&-&}{2 LEFT}{DOWN}{&ε&}{&-&}{2 LEFT}{DOWN}{&-&}{&+&":NV=11:NH=11:GOTO 40
20 A=PEEK(197):B=PEEK(653)AND3
25 H=(B>1)-(B=1):V=(A=55)-(A=63):IF V=0 AND H=0 THEN 20
30 NV=OV+V+19*((OV=18 AND V=1)-(OV=0 AND V=-1))
35 NH=OH+H+20*((OH=19 AND H=1)-(OH=0 AND H=-1))
40 PRINT V$(OV);TAB(OH);"{2 SPACES}{2 LEFT}{DOWN}{2 SPACES}{2 LEFT}{DOWN}{2 SPACES}";V$(NV);TAB(NH);FG$
45 OH=NH:OV=NV:GOTO 20
```

Now, instead of FG\$ being a single character, it is eight visible characters plus nine cursor-movement instructions. Yet all 17 characters are PRINTed so quickly that they seem to appear on the screen all at once. POKE statements in a FOR-NEXT loop appear on the screen much more slowly, so that you would see each piece of the eight-character figure appear separately, and it would move like an inchworm, a part at a time.

For a direct, head-to-head comparison of POKE and PRINT in screen movement, here's a program that creates a very large figure and gives you the option of moving it either with POKES or PRINTs. For simplicity, Program 5-5 allows the ship to move only left to right.

Program 5-5. Laser X Jet

```

10 PRINT"{CLR}{4 DOWN}{5 RIGHT}{RVS}LASER X JET
{OFF}"
20 PRINT"{5 DOWN}GIVE US A FEW SECONDS.
30 CLR:S=32
40 FOR I=7168TO7679:POKEI,PEEK(I+25600):NEXT
50 FOR I=7176 TO 7311:READ X:POKE I,X:NEXT
230 DATA0,0,0,0,0,0,63,2
240 DATA0,0,0,0,0,0,255,0
250 DATA0,0,0,0,0,0,252,32
260 DATA4,9,10,19,20,36,64,127
270 DATA0,17,153,149,83,81,0,0
280 DATA16,72,68,66,65,64,0,0
290 DATA0,0,0,0,0,128,64,32
300 DATA195,192,192,223,216,216,216,255
310 DATA255,0,0,255,160,64,160,255
320 DATA240,255,0,255,160,64,191,240
330 DATA0,254,1,252,198,195,255,0
340 DATA0,0,255,15,1,3,252,0
350 DATA0,0,192,248,255,255,0,0
360 DATA0,0,0,0,0,252,0,0
370 DATA224,24,198,192,192,198,24,224
380 DATA3,28,224,7,7,224,28,3
390 DATA0,0,3,60,252,3,0,0
400 DATA 7900,7901,7902,7903,7881,7859,7860,7882,7
904,7905,7883,7861,7862,7884,7906
410 DATA 7885,7907,7908,7909,7910,7911,32,17,16,15
,32,32,1,4,8,9,5,2,3,6,9,7,10,11,12
420 DATA 13,14
430 L$=" ABC{DOWN}{4 LEFT} DEFG{DOWN}{4 LEFT}HIIJK
LMN":FL$=" QPO":D$="{HOME}{8 DOWN}"
500 DIML(20):DIMY(20):DIMP(29)
510 FORX=0TO20:READL(X):NEXT?:FORX=0TO20:READY(X):N
EXT
520 PRINT "{CLR}PRESS {RVS}F7{OFF} TO MOVE WITH PO
KES"
530 PRINT "PRESS {RVS}F5{OFF} TO MOVE WITH PRINT"
540 POKE 36879,8
550 A=PEEK(197):ON-(A=63)-2*(A=55) GOTO 700,800:GO
TO 550
560 A=PEEK(197):ON-(A=63)-2*(A=55) GOTO 700,800:GO
TO 560
700 POKE 36869,255:PRINT "{CLR}":X=L(0)-7900:FOR I
=0 TO 20:L(I)=L(I)-X:NEXT
710 FORX=4TO20:POKEL(X),Y(X):NEXT
720 A=PEEK(197):IF A=63 THEN GOSUB 750:GOTO 710
730 IF A=55 THEN 800
740 FOR I=1 TO 3:POKE L(I),32:NEXT:GOTO 710
750 IF L(20)=8184 THEN RETURN

```

5 Getting Your Figures Moving

```
760 FOR X=0 TO 20:L(X)=L(X)+1:NEXT:FOR X=0 TO 3:PO
  KE L(X),Y(X):NEXT:RETURN
800 POKE 36869,255:PRINT "{CLR}":X=3
810 PRINT D$TAB(X)L$:A=PEEK(197):IF A=63 THEN 700
820 IF A<>55 THEN PRINT D$TAB(X+41)"{4 SPACES}":GO
  TO 810
830 X=X+1:IF X=80 THEN X=3:PRINT "{CLR}"
840 PRINT D$TAB(X+41)FL$:GOTO 810
```

This program uses another technique — the flame from the engines is displayed only when the jet is moving. This is done by treating the flame as a separate figure, which is moved along in tandem with the jet, but is erased when the jet stops.

Reading the Joystick

The joystick is read by PEEKing two locations. At location 37137, you read the up, down, and left positions of the joystick and the joystick button. At location 37152, you read the right position.

If the joystick is pushed in a particular direction, a certain bit will be set to 0; if the joystick is not being pushed in that direction, that bit will be set to 1. Here are the tests for each direction:

1. If the joystick is pushed left, PEEK(37137)AND16 = 0.
2. If the joystick is pushed down, PEEK(37137)AND8 = 0.
3. If the joystick is pushed up, PEEK(37137)AND4 = 0.
4. If the joystick button is pressed, PEEK(37137)AND32 = 0.
5. If the joystick is pushed right, PEEK(37152)AND128 = 0.

(Remember that the first four tests use location 37137, while the last uses location 37152.)

If any of these expressions is true, then the joystick is being pushed in that direction. Remember, too, that the joystick can be pushed to a diagonal position — it is possible for both 1 and 2 to be true, or 1 and 3, and so on. And the joystick button can be pressed at the same time.

Before you can read the joystick at these locations, however, you have to first prepare the VIC to read it. To prepare to read 37137 (up, down, left, button), you must POKE a 0 into location 37139. To prepare to read 37152 (right), you must POKE 127 into location 37154.

Unfortunately, location 37154 also has an effect on the way the keyboard is read, and if you have POKEd that location to allow you to read the joystick, it will make your computer ignore certain keypresses afterward. Pressing RUN/STOP-RESTORE will correct the problem, but it also stops your program. The best solution is

to POKE 37154,127 right before you PEEK(37152), and immediately afterward set things back to normal with POKE 37154,255.

Program 5-6 is an adaptation of Program 5-3, which reads the joystick instead of the keyboard.

Program 5-6. Joystick Movement

```

5 DIM V$(22):V$(0)="{HOME}":FOR I=1 TO 22:V$(I)=V$
  (I-1)+"{DOWN}":NEXT
10 POKE 36879,42 :PRINT"{CLR}{BLK}":POKE 657,128:F
  G$="Q"
15 POKE 37139,0:DD=37154:S1=37137:S2=37152:GOTO 45
20 POKE DD,127:B=(PEEK(S2)AND128)=0:POKE DD,255:A=
  PEEK(S1)
25 H=((A AND16)=0)-B:V=((A AND4)=0)-((A AND8)=0)
30 IF H=0 AND V=0 THEN 20
35 NV=OV+V+22*((OV=21 AND V=1)-(OV=0 AND V=-1))
40 NH=OH+H+21*((OH=20 AND H=1)-(OH=0 AND H=-1))
45 PRINT V$(OV);TAB(OH);" ";V$(NV);TAB(NH);FG$
50 OH=NH:OV=NV:GOTO 20

```

In line 15, the program POKes 37139 with 0 to allow us to read up, left, down, and the button. But instead of setting 37154 to 127 right here, the variable DD is set to 37154.

Then when the program actually reads the joystick in line 20, POKE DD,127 enables us to read 37154 (joystick right); and immediately after we read it, POKE DD,255 sets it back to normal again.

Line 25 sets *H* and *V* to -1, or 0, depending on which way the joystick is being pushed. The rest of the program is identical to Program 5-3.

The joystick button. As long as we're reading the joystick, let's find a use for the button. By changing lines 10 and 45 in Program 5-6 and adding line 12, we can make the figure invisible when the joystick button is pressed, and then make it reappear when the button is released.

```

10 POKE 36879,42:PRINT"{CLR}":POKE 657,128
12 DIM FG$(1):FG$(0)=" ":FG$(1)="Q"
45 PRINT V$(OV);TAB(OH);" ";V$(NV);TAB(NH);FG$(-(
  A AND 32)=32))

```

The subscript of FG\$(*n*) at the end of line 45 is equal to 1 if the joystick is not pressed and 0 if the joystick is pressed. You can make the figure appear only when the joystick is pressed by changing line 12 so that FG\$(0) = "Q" and FG\$(1) = "".

Animation

Animation — making figures move smoothly and naturally — is tedious to program and hard to do in BASIC. However, it can make a great difference in the way your game affects players. The more real the game world seems, the more intriguing it is to watch, and the more fun the game is to play.

The movement routines we've used so far have given fairly good movement, but to make the motion smooth, and especially to give a lifelike quality to figures, you'll need redefined characters. Remember Mario in *Donkey Kong*? Each different position he gets into is another character.

Animation in Place

The easiest sort of animation is movement within a single character. This is done by POKEing or PRINTing different characters in the same location on the screen. If the characters differ only slightly from each other, the resulting figure will seem to be moving in place. You might try this:

```
10 SC=256*PEEK(648):POKE 36879,8:PRINT "{CLR}"
20 N=86-5*(N=86):FOR I=0 TO 50:NEXT
30 POKE SC+76,N:GOTO 20
```

Other interesting effects are possible with the built-in character set. This program lets you cycle through a series of ten different animation sequences. Just press any key to get from one animation sequence to the next.

```
10 SC=256*PEEK(648):POKE 36879,8:PRINT "{CLR}":Z=1
   :X=Z
20 DIM N(40):FOR I=1 TO 40:READ N(I):NEXT
30 X=X+1:IF X>Z+3 THEN X=Z
40 POKE SC+76,N(X):IF PEEK(197)<>64 THEN Z=Z+4:X=Z
   :IF Z>40 THEN Z=1
50 FOR I=0 TO 100:NEXT:GOTO 30
100 DATA 123,126,124,108
110 DATA 76,79,80,122
120 DATA 73,75,74,85
130 DATA 115,114,107,113
140 DATA 71,66,72,103
150 DATA 103,77,100,100
160 DATA 100,111,121,98
170 DATA 73,93,85,93
180 DATA 126,97,127,226
190 DATA 95,160,223,32
```


If you'd like to see all these at once, change lines 30, 40, and 50:

```

10 SC=256*PEEK(648):POKE 36879,8:PRINT "{CLR}":X=1
20 DIM N(40):FOR I=1 TO 40:READ N(I):NEXT
30 X=X+1:IF X>4 THEN X=1
40 FOR T=0 TO 36 STEP 4:POKE SC+10+T*11,N(X+T):NEX
  T
50 FOR I=0 TO 20:NEXT:GOTO 30
100 DATA 123,126,124,108
110 DATA 76,79,80,122
120 DATA 73,75,74,85
130 DATA 115,114,107,113
140 DATA 71,66,72,103
150 DATA 103,77,100,100
160 DATA 100,111,121,98
170 DATA 73,93,85,93
180 DATA 126,97,127,226
190 DATA 95,160,223,32

```

Animation and Movement Together

Now let's take animation within one location and combine it with a simple movement routine to make an inchworm-like animation sequence across the screen:

Program 5-7. Inchworm

```

10 SC=256*PEEK(648):POKE 36879,8:PRINT "{CLR}":Z=1
  :X=21
20 DIM N(16):FOR I=1 TO 16:READ N(I):NEXT
30 Z=Z+1:IF Z>16 THEN Z=1:X=X-1:IF X<0 THEN X=16
40 POKE SC+X,N(Z)
50 FOR I=0 TO 50:NEXT:GOTO 30
100 DATA 103,106,118,225,245,244,229,160,231,234,2
  46,97,117,116,101,32

```

Notice that the entire animation sequence is completed in one position before moving to the next. Here is a slight modification of the same program that makes smooth, continuous animation across the screen. This time, however, two characters are changed every time. At first the left-hand character is fully dark, and the right-hand character is completely white. Each time through the loop a bit more of the left-hand character is light and a bit more of the right-hand character is dark, until the loop is complete. Then the character address is changed and the cycle begins again.

Program 5-8. Smooth Animation

```
10 SC=256*PEEK(648):POKE 36879,8:PRINT "{CLR}":Z=1
   :X=21:Y=20
20 DIM N(16):FOR I=1 TO 16:READ N(I):NEXT
30 X=X-1-21*(X<0):Y=Y-1-21*(Y<0)
40 FOR Z=1 TO 8:POKE SC+X,N(Z+8):POKE SC+Y,N(Z):NE
   XT
50 GOTO 30
100 DATA 103,106,118,225,245,244,229,160,231,234,2
   46,97,117,116,101,32
```

If this isn't slow enough, try it with player control added. In this program, the Commodore and SHIFT keys control left-right movement:

Program 5-9. Player-Controlled Animation

```
10 SC=256*PEEK(648):POKE 36879,8:PRINT "{CLR}":Z=1
   :X=20:T=-.125:POKE 657,128
20 DIM N(16):FOR I=1 TO 16:READ N(I):NEXT
25 DIM Y(22),R(22):FOR I=1 TO 22:Y(I)=I-1:R(I)=I-2
   :NEXT:Y(0)=21:R(0)=20:R(1)=21
30 X=X+T+22*((INT(X+T)=22)-(INT(X+T)=-1))
40 POKE SC+X,N(Z+8):POKE SC+Y(X),N(Z):POKE SC+R(X)
   ,32
45 A=PEEK(653):ON -(A=1)-2*(A=2) GOSUB 60,70
50 Z=Z-SGN(T):Z=Z+8*((Z=9)-(Z=0)):GOTO 30
60 IF SGN(T)=-1 THEN T=-T
61 RETURN
70 IF SGN(T)=1 THEN T=-T
71 RETURN
100 DATA 103,106,118,225,245,244,229,160,231,234,2
   46,97,117,116,101,32
```

If we checked for player input only once in each cycle, this program would run faster and be easier to write, but the response to the player's input would be much slower. Machine language could do the same thing so fast you could hardly see what was going on. This is one area where, even though the programming *can* be done in BASIC, it just isn't worth the effort to get such detailed animation.

All these demonstrations have used the abstract built-in graphics characters. Here's a program that shows how to make your VIC produce more humanlike motion. We'll use the characters we designed in Chapter 4 to create a running and jumping athlete. To make the athlete run, press the CURSOR LEFT/RIGHT

key; to make the athlete jump, press the CURSOR UP/DOWN key.

Program 5-10. Athlete in Action

```

10 POKE 36869,255:POKE 52,28:POKE 56,29:CLR
20 GOSUB 900:PRINT"{CLR}"
30 DIM CH$(6)
35 CH$(1)=CHR$(33):CH$(0)=CHR$(34)+CHR$(35):CH$(2)
   =CHR$(36):CH$(3)=CHR$(37)
40 CH$(4)=CHR$(39)+"{UP}{LEFT}"+CHR$(38):CH$(5)=CH
   R$(37)+"{LEFT}{UP} ":CH$(6)=CHR$(36)
50 S$="{HOME}{DOWN}":FOR I=1 TO 16:S$=S$+"{DOWN}":
   NEXT:T=130
60 GOSUB230
100 A=PEEK(197):ON -(A=23)-2*(A=31) GOSUB 200,250:
   GOTO 100
200 PRINT S$TAB(P)CH$(0):FOR X=0 TO T:NEXT
210 Q=P:P=P+1:IF P>20THEN P=0:PRINT S$TAB(Q)"
   {2 SPACES}":GOTO 230
220 IF PEEK(197)=64 THEN 220
230 PRINT S$TAB(Q)" "S$TAB(P)CH$(1):FOR X=0 TO T:N
   EXT:RETURN
250 FOR I=2 TO 6:PRINT S$TAB(P)CH$(I):FOR X=0 TO T
   /4:NEXT
260 IF PEEK(197)=64 THEN 260
270 NEXT:RETURN
900 FOR C=7424 TO 7487:READ D:POKE C,D:NEXT:RETURN
905 DATA 0,0,0,0,0,0,0,0
910 DATA 24,24,18,124,88,20,100,0
915 DATA 1,1,1,3,6,1,3,0
920 DATA 128,128,32,224,128,64,32,0
925 DATA 0,56,56,16,124,16,124,68
930 DATA 56,56,16,124,16,124,68,68
935 DATA 0,0,0,56,56,84,124,16
940 DATA 56,40,40,0,0,0,0,0

```

In the initialization routine (lines 10-60) the string array CH\$(n) is set up to hold the characters or groups of characters that will produce each stage in the figure's movement. Sometimes a single character will show the full figure, but in midrun and midjump, two characters must be printed, along with needed cursor-movement characters and spaces required to erase other characters. S\$ is set up to home the cursor and then drop down the correct number of lines.

The main loop at line 100 is one line long, doing nothing but getting keyboard input and deciding whether to make the figure run or jump.

5 Getting Your Figures Moving

The run subroutine at line 200 is the most complex part of the program, since the figure must not only be PRINTed, but must also wrap around when it reaches the end of the line. The delay loops (FOR X = 0 TO T:NEXT) are present to show how you can change the smoothness and speed for different effects. By changing the value assigned to T in line 50, you can make the runner's movement slower but make the individual steps clearer, or you can speed up the figure to a virtual blur. The value shown here, 130, emphasizes the individual steps. A value of zero will of course be much faster.

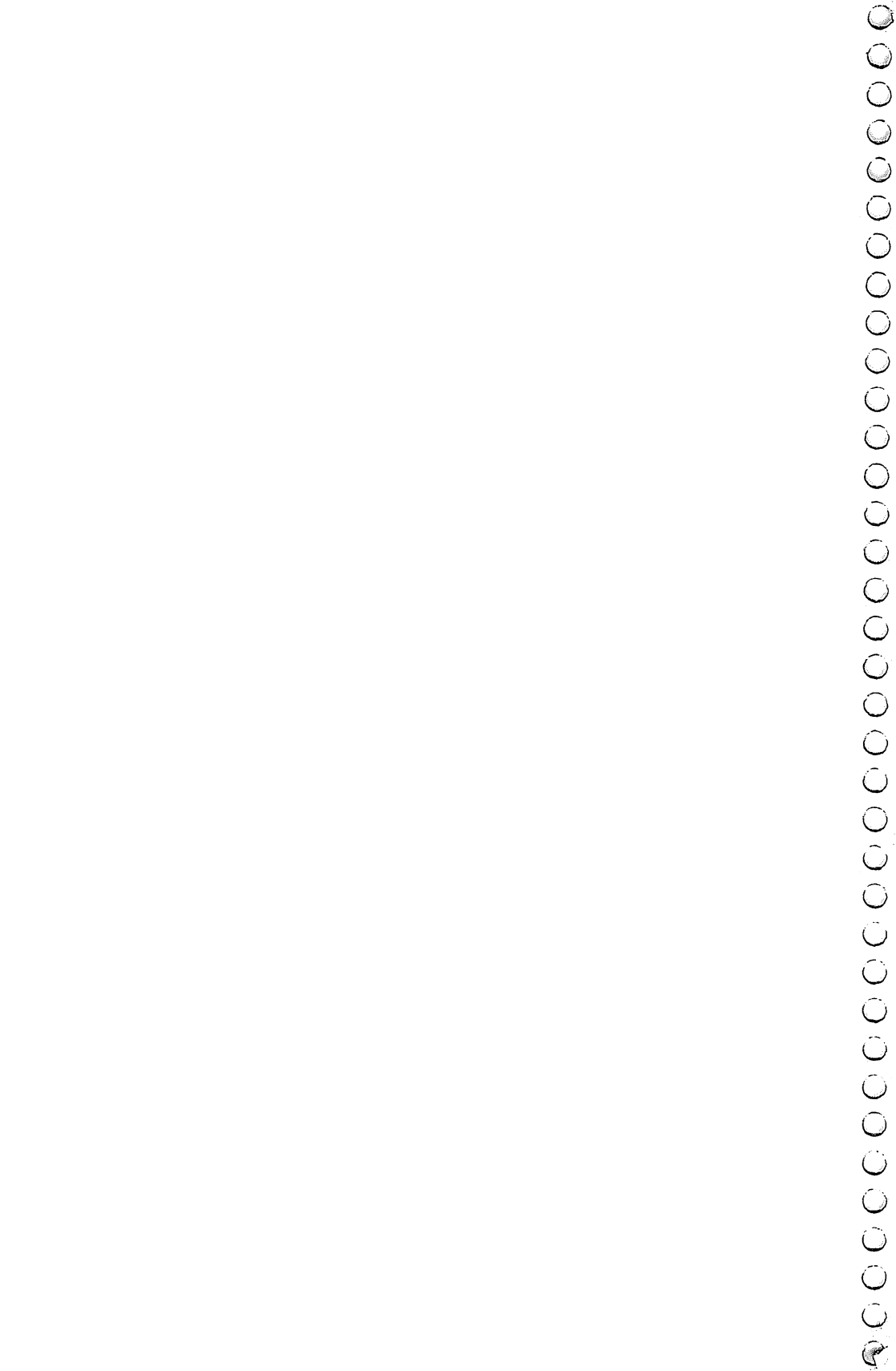
The jump subroutine at line 250 is much simpler; even though jumping requires five steps (FOR I = 2 TO 6), jumping never has to wrap around the end of a line.

The character set routine at line 900 is the one we used in Chapter 4 to set up the athlete characters.

If you want to devote the programming time and have the computer memory to do it, you could add many other intermediate characters to get far more fluid, lifelike motion. It's a trade-off. Especially when you're programming in BASIC, every feature you decide to add will cost you something somewhere else. To get speed, you must keep your figures small and your motions fairly jerky. To get large, detailed figures, you must sacrifice speed, memory, and smoothness. To get natural-looking animation, you must give up speed and memory. It should always be decided on the basis of what is more important for the game. Some games need animation; with many others it would add nothing.



6 **Collisions**



6 Collisions

When Pac-Man eats dots, points get added to the score and the dot disappears; when he eats a power dot, the ghosts change color for a while. When Pac-Man touches a ghost, he dies — unless he recently ate a power dot. Most of the things that happen in a videogame happen because something on the screen bumped into something else.

There are other things that can cause changes. In *Donkey Kong*, for instance, if it takes you too long to get to the top of the screen, you run out of time and Mario wipes out. In most games, if your score reaches a certain level, you get a bonus player-figure, an extra turn. But these are exceptions: the overwhelming majority of events are caused by collisions.

Checking for Collisions

Checking for collisions is really very simple. All you have to do to see if your figure is touching something is PEEK at the address in screen memory where the figure is about to be PRINTed or POKEd and see what's there.

Remember back in Chapter 3 when we created a screen display made of stars? Let's put that display on the screen and move a spaceship around on it using the wraparound movement routine from Chapter 5. But before we move the spaceship, we'll check to see if there's a star in the place where the spaceship is trying to go. If there is, the spaceship can't make the movement. Use the Commodore and SHIFT keys for left/right movement and f5 and f7 for up/down movement.

This program isn't quite a game yet, but as you move around the screen, you'll see that we're getting very, very close.

Program 6-1. Spaceship Collisions

```
10 SC=PEEK(648):CM=37888+256*(SC AND2):SC=SC*256:G
  OSUB 500
20 FG=90:OC=PEEK(CM)AND15:NC=5
30 POKE 657,128
90 GOTO 280
100 A=PEEK(653)AND3:B=PEEK(197)
190 H=(A>1)-(A=1):V=(B=55)-(B=63):IF H=0 AND V=0 T
  HEN 100
200 NH=OH+H+22*((OH=21 AND H=1)-(OH=0 AND H=-1))
```

6 Collisions

```
210 NV=OV+V+23*((OV=22 AND V=1)-(OV=0 AND V=-1))
220 A=PEEK(SC+NH+NV*22):IF A<>32 THEN NV=OV:NH=OH:
    GOTO 100
280 POKE CM+OH+22*OV,OC:POKE CM+NH+22*NV,NC
290 POKE SC+OH+22*OV,32:POKE SC+NH+22*NV,FG:OH=NH:
    OV=NV:GOTO 100

500 POKE 36879,8:PRINT "{BLK}{CLR}":Q=100*RND(9):Q
    1=6*RND(9)
510 Q=80*RND(9)+20:Q1=6*RND(9)+2
520 FOR I=0 TO Q:X=505*RND(9):N=46:GOSUB 540:NEXT
530 FOR I=0 TO Q1:X=505*RND(9):N=81:GOSUB 540:NEXT
    :RETURN
540 POKE SC+X,N:RETURN
```

Line	Function
10-90	This is the program initialization handler. Since programs always start running at the lowest-numbered line, you always have to have at least one initialization line at the beginning of the program, even if all it does is say GOTO 1000, where the real initialization takes place. You might notice that this section ends with line 90, but there are no line numbers from 30 to 90. This is so more initialization routines can be added later, either with a few commands or a GOSUB to a more complex routine.
10	Set SC at the start of screen memory and CM at the start of color memory. GOSUB 500 to draw the screen.
20	Set FG to the screen code of the character that will serve as our spaceship figure — in this case, the diamond shape. Set OC to the current color memory value (white) and NC (new color) to the color of the spaceship, green.
30	Disable character set switching.
90	Go to the end of the movement loop to draw the spaceship on the screen and then wait for the player's instructions.
100-190	This is the main loop of the program. Any command in this section of the program will be executed every time the program passes through the loop, whether the player has called for movement or not. Opponents' movements, timing, display changes, and so forth are all controlled through this loop. As before, there is plenty of space left for more lines.
100	Set A to show whether SHIFT or Commodore is pressed. Set B to show if any other key has been pressed.

- 190 Set *H* and *V* to their appropriate values if the right keys have been pressed. If no movement is called for, return to the beginning of the loop at line 100.
- 200-290 This is the player-figure movement handler. The program won't execute these lines unless the player has commanded a movement. As always, there is plenty of room left to add other routines. But any routine that is accessed from this section will be executed *only* when the player commands a movement.
- 200-210 Set *NH* and *NV* to the new values if this movement is executed, including wraparound if necessary.
- 220 Set *A* to the value of whatever character is already at the new player location. If it is not a blank (screen code 32), cancel the movement by setting *NH* and *NV* to equal *OH* and *OV*; then return to the main loop at line 100.
- 280 Erase the spaceship color at the old location and **POKE** it into color memory at the new location.
- 290 Erase the spaceship at the old location and **POKE** it into screen memory at the new location. Return to the main loop at line 100.
- 500-540 The screen setup subroutine. This is almost identical to the random starfield program in Chapter 3.

Making Things Happen

Now we have a spaceship moving through the starfield, and our program notices when it collides with stars. All we need now to make it a game is to have more things happen, some of them depending on the player's choices, and some of them happening automatically.

Let's say that our spaceship is on a fuel-gathering mission. It needs to visit large nova stars, to gather rare gases from the clouds surrounding them. Unfortunately, if the spaceship passes too close to regular, smaller stars, it suffers damage — too many such incidents before the spaceship reaches a space station will cause the spaceship to malfunction, and the crew will have to abandon it.

How does this story translate into programming?

Collision handling. We'll allow the player-figure to move onto the squares occupied by stars and novae. This time, however, we will be keeping score. Every time the ship collides with a small star, the player loses points. Also, it will come one step closer to destruction, which is signified by changing the space-

ship's color from red to cyan, purple, green, blue, and yellow.

Every time the ship collides with a nova (large star), the player wins points, and the nova, all its important gases stripped away, appears on the screen as a small star.

A new element will be introduced — a space station, which will be a red checkerboard square (screen code 102). The spaceship changes back to red when it collides with the space station, but gets no additional points. After a while, the spaceship gets out into deep space and the space stations stop appearing.

When the spaceship moves onto a small star or the space station, the program must “pick up” the star and then put it back when the spaceship leaves. With the novas, however, since the spaceship has picked up all the gases surrounding them, the program will put a small star in their place when the spaceship moves away.

The moving starfield. Obviously, the spaceship will soon get all the available novas. For the game to continue, the screen has to be refreshed. One way to do this might be to create a new random starfield whenever the last nova has been taken. This is what *Pac-Man* does, when all the dots have been eaten. But it stops the game each time, and forces us to make the program keep track of the number of novas on each screen. A better solution for this game is to have a scrolling display — every now and then, the whole starfield moves upward one or more lines, and a new line of randomly generated stars is added at the bottom of the screen.

As the game goes on, the screen scrolls more often and adds more lines at a time. Naturally, this means the player will have to move faster and faster to get the novas before they scroll off the top of the screen. Also, since the display is moving, it will be possible for the small stars to collide with the spaceship.

How will we do the scrolling? Scrolling is always handled on the VIC by doing a block move — taking the bytes stored at locations A1 through Z1 and moving them to locations A2 through Z2. In BASIC, this would be unbearably slow — the loop to move all 506 bytes of screen memory *and* all 506 bytes of color memory would take several seconds to execute and would slow down the game far too much.

Fortunately, there is a machine language block-move routine built right into VIC BASIC. When you PRINT a line at the bottom of the screen, it makes the whole screen scroll upward to make room for the cursor to move down to the next line. All we need to do to make the screen scroll is to PRINT the new line of stars at

the bottom of the screen, and the VIC automatically makes our scrolling display.

Creating new lines of random stars. One of the problems now is to create those new lines of stars to PRINT at the bottom of the screen. They have to be random, and yet they have to appear almost instantly in order not to slow down the game.

The solution is to create one long string, 255 characters long, which contains a random assortment of blanks, small stars, and novas. Then, using the MID\$ function, we can choose a segment of that string to PRINT each time we need a new line of stars. By generating a random number for the starting position of the string segment, we can still keep up the randomness — but we have to generate only *one* random number each time we scroll, and then the built-in machine language MID\$ function and PRINT command will take care of the rest. There we have it — machine language speed in a BASIC program.

Program 6-2. Mission: Noval

```

10 SC=PEEK(648):CM=37888+256*(SC AND2):SC=SC*256:G
   OSUB 500
20 FG=90:OC=PEEK(CM)AND15:NC=2:PRINT "{WHT}":Q=3:Q
   Q=Q+1:GOSUB 800
30 POKE 657,128:GOSUB 400
40 NH=7:NV=7:C=NH+NV*22:W=32:SS=102:CS=2:TS=0
90 GOTO 290
100 A=PEEK(653)AND3:B=PEEK(197)
110 IF VAL(TI$)>Q THEN GOSUB 600
190 H=(A>1)-(A=1):V=(B=55)-(B=63):IF H=0 AND V=0 T
   HEN 100
200 NH=OH+H+22*((OH=21 AND H=1)-(OH=0 AND H=-1))
210 NV=OV+V+(OV=22 AND V=1)-(OV=1 AND V=-1)
220 POKE CM+C,OC:POKE SC+C,W:C=NH+NV*22:W=PEEK(SC+
   C)
230 IF W<>32 THEN GOSUB 700
280 POKE CM+C,NC:POKE SC+C,FG:IF NC>7 THEN 900
290 OH=NH:OV=NV:GOTO 100
400 FOR I=0 TO 254:T=32
410 IF INT(RND(9)*5)<1 THEN T=46:IF INT(RND(9)*5)<
   1 THEN T=113
420 T$=T$+CHR$(T):NEXT
430 M$=CHR$(19):FOR I=0 TO 22:M$=M$+CHR$(17):NEXT:
   RETURN
500 POKE 36879,8:PRINT "{BLK}{CLR}":Q=100*RND(9):Q
   1=6*RND(9)
510 Q=80*RND(9)+20:Q1=6*RND(9)+2
520 FOR I=0 TO Q:X=505*RND(9):N=46:GOSUB 540:NEXT

```

6 Collisions

```
530 FOR I=0 TO Q1:X=505*RND(9):N=81:GOSUB 540:NEXT
:RETURN
540 POKE SC+X,N:RETURN
600 POKE SC+C,W:POKE CM+C,OC:T=1+INT(RND(9)*167)
605 PRINT M$;MID$(T$,T,22*INT(QQ-Q)-1);:P=P-INT(7*
Q):GOSUB 800
610 OC=PEEK(CM+C):W=PEEK(SC+C):IF W<>32 THEN GOSUB
700
615 POKE SC+C,FG:POKE CM+C,NC
620 PRINT "{HOME}"TAB(8)STR$(P)" POINTS{3 SPACES}"
630 TS=TS+1:IF TS>10*(QQ-Q)THEN GOSUB 850
640 IF NC>7 THEN 900
690 RETURN
700 IF W=46 THEN P=P-100:NC=NC+1:RETURN
710 IF W=102 THEN NC=CS:W=104:RETURN
720 IF W=104 THEN RETURN
730 P=P+(8*NH)*INT(QQ-Q):W=46:RETURN
800 TI$="000000":Q=99*(Q/100):RETURN
850 TS=0:T2=506-INT(RND(9)*22):POKE SC+T2,SS:POKE
{SPACE}CM+T2,CS:RETURN
900 FOR I=0 TO 20:FOR X=0 TO 7:POKE CM+C,X:NEXT:NE
XT
910 PRINT "{CLR}"P" POINTS":PRINT:PRINT:PRINT "THE
END":END
```

Line **Function**
10-90 *Initialization.* Most variables are assigned their starting value here.

Variables Used

SC = starting address of screen memory
CM = starting address of color memory
FG = screen code of the character used for the player-figure
OC = the original color of every square in color memory
NC = the current color of the player-figure
Q = the difficulty level. This starts at 3, then decreases steadily throughout the game.
QQ = the original difficulty level + 1; Q is subtracted from QQ several times during the game to set certain values that must get larger rather than smaller as the game gets harder.
NH = the new horizontal position (column number) of the player-figure
NV = the new vertical position (row number) of the player-figure
OH = the old horizontal position of the player-figure
OV = the old vertical position of the player-figure
C = the number of bytes to add to SC and CM to find the player-figure's current location in memory

W = the character that was stored at the new player position *before* the player-figure moved there; it is usually put back after the player-figure moves on.
 SS = the screen code of the character that represents the space station
 CS = the color of the space station
 TS = the variable used to time the appearance of the space station

Subroutines Called

GOSUB 400 sets up the long random string that will create the new lines added to the bottom of the screen.

GOSUB 500 sets up the screen display.

GOSUB 800 initializes the timer function that controls how often the screen will scroll.

- 100-190 *Main loop.* Everything that happens constantly, whether the player moves the player-figure or not, is controlled in this section.
- 100 Set A and B to show what keys, if any, are pressed.
- 110 Check the timer to see if it's time to scroll the screen. (The variable TI\$ is automatically changed by the operating system to show how much time has passed since it was last set to equal 0. If you don't know how to use TI\$ and can't figure it out from this line and the subroutine at 800, see *Programmer's Reference Guide to the VIC.*) If it is, jump to the scroll routine at 600.
- 111-189 Open lines, where other routines can be added.
- 190 Set H and V to equal the horizontal and vertical movement called for by the player. If no movement is called for, go back to line 100.
- 200-290 *Movement loop.* This is where player movement is carried out, along with any other routines that happen only when the player moves.
- 200-210 Set NH and NV to the new player-figure location.
- 220 Set the old player-figure location back to its previous values. POKE color memory with the nonplayer-figure color, OC, and POKE screen memory with W, the character that should be left behind when the player leaves that spot. Then set C to equal the new player-figure offset, and set W to equal the character currently stored at the new location.
- 230 If the character stored at the new location is something other than a blank (32), go to the collision routine at line 700.

6 Collisions

- 231-279 Lines available for other routines.
- 280 POKE the player-figure and its color into screen memory and color memory. If the color is greater than 7, then the player-figure has collided with too many small stars — jump to the end routine at line 900.
- 290 Set OH and OV to equal NH and NV; then jump to the main loop at line 100.
- 400-430 *Random line generator.* This routine creates the 255-character string T\$, assigning each character in the string the value T; T has a randomized chance of being 32 (a blank), 46 (a small star), or, the least likely chance, 113 (a nova). Then the string M\$ is set up to equal the right number of CURSOR-DOWN characters to PRINT the random segment of T\$ at the bottom of the screen.
- 500-540 *Screen setup.* The same routine we've used before to generate a random starfield.
- 600-690 *Scroll routine.* This section controls everything that happens whenever the screen scrolls.
- 600 Erase the player-figure by POKEing OC and W into color and screen memory. Since we don't want the player-figure to move upward when the rest of the screen scrolls, we will erase the player-figure, execute the scroll, and then POKE the player-figure back into the same location in memory. The starfield will seem to move, while the spaceship will seem to hold still.
- 605 Then set T to a random number between 0 and 166. Scroll the screen by PRINTing M\$ and the segment of T\$ starting at position T. At the easiest level, PRINT a segment only one screen line in length. At harder levels, PRINT up to four screen lines at a time.
- Then subtract from the player's score a number of points equal to seven times the difficulty level, so that at harder levels of play, fewer points will be subtracted each time the screen scrolls. This subtraction is included so that players will have a greater incentive not to simply sit there avoiding the moving stars.
- 610 Then jump to the timer subroutine at line 800. Set OC and W to the values that are now, after the scroll, in the player's location in color memory and screen memory. If W is not a blank, jump to the collision routine at line 700.
- 615 POKE the spaceship and spaceship color into memory.

- 620 PRINT the updated score at the top of the screen. Notice that the program PRINTs STR\$(P) instead of simply PRINTing P. This is because the VIC automatically skips a space after PRINTing numeric variables. If a number was already in that space, it will be left there, which makes the score inaccurate if the number of digits in the score changes. Perhaps the best way to see how this works is to change the statement to PRINT "{HOME }"TAB(8)P" POINTS " and see what happens.
- 630 Add one to the Space Station Timer (TS); depending on the difficulty level, if TS is beyond the limit, jump to the space station routine at line 850.
- 700-730 *Collision routine.* There are four possible collisions.
If the spaceship has collided with a small star (46), the score (P) is decreased by 100, and NC, the spaceship's color, is increased by one.
If the spaceship has collided with a new space station (102), NC, the spaceship's color, is set back to 2. Then W is set to 104, a "used" space station, so that it can't be used again.
If the spaceship has collided with a used space station (104), nothing happens.
If the spaceship has collided with a nova (the only other possibility), the score is increased, with greater increases at the higher difficulty levels and at lower positions on the screen. W is then set to 46, a small star, so that it becomes another obstacle to the spaceship.
- 800 *Timer reset routine.* The variable TI\$ is set back to 000000, which starts it over again. Also, the difficulty level Q is set to a slightly lower number. To make the game get harder faster, change the 99 to a lower number; to make it stay easy longer, change the 99 to 99.5 or 99.8 — any number greater than 99 but less than 100.
- 850 *Space station routine.* This one-line subroutine puts a space station in a random location on the bottom line of the screen and sets its color to red.
- 900 *End routine.* This routine makes the spaceship flash through the colors, prints the final score, and exits the game.

Star-Eater

Here's a simple gobble game. You are a hungry dot, out to eat stars (asterisks). You get points for eating them. The trouble is, each one you eat turns into a plus — and if you accidentally eat a plus, you *lose* points. Also, every plus you eat turns into a block of stone, which you can't get past. Your task is to eat as many stars as you can before time runs out. You can wrap around the screen both horizontally and vertically; left-right movement is done with the SHIFT and Commodore keys, and up-down movement is done with f5 and f7.

With what you already know about game programming, you should have little trouble figuring out exactly what's going on. Lines 0-99 are initialization; lines 100-199 are the main loop; lines 200-299 are the movement loop; lines 300-399 are the end routine; lines 600-699 are the set level subroutine; and lines 700-799 are the collision routine. It is all familiar ground.

Let me just call your attention to a few details.

Notice the variable *M*. During initialization, it is set to the total number of stars on the screen by adding 1 to *M* each time a star is POKEd to the screen (that is, each time the random number *X* is a 3). Then, during the collision routine, *M* is decreased by 1 each time the player eats a star. This means that when the last star is eaten, *M* will equal zero — and that is *one* of the conditions that will end the game. Upon entering the end routine at line 300, you get different messages, depending on whether or not you ate all the stars.

Another new feature is the choice of levels. At the beginning of the game, you are asked what level you want. This is accomplished in the subroutine starting at line 600. If you choose "easy," the variable *N* is set to 300; 200 if you choose "hard"; and 100 if you choose "superhuman."

Notice how this variable is used throughout the program. In the Collision Routine, the score is *added to* by an amount equal to *N* minus the current value of the timer (TI\$), but when the player eats a plus, the score is *subtracted from* by an amount equal to TI\$. That means that the later you get in the game, the fewer points you get for each star you eat, and the more points you lose for each plus you eat.

Also, the TIME message at the top of the screen (see line 100) tells you how much time you have left. The countdown is created by subtracting TI\$ from *N*. And if you don't eat all the stars before

the timer reaches the value of N , the game ends — with a different message at line 300.

Program 6-3. Star-Eater

```

5 DIM CH(3):CH(0)=102:CH(1)=32:CH(2)=43:CH(3)=42
10 DIM CL(3):CL(0)=2:CL(1)=1:CL(2)=5:CL(3)=7
15 SC=PEEK(648):CM=37888+256*(SC AND 2):SC=SC*256
20 PRINT "{CLR}":FG=81:CF=0:M=0:GOSUB 600
25 FOR I=22 TO 505:X=INT(RND(1)*4):M=M-(X=3)
30 POKE SC+I,CH(X):POKE CM+I,CL(X):NEXT
35 TI$="000000":OH=10:OV=10:FL=OH+OV*22:OC=CL(1):W
  =CH(1)
40 POKE SC+FL,FG:POKE CM+FL,CF
100 PRINT "{HOME}SCORE "STR$(P)" TIME "STR$(N-VAL(
  TI$))" "
110 A=PEEK(653)AND 3:B=PEEK(197)
120 IF VAL(TI$)>N OR M=0 THEN 300
190 H=(A>1)-(A=1):V=(B=55)-(B=63):IF H=0 AND V=0 T
  HEN 100
200 NH=OH+H+22*((OH=21 AND H=1)-(OH=0 AND H=-1))
210 NV=OV+V+22*((OV=22 AND V=1)-(OV=1 AND V=-1))
220 POKE SC+FL,W:POKE CM+FL,OC:FL=NH+NV*22:E=W
230 W=PEEK(SC+FL):OC=PEEK(CM+FL):IF W<>32 THEN GOS
  UB 700
280 POKE CM+FL,CF:POKE SC+FL,FG
290 OH=NH:OV=NV:GOTO 100
300 PRINT "{CLR}":IF M=0 THEN 320
305 IF M=0 THEN PRINT "{2 DOWN}YOU GOT ALL THE STA
  RS!"
310 PRINT "TIME'S UP!":PRINT "STARS LEFT: "M:GOTO
  {SPACE}325
320 PRINT "YOU GOT ALL THE STARS!":PRINT "TIME LEF
  T: "N-VAL(TI$)
325 PRINT "SCORE: "P
330 PRINT "{3 DOWN}PLAY AGAIN? (Y OR N)"
340 B=PEEK(197):IF B=28 THEN END
345 IF B=11 THEN RUN
350 GOTO 340
600 PRINT "{6 SPACES}STAR-EATER":PRINT "{2 DOWN}CH
  OOSE YOUR LEVEL:"
605 PRINT "1 - EASY":PRINT "2 - HARD":PRINT "3 - S
  UPERHUMAN"
610 N=PEEK(197):IF N=64 THEN 610
620 N=300+100*(N=56)+200*(N=1):PRINT "{CLR}":RETUR
  N
700 IF W=CH(0) THEN 730
710 IF W=CH(2) THEN P=P-VAL(TI$):W=CH(0):OC=CL(0):
  RETURN

```

```

720 P=P+INT(N-VAL(TI$)):W=CH(2):OC=CL(2):M=M-1:RET
URN
730 NH=OH:NV=OV:FL=OH+OV*22:W=E:OC=PEEK(FL+CM):RET
URN
3345 IF B=11 THEN RUN

```

Collision Tracking

So far we have only used PEEKs to see whether a collision is taking place. There is a good reason for this. Most of the time, you will be creating games where there are random elements, things on the screen that your program won't be keeping track of. There'll be no way for the computer to know there's been a collision without PEEKing.

In some games, however, the only things on the playfield besides the player-figure will be computer-controlled figures, and your program will therefore be keeping track of the location of every single item on the screen. Then you can find out about collisions by comparing the locations of the various items. The game "Moonraker," Program 10-3, uses this method. But it only works when there are few figures on the screen and no possibility of random items to run into.

Spark: Combining PRINT with POKES and PEEKS

One last game before we move on to sound and music. One thing we haven't done yet is make the computer control a figure moving around the screen. In the game "Spark," we'll use a screen display we created in Chapter 3, except that it is in reverse video. You have a spark on a wire, racing around and around. It's your job to get the spark off the end of the wire. But you don't control the spark — instead, you control an interruption in the wire. The spark will keep moving at a steady pace, controlled by the computer.

Using f5 (up) and f7 (down), you control the vertical position of the break in the circuit. The SHIFT and Commodore keys control its horizontal position. Press SHIFT and it has one horizontal position; press Commodore and it has the other; press SHIFT and Commodore together and the break completely disappears. But this is one case where a picture is worth more than words — you have to practice the game to really understand what your controls will do.

The top and bottom of the wire also shift randomly, constantly undoing your work. You have a time limit — your score is

how much time you had left when you got the spark off the wire.

As you examine the program, you'll see that the spark never exists in screen memory at all — it is only POKEd into color memory. But the spark's movement routine in the main loop from lines 100 to 190 PEEKs into screen memory in order to see in what direction it is supposed to go next. Using PRINT for screen changes gives this game machine language speed in its response to the player's keyboard input. This method won't work for all programs, of course, but for this one, the computer is running a fairly fast-moving figure through a fairly complex movement pattern, all *in BASIC*. If you program carefully enough, you can get real speed in BASIC games.

Program 6-4. Spark

```

10 DIM A$(6),S$(8):POKE 36879,1:PRINT "{CLR}{WHT}"
   :GOSUB 500
20 FOR I=0 TO 7:PRINT S$(I)A$(0):NEXT I
30 SM=PEEK(648):CM=37888+256*(SM AND 2):SM=SM*256
40 OC=PEEK(CM):NC=2:OL=32:NL=54
50 DIM MV(4,4):GOSUB 700:FR=0:CH=4:P=300:POKE 657,
   128
60 Y=1:GOSUB 310:A=2:PL=6:GOSUB 200
100 IF VAL(TI$)>1 THEN GOSUB 300
110 POKE CM+NL-MV(CH,FR),OC:POKE CM+NL,NC
120 A=PEEK(653)AND3:B=PEEK(197):IF A>0 THEN GOSUB
   {SPACE}200
130 IF B<64 THEN GOSUB 250
140 E=PEEK(SM+NL):GOSUB 400:CH=E
145 IF CH=4 AND YY<>0 THEN GOSUB 450
150 E=PEEK(SM+OL):GOSUB 400:FR=E:IF CH=FR THEN FR=
   4+(FR=4)+D
160 IF CH<4 THEN OL=NL
170 IF CH<0 THEN 600
180 NL=NL+MV(CH,FR):D=SGN(MV(CH,FR))=-1
190 YY=0:GOTO 100
200 PH=A:IF A=3 THEN PH=0:GOTO 290
250 PRINT S$(PL)A$(0):YY=SGN(MV(CH,FR))
260 PL=PL+(B=55)-(B=63):PL=PL+8*(PL>7)-8*(PL<0)
290 PRINT S$(PL)A$(PH):RETURN
300 Y=INT(RND(9)*2):P=P-10
310 PRINT "{HOME}{2 SPACES}TIME LEFT{3 SPACES}"STR
   $(P) " "
320 PRINT "{HOME}{DOWN}"A$(3+Y)S$(8)A$(6-Y)
330 IF P=0 THEN 600
340 TI$="000000":RETURN
400 E=E-128-73:IF E>2 THEN E=E-9:IF E>3 THEN E=E-7
410 RETURN

```

6 Collisions

```
450 IF YY=-1 THEN OL=CH-22*INT(CH/22)+396:RETURN
460 OL=CH-22*INT(CH/22)+22:RETURN
500 FOR I=0 TO 6:A$(I)=CHR$(18):NEXT I
520 FOR X=1 TO 20:A$(0)=A$(0)+CHR$(221):NEXT
530 FOR I=1 TO 5 STEP 4:FOR X=1 TO 10:A$(I)=A$(I)+
  CHR$(106)+CHR$(107):NEXT:NEXT
540 FOR I=2 TO 6 STEP 4:A$(I)=CHR$(221):FOR X=1 TO
  9
545 A$(I)=A$(I)+CHR$(106)+CHR$(107):NEXT:A$(I)=A$(
  I)+CHR$(221):NEXT
550 FOR I=0 TO 2:A$(I)=A$(I)+CHR$(13)+CHR$(18):NEX
  T
560 FOR X=1 TO 20:A$(0)=A$(0)+CHR$(221):NEXT
570 FOR I=1 TO 3 STEP 2:FOR X=1 TO 10:A$(I)=A$(I)+
  CHR$(117)+CHR$(105):NEXT:NEXT
580 FOR I=2 TO 4 STEP 2:A$(I)=A$(I)+CHR$(221):FOR
  {SPACE}X=1 TO 9
585 A$(I)=A$(I)+CHR$(117)+CHR$(105):NEXT:A$(I)=A$(
  I)+CHR$(221):NEXT
590 S$(0)=CHR$(18)+"{HOME}{2 DOWN}":FOR I=1 TO 8:S
  $(I)=S$(I-1)+"{2 DOWN}":NEXT:RETURN
600 PRINT "THE END":END
700 FOR I=0 TO 4:FOR X=0 TO 4:READ MV(I,X):NEXT:NE
  XT:RETURN
710 DATA 0,-1,-1,22,-1,1,0,-22,1,1,-1,-22,0,-1,-1
720 DATA 22,1,1,0,1,22,-22,-22,22,0
```



7
Sounds
and Music



7

Sounds and Music

Just how important is sound in a videogame? Try playing a favorite game with the sound turned off. You'll be amazed at how much you'll miss the various tones, noises, and melodies that you are accustomed to hearing while you play the game. Your timing will probably be off, your score will be lower, and most important, you won't enjoy the game as much. Why?

Uses of Sound

Sounds help you while you play a videogame by providing feedback of your game actions. As you play *Joust*, for example, there are sounds which give you information on almost every event that occurs. A collision in which you defeat the other knight is different from one in which *you* are defeated. Colliding with different surfaces results in different sounds. When you receive a bonus knight, there is yet another sound. These sounds aid you as you play, for as you concentrate on keeping your own knight alive, the sounds tell you what's going on elsewhere on the screen. *Pac-Man* also provides sounds as information, although to a lesser degree. Each dot that is eaten, and thus each point that is awarded, produces a sound. You *know* you are gaining points without having to actually look to check. When a power dot is eaten, another sound is made which signals that ghosts can now be chased. Sound as information is a valuable aid in a game.

Sounds also affect your mood as you play, drawing you into the game's unique world. Imagine the loss you feel when your *Pac-Man* is caught by one of the ghosts and the short "Sorry" sound routine plays. The fierce stamping sounds at the opening of *Donkey Kong* only urge you on to the top. Never mind the anger of the ape, you tell yourself. Part of this mood is generated by the pace of the sounds and music.

One of the most successful arcade games, *Space Invaders*, uses a repetitive sound as the aliens move across the screen. As their numbers decrease, the sounds become louder and faster. The effect is one of increasing excitement and anxiety.

Many games have different levels of difficulty. Perhaps your game will, too. Often the screen alters slightly when a new level of difficulty is reached. A faster background sound or a higher-pitched noise can increase the pace of the action. The high-pitched, modulating drone that seems to yell "Danger! Danger!" as the ghosts in *Pac-Man* chase you is a good example.

Pay close attention to the sounds you hear the next time you play a video arcade game. What sounds are used for what effects? What do those sounds do to and for the player? How would the game be less enjoyable or harder to play if the sounds were eliminated? Seeing how other game creators use sound will help you decide how and when to use sound in your own games.

Creating Sounds

The VIC creates sounds by addressing five memory locations, called sound registers. The sound output is created and controlled by POKEing values to these memory locations. The memory locations and corresponding sound registers are:

Memory Location	Sound Register
36874	Low tone register
36875	Middle tone register
36876	High tone register
36877	Noise register
36878	Volume control register

Each tone register can produce 128 tones or notes, simply by using a POKE command in this way:

POKE y,x

where y is one of the five sound register memory locations and x equals a number between 128 and 255. For example, using the command POKE 36874,128 produces the lowest possible tone, approximately a low B.

The noise register can be addressed in the same way, but instead of producing a tone, it will create "white noise." Varying the value will change the noise generated from a bass roar (128) to a piercing hiss (255).

The fifth sound register controls the volume of the sound produced on the VIC. There are 16 volume settings, ranging from 0 to 15. Zero, however, turns the sound off, so you must enter a value between 1 and 15. Setting the volume register alone will not

produce a sound. You also must set one or more of the tone registers for sound to be created.

There are numerous sources for small routines that duplicate various sounds or sound effects. One such source is *Personal Computing on the VIC-20*, the manual that came with your computer. Appendix G in the back of the manual has a number of sound effects already worked out. In COMPUTE!'s *Second Book of VIC*, John Heilborn's article, "Making Sound with Blips," shows you how to do really convincing sound effects and includes some excellent examples. Such prewritten routines are useful if you experiment with them, changing values or adding time delays to slow the sounds down.

The two following routines can be used in just this way. Type each in and save, so you can later load them for alteration.

Program 7-1. Zap

```
10 L=36874
20 M=36875
30 H1=36876
40 N=36877
50 V1=36878
100 POKE N,175
110 FOR B1=1 TO 15
120 POKE V1,B1
130 FOR T=1 TO 5
140 NEXT T
150 NEXT B1
160 POKE N,0
170 POKE V1,0
```

This program produces a short "zap" sound, something you may find useful in a game that has objects moving quickly, or perhaps for firing projectiles. Notice that the values for the memory locations have been placed at the start of the routine and given a variable name. This just makes it simpler to type in a program, which may use a memory location several times. See what happens when you change the numbers in line 100, where the noise register is being used; in line 110, which changes the volume from 1 to 15; and in line 130, which is a short delay. Try it for yourself.

Replace lines 100-170 in the first program with these lines. Remember to keep lines 10-50, or there will be no memory locations available for the routine.

Program 7-2. Blip

```
100 POKE V1,15
110 FOR B1=1 TO 5
120 FOR B2=250 TO 240
130 POKE H1,B2
140 NEXT B2
150 POKE H1,0
160 NEXT B1
170 POKE V1,0
```

This routine creates a short "blip," another sound effect you may find useful in your own game. Again, by altering the numbers in lines 100, 110, and 120, you can change the sound produced.

Notice in these sample routines that values of zero are POKEd in for both the sound (or noise) and volume registers. This is the easiest way to turn off the sound. If you do not place a POKE statement such as these in your sound routines, the sound will continue until you press RUN/STOP-RESTORE.

Another thing that may help when you are trying sounds on your VIC is to place this line at the beginning of your program:

```
POKE 36879,8:PRINT "{CLR}"
```

This both darkens the screen and clears it before any sounds are produced. Sometimes the VIC makes static noise when the screen is filled with printing in a bright display.

Writing a Sound Program

Although you may have experimented with the sample routines, or others you found, it would be useful to work on a program in more detail:

Program 7-3. Keysound

```
10 POKE 36879,8:PRINT "{CLR}"
20 POKE 36878,15:HI=36876
30 GET A$:IF A$="" THEN 30
40 FOR Y=128 TO 255
50 POKE HI,Y:NEXT
60 POKE HI,0
70 GOTO 20
```

Program Explanation

Line	Function
10	Create a dark, cleared screen

- 20 Memory location 36878 sets the volume. Maximum volume is 15. HI is established as equalling 36876 (high tone register) so it won't have to be typed in each time it is used.
- 30 Wait until the user presses a key.
- 40-50 Play each tone of the high sound register, from 128 to 255.
- 60 Turn off sound register HI.

With this one simple program, you can quickly see several different applications and create varying effects, just by altering a line or two. For example, by changing line 50 to:

```
50 POKE HI, Y:FOR T=0 TO 1:NEXT: NEXT Y
```

you will create a slight delay after each of the sound increments. Changing it to read T = 0 TO 1000 will separate the notes even more. Can you think of an application in a game (preferably your own)?

Change line 50 back to its original state and then alter line 40 to read:

```
40 FOR Y=128 TO 255 STEP 2
```

This would use the original sound, but it would be shorter by half.

Making the pitch drop from highest to lowest is also quite simple. To do this, change line 40 to:

```
40 FOR Y=255 TO 128 STEP -1
```

An even more elaborate sound can be created with this program by changing and adding these lines:

Program 7-4. Keysound UFO

```
40 FOR Y=128 TO 255 STEP 5
50 POKE HI, Y:NEXT
60 FOR Y=255 TO 128 STEP -5
70 POKE HI, Y:NEXT
80 GOTO 40
```

This new program will keep running until you hit the RUN/STOP key. At that point you'll hear another tone which you can stop only by pressing the RUN/STOP key along with the RESTORE key. This resets the volume value to zero.

The sound made by this program is a type of noise often made by games using spacecraft, such as a UFO. To get even

closer to such a sound, change both 128 values to 240. A good *Pac-Man* imitation can be obtained by changing the value in line 40 from 128 to 210.

You'll notice that most of the changes made were in the lines containing the FOR-NEXT loops. These loops are essential in most sound effects, and you should be familiar with this command's operation before you attempt to write any detailed sound effects routines. Once you are comfortable with the format of a sound routine, you should be able to create almost any sound you want.

You now have the background needed to create sound effects. What do you do with them? Place them in a game program, *your* game program.

Background Sound

One possible use for sound or music in a game program is as background, which can be useful as a mood producer. Remember the importance of sound in affecting the mood of a player during a game. It is important, and your game will certainly be better for its inclusion. Although the music you choose for your game will probably be different from the following example, once you've looked through this section, you should be able to apply its techniques.

If you want this sound to be continuous throughout your game, the routine will have to be included as part of the main loop of the game program. If you want a high-pitched drone that alternates between two pitches during the game, for example, you might set up something like this:

Program 7-5. Background Sound

```
10 POKE 36879,8:PRINT "{CLR}":POKE 36878,15
20 HI=36876:A=248:B=250
30 POKE HI,A:GOSUB 100
40 POKE HI,B:GOSUB 100
50 GOTO 30
100 FOR T=0 TO 300:NEXT:RETURN
```

The *A* and *B* values in line 20 represent the two alternating pitches, while line 100 slows the routine. Adding a delay to the main loop of the program simply to slow the sound would be wasteful, since it would also slow down the game. Inserting a necessary delay, then, is better done in the sound routine.

This routine has yet to be added to a main loop of a program, however. If this background sound of alternating pitches is to be used, there must also be a way for the pitches to change. A simple way to accomplish this is to alter the pitch each time the main loop of the program executes. By adding this sound routine to the main loop of the "Mission: Nova!" game program from Chapter 6, you can hear a constant background sound as you play.

Program 7-6. Main Loop Background Sound

```
101 POKE 36878,10
102 S1=S1=0
103 IF S1<>0 THEN POKE 36874,170:GOTO 110
104 POKE 36874,190
```

The main loop of the Mission: Nova! program begins on line 100. Line 101 POKEs in the volume setting for your background sound. Line 102 determines if S1 is either -1 or 0. The first time through the main loop, S1 is 0, so the POKE command in line 104 is executed. The second time through the main loop, S1 is -1, so the POKE in line 103 executes. Each time it alternates, creating the two sounds. Notice that there is no delay loop to slow the sound down. Add a delay, such as FOR T1 = 0 TO 300:NEXT as new line 105. The game slows down, doesn't it? In fact, it slows down far too much, so the line should be removed.

Using a format such as this in another game, perhaps your own, you could make the sound correspond to the movements of the characters. Because game characters can make location changes each time through the main loop, the pitch would change with every character movement. This is difficult to see in the Mission: Nova! program, but this background sound technique can be quite useful, especially in a short game which has a small number of characters moving on the screen.

Sound Subroutines

Many sounds in a game will not occur throughout the entire game program, but will instead be heard only at the time of a certain event. The music which plays when a game character is lost, for example, would fit in this category. Perhaps you want sound when the fire button on the joystick is pressed, or a loud tone from the noise register when a target is hit. In these situations, the sounds are called upon as subroutines.

Unfortunately, when a program goes to a subroutine, it leaves

the main loop and everything comes to a stop until the program returns. Sometimes this is very noticeable, for you can see the characters on the screen stop as a noise or sound is executed by the subroutine. Although this is a drawback, it is often used because of the complexity of incorporating a sound routine into the main program loop. Many programmers use the subroutine technique since it is relatively simple to program. Perhaps in some games the visual delay will not be as noticeable. As the game designer and programmer, you must make the decision. Will the delay harm the overall game execution? Will it lessen the player's enjoyment of the game? If you think not, try the subroutine technique.

The following example program uses a subroutine to access the sound only when it is called for. A character moves back and forth on the screen, which also displays a time clock. Every five seconds the sound subroutine is called, and you will see the character stop as the routine is performed.

Program 7-7. Subroutine Sound

```
10 POKE 36878,15:POKE 36879,8
20 L=7680:H=83:TI$="000000"
30 D=1:S=32:X=36876
40 PRINT "{CLR}"
50 POKE L,H:PRINT "{HOME}{5 DOWN}"TI$:REM
   BEGIN MAIN LOOP
60 IF TI$="000005" THEN G=1
70 IF TI$="000010" THEN G=1
80 IF TI$="000015" THEN G=1
90 IF TI$="000020" THEN GOSUB 150:GOTO 20
   0
100 ON G GOSUB 150
110 IF D=1 THEN L=L+1:POKE L-1,S
120 IF D=-1 THEN L=L-1:POKEL+1,S
130 IF L=7701 THEN D=-1
140 IF L<7680 THEN D=1
145 GOTO 50:REM END OF MAIN LOOP
150 FOR F=255 TO 127 STEP -1:POKE X,F:NEXT
   T
160 G=0:RETURN
200 PRINT "{CLR}{5 DOWN} THE DEMO IS OVER
   ..."
```

Program Explanation

Line	Function
10	Volume control is set to maximum, and the screen is

- darkened. Again, this makes the character POKEd in more visible and reduces the monitor noise interference.
- 20 Establish the variables. L is the location of the character; H is the POKE number for the "heart" character. The time clock is set at 0.
- 30 Set the variables for the character's direction. When $D = 1$ it will move from left to right as per line 110. When $D = -1$ it will move from right to left as per line 120.
- 40-50 First the screen is cleared, and then the main loop begins by POKEing in the character at its first location and printing the time clock five lines below so that it is out of the way.
- 60-80 The clock is checked and compared with the values of 5, 10, and 15. If it is one of these values, then G is assigned the value of 1.
- 90 The clock is checked to see if it is to the 000020 mark yet. If it is, the sound routine is played and the program will GOTO line 200, where it will end.
- 100 The ON G GOSUB command means, in effect, that if G does not equal 0, the program will GOSUB 150, the beginning of the sound routine. This is actually a shortcut in programming, for if this line was not included, lines 60-80 would each need a THEN GOSUB 150.
- 130-140 Change the direction of the moving character when it reaches the end of the row. If this was deleted, the character would keep moving off the screen.
- 145-160 Line 145 is the end of the program's main loop. It also returns the program to line 50 for a repeat of the program. Lines 150 and 160 are the sound subroutine. Notice that the value of F goes below 128, which automatically turns off the sound at the end of the loop. Line 160 also resets the value of G to 0 so the sound will not keep repeating. Make sure the RETURN is included at the end of every subroutine, so it will go back to the main loop of the program.

You could easily insert other conditional statements to make sounds at any number of events. One example would be creating a sound each time the character reached the end of a line and changed direction. All you have to do is eliminate lines 60-80 and line 100 and add :GOSUB 150 to the end of both lines 130 and 140. It should look like this:

Program 7-8. Subroutine Sound #2

```
10 POKE 36878,15:POKE 36879,8
20 L=7680:H=83:TI$="000000"
30 D=1:S=32:X=36876
40 PRINT "{CLR}"
50 POKE L,H:PRINT "{HOME}{5 DOWN}"TI$:REM
   BEGIN MAIN LOOP
90 IF TI$="000020" THEN GOSUB 150:GOTO 20
   0
110 IF D=1 THEN L=L+1:POKE L-1,S
120 IF D=-1 THEN L=L-1:POKE L+1,S
130 IF L=7701 THEN D=-1:GOSUB 150
140 IF L<7680 THEN D=1:GOSUB 150
145 GOTO 50:REM END OF MAIN LOOP
150 FOR F=255 TO 127 STEP -1:POKE X,F:NEX
   T
160 G=0:RETURN
200 PRINT "{CLR}{5 DOWN} THE DEMO IS OVER
   ..."
```

Eliminating lines 60-80 stops the sound subroutine from executing every five seconds, and instead executes it only with the GOSUB commands in lines 130 and 140. The sound still occurs at the end of the program, when the clock reads 000020.

Experiment with the program to see what kind of changes you can make to add more sounds. Spend some time with your own game design, and decide if this kind of sound subroutine will work in your game program.

No-stop Subroutines

Although adding sound to your game by using subroutines does stop the action for a short moment, it is a simple way to program. Some games, however, would benefit from having sound which does *not* cause the game action to stop. It is more difficult to add sound using this technique, but it is often preferable to using subroutines which halt a character every time they are executed. Remember that each time a routine is added into a program's main loop, the game action will slow down. Sometimes this will have little or no effect on the game's appearance and play. Other times it will be noticeable. Again, as game designer and programmer, you must make the decision.

The example program which follows uses this subroutine technique. The sound executes without stopping the game action, and in fact is heard only when the two characters on the

screen come in contact. One character moves randomly about the screen, while the other is user-controlled through the keyboard. The @ key moves the character up, the : key moves it to the left, the = to the right, and the space bar moves it down.

Program 7-9. Sound Game

```

10 POKE 36878,15:POKE 36879,8
20 L=7680:M=7900:H=83:S=32:P=42:X=36876:N
   =125
30 PRINT "{CLR}"
40 POKE L,H:POKE M,P:N=N-5:POKE X,N:REM B
   EGIN MAIN LOOP
50 IF PEEK(197)=53 THEN M=M-22:POKE M+22,
   S
60 IF PEEK(197)=46 THEN M=M+1:POKE M-1,S
70 IF PEEK(197)=45 THEN M=M-1:POKE M+1,S
80 IF PEEK(197)=32 THEN M=M+22:POKE M-22,
   S
90 IF M>8185 THEN M=8185
100 IF M<7680 THEN M=7680
110 R=INT(RND(1)*4)
120 POKE L,S
130 IF R=0 THEN L=L-22
140 IF R=1 THEN L=L+22
150 IF R=2 THEN L=L-1
160 IF R=3 THEN L=L+1
170 IF L>8185 THEN L=8185
180 IF L<7680 THEN L=7680
190 IF L=M THEN N=255
200 IF N<1 THEN N=125
300 GOTO 40:REM END MAIN LOOP

```

Program Explanation

Line	Function
10	This program starts out much like the other examples in this chapter. The screen is changed, and the volume is set to the maximum level of 15.
20	Variables are set as follows: L = starting location of randomly moving character M = starting location of user-controlled character H = POKE value for the random character S = POKE value for a character space to erase any previous images of moving characters P = POKE value for user-controlled character X = sound register value N = tone value of the sound register. Note that this starts

7 Sounds and Music

- as less than 128 so that it cannot be heard. When you want the sound to execute, you need only assign a value higher than 128, which is done in line 190.
- 30-40 Clear the screen and POKE in the two characters at their starting locations. The value of N is also reduced by five. This is necessary in the main loop when the sound is called for. Although the value of N decreases by five each time through the main loop, note that line 200 reestablishes N as 125 when its value falls below 1. This prevents the VIC from trying to POKE a negative number into location 36876. The sound register is then set to tone N . Again, you won't hear this until the value is greater than 128, so it will be silent at the beginning of the program.
- 50-80 The keyboard is checked and the character moved accordingly. The last portion of each line erases the previous image.
- 90-100 Keep the user-controlled character on the screen.
- 110-170 Establish a value between 0 and 3 for R . The random character moves according to which number was chosen. This is a simple way to create random movement. Line 120 erases the random character just before it moves. The timing for POKEing in characters and erasing them has an impact on how smoothly the characters move. Lines 170 and 180 keep the random character from leaving the screen.
- 190 The sound executes when the two characters meet; in other words, when $L = M$. The value for N is then set to 255, the highest possible value. You will hear a descending sound; each time the main loop is executed, line 40 reduces N by five.
- 300 End the main loop and send the program back to line 40.

As with the other sound routines, it would be just as easy to set up a modulating tone, or to insert a clock so that the sound plays for a specified number of seconds before stopping. There are a variety of sounds you can create and applications to use them in.

Mission: Nova! Sounds

Using these techniques for sound subroutines, you can add additional effects to the game program Mission: Nova! You

already have a background sound in the program's main loop, and now you want to hear something when objects meet on the screen. Not only will this add to the enjoyment of the game, but it will also provide feedback as you play. You'll know certain things are happening without having to visually check them every moment.

To add sound subroutines to Mission: Nova!, you first have to locate the lines which test for contact between objects on the screen. This is done in the section of lines 700-730.

Line 700 tests to see if the starship is in contact with a small star. If it is, your score decreases and the ship changes color.

Line 710 tests to see if the ship is on the large space station. Remember that your ship is then renewed and brought back up to full power.

Line 720 tests to find out if the spacecraft is on the small space station. Nothing is awarded for this in the game.

Line 730 tests for the only remaining possibility, that the spacecraft is in contact with a large, nova star. Points are given for this.

You'll want four separate sound effects, then, for these four possibilities. Each sound should somehow indicate a positive or negative result during the game, providing player feedback. Obviously, then, four sound subroutines are necessary.

All four lines must include a GOSUB command so that the VIC refers to the proper sound subroutine. They could look like this:

```
700 IF W=46 THEN P=P-100:NC=NC+1:GOSUB 11
    00:RETURN
710 IF W=102 THEN NC=CS:W=104:GOSUB 1200:
    RETURN
720 IF W=104 THEN GOSUB 1250:RETURN
730 P=P+(8*NH)*INT(QQ-Q):W=46:GOSUB 1150:
    RETURN
```

Note that each GOSUB is placed *before* the RETURN command.

Now the separate sound subroutines can be written and placed at the end of the program.

Program 7-10. Small Star Collision

```
1100 POKE 36878,10
1110 POKE 36877,200
1120 FOR ET=1 TO 100:NEXT
1130 POKE 36878,0:POKE 36877,0:RETURN
```

Program 7-10 creates a short buzz or zap sound to indicate that points have been lost and your starship has been damaged. The noise register is used and a short delay is added so that the sound can be heard. After that, both the volume and noise registers are turned off, so they won't interfere with other sound effects later. Notice the RETURN command at the end of line 1130. It must be included to send the program back to line 700.

Program 7-11. Meeting the Large Space Station

```
1200 POKE 36878,10
1210 S1=36876
1220 FOR SY=128 TO 255
1230 POKE S1,SY:NEXT
1240 POKE 35878,0:POKE S1,0:RETURN
```

As the spaceship meets a large space station, this subroutine plays a quick scale, using the high tone register. Because it rises from the lowest to the highest tone, it sounds like something filling up, which is what the station does for the starship, returning it to its original power level.

Again, the volume level is set, values for the high tone register and its tones established, and then POKEd in. Finally, the volume and high tone registers are turned off, and the subroutine RETURNS to line 710.

Program 7-12. Landing on the Small Space Station

```
1250 POKE 36878,10
1260 FOR SMT=1 TO 5:NEXT
1270 FOR SMY=250 TO 240
1280 POKE 36876,SMY:NEXT
1290 POKE 36876,0
1295 POKE 36878,0:RETURN
```

Since there is no effect when the spacecraft touches the small space station, this blip sound will work well. However, a sound is useful, if only to tell the player that the small station does not give the spacecraft an increase in power, as the large station does.

Program 7-12 is very similar to the Blip routine used earlier in the chapter. The volume is set, as are variables for the short delay and the sliding tones. Then the high tone and volume registers are turned off, and the subroutine RETURNS to line 720.

Program 7-13. Meeting a Nova Star

```
1150 POKE 36878,10
1160 LS=36876
```

```
1170 FOR LY=128 TO 255 STEP 2  
1180 POKE LS,LY:NEXT  
1190 POKE 36878,0:POKE LS,0:RETURN
```

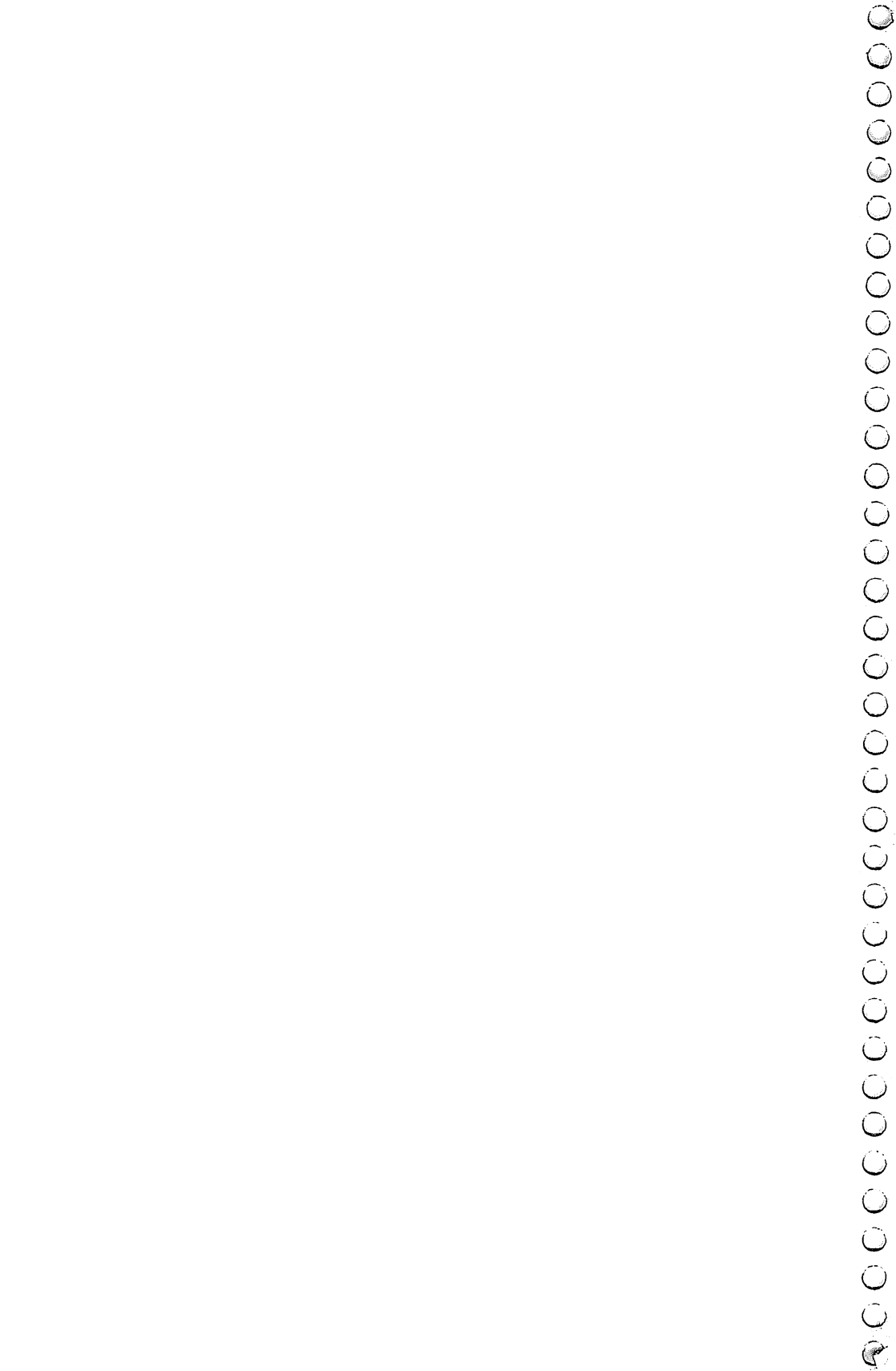
Program 7-13 creates a sound almost identical to the one used when the spaceship lands on the large space station. The major difference is that in line 1170, the STEP 2 command is included. This simply shortens the sound to half of what it was in the large space station subroutine. Again, the sound is one of filling up, because points are awarded. Making it only half as long fits in well with the game, for it is less beneficial to the player. Points are awarded, true, but the ship remains at the same power level.

Adding these subroutines to the game program does increase the player's enjoyment. Note the difference in the game after you've added these routines. It should be more exciting, more entertaining, and easier to play with sound.

Make Use of Sound

As you've seen, sound can make a program more complex. It also complicates your decision-making process as you determine what kind of game you want to create and what sound effects you want to use. Yet it is worth the time and the effort to include sound, simply for the impact it has on a game. A game without sound is simply not a game to most players. Something vital is missing.

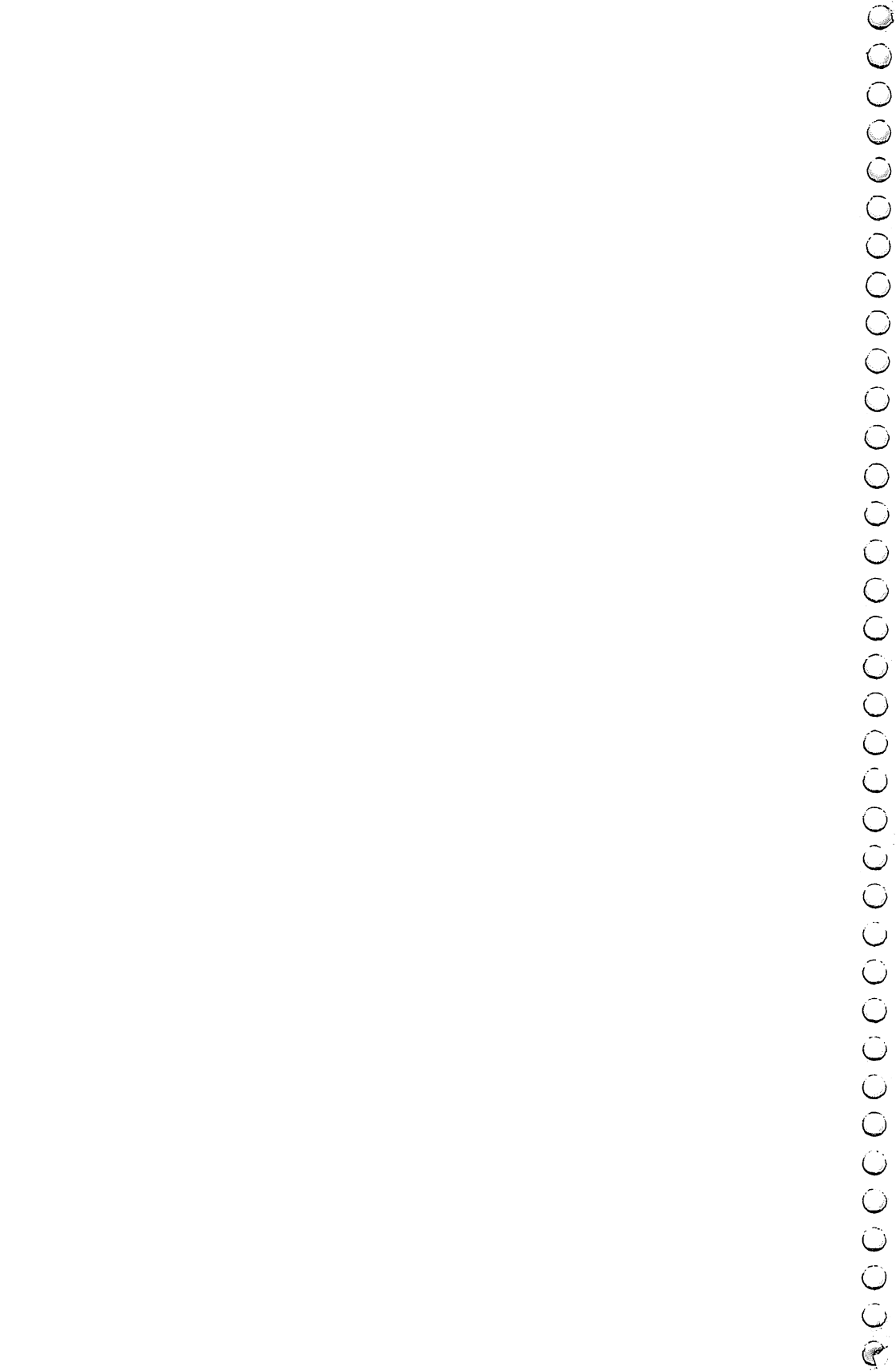
You have several techniques available to you as you plan the sound for your game. Background sound or music can be used, or a subroutine can be inserted into the main program. Action can stop for a moment when sound is played, or it can continue, depending on the complexity of the program.





8

**Introductions,
Instructions,
and Farewells**



8 Introductions, Instructions, and Farewells

Introductions, instructions, and farewells add an element that every player appreciates, another dimension to the game world you've created. Just like sound effects or custom characters, they are not *absolutely* necessary, but without them your game will seem pale and one-dimensional. Introductions and farewells make the game more interesting, entertaining a player while a screen display is set up, or providing incentive to play again for a higher score at game's end. Without instructions, a player could be confused or frustrated at the outset while trying to discover cursor directions, scoring procedures, or the goals of the game.

Even though introductions and instructions appear at the opening of a game, this part of game design is best approached *after* your program has been written. It is only then that you know the amount of remaining memory available for these extras. Through the constant changing and revising of your program, it is probably not until this point that you know exactly how your game will appear. Of course, there are exceptions, for you may have your design well-outlined before you begin to program.

Memory

After your program is completed, and before the introductions and instructions are written, you have to find out how much memory is still available. This is important with the unexpanded VIC, for its 3583 available bytes can be quickly used in a program with a custom character set, sound effects, and animation. It is little use to create a complex introduction, with involved music and graphics, if it will not run because of memory limitations. At times you'll be forced to use every available byte, for you may have only several hundred bytes left after the program. You need an accurate idea of what you have to work with.

After you have your program written and loaded, use the

command PRINT FRE(0) to find out how much memory is available. If you have a small amount to work with, use the command frequently, checking as you get closer and closer to the end of memory. It could save you considerable anguish.

Introductions

Introductions can take many forms and shapes. Perhaps you want another sound effect, or a short piece of music, to entertain the players while they wait for the game screen to set up. If the memory is available, a brief description of the game situation can be inserted. This can heighten the player's involvement in the world you've created. At the very least, you'll want to display the title before the game begins.

Since you've already written your program, you probably won't be able to place the introduction at the beginning without renumbering the entire program. A simple way to solve this problem is to place the introduction in a subroutine that is called by the first line of the program. In the sample game from Chapter 6, *Mission: Nova!*, it would look like this:

```
5 GOSUB 1300
```

The program would then move to the subroutine and execute the introduction, returning to the opening of the program. The following example subroutine can be inserted into the *Mission: Nova!* program to serve as an introduction.

Program 8-1. Introduction

```
1300 PRINT "{CLR}":POKE 36879,122
1305 PRINT "{2 DOWN}{4 RIGHT}{BLK}MISSION
      : NOVA!"
1310 PRINT "{2 DOWN}{RIGHT}{BLU}MANEUVER
      {SPACE}YOUR STAR"
1315 PRINT "{RIGHT}SHIP ACROSS THE"
1320 PRINT "{RIGHT}GALAXY."
1325 PRINT "{2 DOWN}{RIGHT}AVOID THE STAR
      S AND"
1330 PRINT "{RIGHT}REFUEL AT NOVAS."
1335 PRINT "{2 DOWN}{RIGHT}LAND AT THE SP
      ACE"
1340 PRINT "{RIGHT}STATION FOR FULL"
1345 PRINT "{RIGHT}POWER."
1350 PRINT "{4 DOWN}{RIGHT}HIT ANY KEY"
1355 GET A$:IF A$="" THEN 1355
```

As the program runs, notice that the screen changes color. This is a nice effect achieved simply by POKEing a different value into location 36879. The only thing to be careful of, however, is that the screen will still display your words. If the screen background color is the same as the character color, the words will seem invisible.

It is also simple to change the character color of the title, as is done in the above sample subroutine in line 1305. Again, it may be difficult to see the different color unless you've chosen one considerably different from the background. Experimenting with several variations takes only a little time and cannot harm your program. To turn the character color back to the original, simply insert the proper keystrokes within the next PRINT statement. In line 1310 this is done by hitting the [control] and [blue] keys at the same time.

Since the VIC's screen is relatively small, only 23 lines by 22 columns, it's a good idea to double-space between each line, or between short paragraphs, as is done in the sample subroutine. This makes it easier to read. Notice, too, that the lines are justified on the left. This can be done several ways. First of all, you can just print the lines from the leftmost column. Sometimes, if you have short lines, you will want to indent. You can use the cursor commands within the quotation marks, or you can use the TAB command to indent every line the same distance. The statement

PRINT TAB(1)"MANEUVER YOUR STAR"

would print the line one column from the left, in the second column. You could replace line 1310 in the sample subroutine with this line, for instance.

Although you have only one screen so far, often you'll use a lengthy introduction, with several screens of print. Instead of scrolling the introduction or printing it one line at a time at a slow speed, it may be easier to display each screen full of print separately. At the end of each screen, as in Program 8-1, you could add the line:

PRINT "HIT ANY KEY"

followed by:

GET A\$:IF A\$="" THEN XXX

where XXX is the line number of the GET A\$ command.

This will hold the screen until the player presses a key, then

will display the next screen. At the end of the PRINT statements for the last screen, be sure to include a RETURN command.

Our sample subroutine takes up only one screen so far. If that is all you're using, to return to the game program you need only add a line such as this:

```
1346 FOR T1 =0 TO 5000:NEXT:RETURN
```

and eliminate lines 1350 and 1355.

The delay is to give the player enough time to read the introduction. If your introduction is longer or shorter, you can alter the delay loop accordingly.

Adding sound is quite easy. Before the program RETURNS, insert the POKE statements for whatever effect you want. The sound will then play before the game begins and while the introduction is still on the screen. You can include several effects if memory is still available.

Instructions

Depending on the memory still left, you may be restricted to using a short introduction in order to include instructions. Of the two, the instructions are more important to the player's understanding of the game. At the very least, you'll want to tell the player how to move user-controlled characters. If it's a joystick-controlled game, a note that a joystick is needed would be sufficient. If the keyboard is used to control the character, more explanation is necessary.

Program 8-2. Instructions

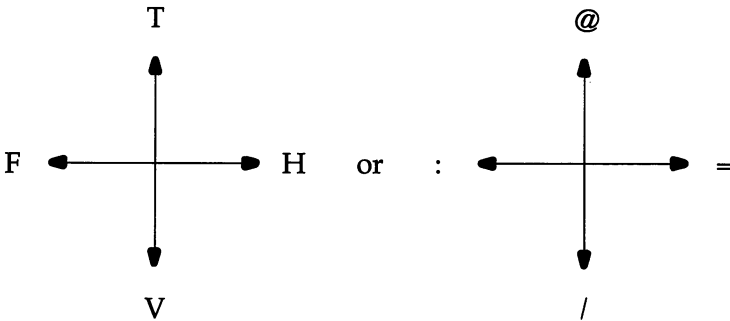
```
1360 PRINT "{CLR}{3 DOWN}{RIGHT}HIT F5 KE
Y FOR 'UP'"
1370 PRINT "{2 DOWN}{RIGHT}HIT F7 KEY FOR
'DOWN'"
1380 PRINT "{2 DOWN}{RIGHT}HIT COMMODORE
{SPACE}KEY"
1390 PRINT "{RIGHT}FOR 'LEFT'"
1400 PRINT "{2 DOWN}{RIGHT}HIT SHIFT KEY
{SPACE}FOR"
1410 PRINT "{RIGHT}'RIGHT'"
1415 PRINT "{3 DOWN}{RIGHT}HIT ANY KEY"
1420 GET A$:IF A$="" THEN 1420
```

If the keys are not next to each other, this may be the best way to tell the player which keys move the character each direction. Note that the GET A\$ statement is used again to hold the screen

until the player is ready to continue.

Your game may use keys that are placed next to each other on the keyboard. For example, your game may use a diamond-shaped pattern for character control. If that's the case, then you may want to display something like Figure 8-1 on the screen, showing the corresponding directions and making it easier for the player to understand.

Figure 8-1. Keyboard Display



The TAB command makes it easy to set up a screen like this by positioning the lines of explanation in the right place.

Scoring is another item you may want to include in the instructions. Every player wants to know how points are scored, by what criteria, and toward what goal. These instructions do not have to be lengthy, as the following routine for Mission: Nova! shows:

Program 8-3. More Instructions

```

1430 PRINT "{CLR}{2 DOWN}{RIGHT}SCORE POI
      NTS BY"
1440 PRINT "{RIGHT}REFUELING AT NOVAS"
1445 PRINT "{2 DOWN}{RIGHT}POINTS DEDUCTE
      D FOR"
1450 PRINT "{RIGHT}CRASHING INTO STARS"
1455 PRINT "{RIGHT}AND TAKING YOUR TIME"
1460 PRINT "{2 DOWN}{RIGHT}YOU CAN CRASH
      {SPACE}INTO"
1465 PRINT "{RIGHT}ONLY SIX STARS"
1470 PRINT "{RIGHT}BEFORE LANDING ON"
1475 PRINT "{RIGHT}THE SPACE STATION."
1480 PRINT "{2 DOWN}{RIGHT}THE GAME ENDS"
1485 PRINT "{RIGHT}WHEN YOUR SHIP'S"
  
```

8 Introductions, Instructions, and Farewells

```
1490 PRINT "{RIGHT}POWER IS GONE."  
1495 PRINT "{3 DOWN}{RIGHT}HIT KEY TO STA  
RT."  
1496 GET A$:IF A$="" THEN GOTO 1496  
1497 RETURN
```

If your game is quite complex, more than one screenful of print may have to be seen by the player. Again, the GET A\$ statement lets the player read at his or her own pace, then move on. Note that at the end of this routine, in line 1497, the RETURN statement finally appears, sending the program to begin the game's screen setup.

Setting Difficulty

At times, you'll want to allow the player to set the level of difficulty in the game. If your game lends itself to this, a player can begin at the easiest level and progress to the more difficult. A game such as *Tempest*, which lets the player choose the starting level, is a good example. It will seem that there are actually several games in one if the levels are considerably different. An ideal place to do this is in the instructions, before the game begins. The INPUT or INPUT #1 commands can be used to do this.

Again, a subroutine called at the program's outset is one way to do this. The game "Spark" from Chapter 6 is one in which levels could be set using the statement:

```
5 CLR:GOSUB 800
```

and the following subroutine:

Program 8-4. Spark Levels

```
800 PRINT "{CLR}":POKE 36879,122  
810 PRINT "{3 DOWN}{RIGHT}{BLU}CHOOSE LEV  
EL OF PLAY"  
820 PRINT "{RIGHT}BY HITTING KEY:"  
830 PRINT "{2 DOWN}{RIGHT}1"TAB(8)"EASIES  
T"  
840 PRINT "{2 DOWN}{RIGHT}2"TAB(8)"HARDER  
"  
850 PRINT "{2 DOWN}{RIGHT}3"TAB(8)"DIFFIC  
ULT":PRINT:PRINT:PRINT  
860 OPEN 1,0  
870 INPUT#1,A$:CLOSE 1,0  
880 IF A$="1" THEN P=300:RETURN  
890 IF A$="2" THEN P=200:RETURN  
900 IF A$="3" THEN P=100:RETURN
```

Line	Function
800	Clear the screen and POKE in a screen color change.
810-850	Print the instructions for setting the different levels of play. Note the [control] [blue] keystroke in line 810. This sets the character color back to blue. The program will alter the character color after it has run once if this is omitted.
860	Used to create the INPUT #1 command in the next line so that a ? prompt will not appear on the screen.
870	The INPUT #1 command delays the program until the player provides a response. This command is very useful when asking for information from the player. The CLOSE #1,0 statement completes the instruction begun in line 860.
880-900	Test for the number pressed by the player, and then set the value of P accordingly. P is the time remaining before the game ends. In other words, the most difficult level begins with less time for the player to complete the game. Notice that each line has a RETURN statement at its end. The program then moves back to line 5, and the game program begins setting up its screen.

As you program more involved games, you'll find uses for this method of setting levels of difficulty. Even if you are now designing a game that does not lend itself to this, keep it in mind for later.

Farewells

Assuming that memory is available, the final addition to your game should be a farewell, or end routine. As with the introductions and instructions, this can be as elaborate or simple as you want. Some farewells are strictly entertaining, nudging the player back into the game, persuading the player to try it again, perhaps just to see the end routine. Other farewells are more informative, giving the final score, showing a high score, ranking the player, or asking if another game is wanted. Just as with the introductions and instructions, you can insert this as a subroutine.

A simple farewell, which records the final score, prints a short message, and allows the player to begin a new game, would take only a few lines in a program. In the example game Spark, the score was assigned the variable P in the program. If the player

8 Introductions, Instructions, and Farewells

won, the user-controlled character exited the maze, and the game was over. Running out of time caused P to equal 0, and so ended the game. There were only two possible ways to end. Placing a GOTO command at both program lines could be done this way:

```
170 IF CHG < 4 THEN GOTO 610
```

and

```
330 IF P = 0 THEN GOTO 600
```

When the character exits the maze, then, the program shifts to line 610. If the player loses the game by running out of time, the program goes to line 600. At that point, the following routine could be used:

Program 8-5. Spark End

```
600 PRINT:PRINT "TOO LATE!{2 SPACES}SCORE  
IS 0":GOTO 620  
610 PRINT:IF P>0 THEN PRINT "YOU MADE IT!  
{2 SPACES}SCORE IS":PRINT P  
620 PRINT "{2 DOWN}TO PLAY AGAIN, HIT ANY  
KEY."  
625 POKE 198,0  
630 GET A$:IF A$="" THEN 630  
640 GOTO 5
```

Line	Function
600	Print the message that the player ran out of time and scored 0 points. The first PRINT command is used to force the message to print below the maze. Program then shifts to line 620.
610	If P > 0, then the player has won and this message is printed on the screen, along with the final score.
620	Drop two lines to display the message allowing the player to try again.
625	This POKE is necessary to clear the keyboard buffer of any characters typed in during the preceding game. If this is omitted, the next line will execute immediately, for the VIC will read the buffer and assume a key was pressed for line 630.
630	GET A\$ simply waits for a key to be pressed by the player before continuing.
640	Return the program to the beginning. Notice that the player is returned to the level-setting routine, so that the level of difficulty can be changed.

A more complex end routine could include sound, as well as display a final score and ask the player whether another game is wanted. Inserting sound into an end routine can be difficult at times, depending on how your game program is organized and where you want the sound to play. One way to create sound in an end routine is by using READ and DATA statements. The game Mission: Nova! could use this technique, and the routine could appear as:

Program 8-6. Nova! End

```

900 POKE 36878,15:RESTORE
910 READ N1,D1:X=7+7*(X=7):POKE CM+C,X:IF
    N1=0 THEN 930
920 POKE 36874,N1:FOR EI=1 TO 150*D1:NEXT
    :POKE 36874,0:FOR EI=1 TO 20:GOTO 910
930 PRINT "{CLR}"P" POINTS":PRINT:PRINT"P
    RES ANY KEY TO{2 SPACES}PLAYAGAIN"
940 FOR I=0 TO 3000:A=PEEK(197):IF A<>64
    {SPACE}THEN:GOSUB 450:GOTO 280
945 NEXT I:END
950 DATA 191,4,191,3,191,1,191,4,201,3,19
    9,1,199,3,191,1,191,3,188,1,191,4,0,0
  
```

Line	Function
900	Set the volume register to maximum.
910	The values in the READ statement are set and the space-ship is flashed by the POKE CM + C,X. IF N1 = 0 (the sound register is then turned off), the music ends and the program shifts to line 930.
920	POKEs the high tone register with the values in the DATA statement in line 950, and holds the tones at varying lengths (EI = 1 TO 150*D1). The program then moves back to line 910 to READ the next values in the DATA statement.
930	Print final score and ask the player whether another game is to be played.
940	The delay allows the player time to decide if he or she wants to play another game. If so, the PEEK command reads the keyboard and sends the program to the appropriate lines. If another game is <i>not</i> wanted, the program will end in line 945 after the delay.
950	The DATA statement creates the dirge music which plays at the end of a game.

Design Notes

Programming introductions, instructions, and farewells can be challenging and exciting, for each addition to your game will make it that much more professional in appearance, as well as more playable and entertaining. As long as the memory is available, use it to your advantage to enhance your game. Although we've considered a number of different concepts to include in your game introductions, instructions and farewells, there are a few more suggestions for your use.

Be personal. If the player is asked to INPUT his or her name and then sees it printed in the instructions, or even beside the user-controlled character, the player will be drawn deeper into the game's world. Within the farewell, the player's name could also be printed alongside the final score. These human touches add to any game, as you've probably noticed when you've played video arcade games.

Make use of the color abilities of the VIC. Changing screen colors certainly adds to the game, but reversed or colored characters will add even more, especially in the introduction and instructions. With memory available, you can alter the screen and character colors with every new screen display.

Urge the player to play another game by including a friendly ending. It doesn't have to be something cute, just enough to make another game worth the time. Proclaim the player's victory, hand out promotions or rankings. Your imagination as the game designer can come into play here. Creativity is what makes one game stand above all others.

Your game program is *almost* completed. You've gone from the idea to writing the end routine. There are a few more things to consider, however, in the next chapters.



9
**The Shape
of the Game**



9

The Shape of the Game

We've gone through most of the techniques you'll need to create your own games. Now all you really need to do is hack through the actual programming. The word "hack" is pretty accurate for most of our programming. No matter how logical you are, chances are pretty good that the computer is even more logical, and you'll write many a routine that *should* work — but doesn't. At least *I* certainly program that way sometimes, like a clumsy woodcutter taking one hack after another until finally he splits the log.

And it works. Eventually, even the most stubborn programming problem will yield to your constant work.

There are ways you can make things easier for yourself, and in the meantime make your programs run as smoothly and quickly as possible. However, the programming techniques in this chapter are not "rules" — they're just advice. Good advice, I think, that has helped me in my programming, but remember: any game that plays well is a good game, even if the programming isn't pretty. So don't be concerned if you fudge a bit here and there. If it works, it's correct.

Program Structure

In most of the programs in this book, I've tried to structure my code with three purposes in mind:

1. Make it easy to understand.
2. Make it run fast.
3. Use up the least possible memory.

Ease of Understanding

Why should you make your programs easy to follow? After all, you're writing a game for yourself, not example programs in a book. Nobody's ever going to see your programming, just the results.

There's one exception. You.

You will look at your program again and again. Sometimes

there'll be days, weeks, or months between programming sessions. You'll look at your own program then and wonder what in the world is going on. You'll forget where the value of obscure variables was set. You'll forget why line 55 *had* to come before GOSUB 590. You'll forget that there are three different ways to get into the routine at 400.

That's inevitable. It happens to everybody. And you can't prevent it entirely. What you *can* do, though, is follow several simple programming rules.

Establish patterns and habits. Start routines on even hundred lines — 100, 200, 300, 400, and so on. That way you'll be able to look at an entire routine just by typing LIST 300-399. Sometimes I bend this rule by starting a very short routine at an even fifty — 450 or 550. And in this book I have sometimes put at line 990 a single-line loop that is used many, many times throughout a program. For instance, a loop that reads the keyboard and waits for the player to press a key.

The important thing is to have *habits*. For instance, in this book you always know that if I have a GOSUB or GOTO 100, 200, 300, 400, and so on, the program is going to a main routine. You can quickly scan through each routine, see what each one does, and then know instantly which routine is being called at each GOSUB and GOTO. You know that if a GOSUB or GOTO ends in 50, it's a small routine. A GOSUB or GOTO a line ending in 90 is a one-line routine called many, many times.

Of course, you may prefer to develop different programming customs. But whatever habit you follow, the more consistent you are, the more easily you'll be able to follow your own programs.

Put similar things together. If all your collision handling routines are located together, you won't have to spend half an hour poring over your program to find out where a particular collision is taken care of. You'll know that it's between 700 and 799. If all your DIM statements are in the first line of the program, then you'll never have to hunt around to find out why you're getting a BAD SUBSCRIPT error.

This principle extends even to tiny matters. If you're creating variables to control ten different onscreen figures, why not give their location in screen memory and in color memory and their color value and character code the *same* variable name? In "Moonraker" I used L for my player-figure's location and L1, L2, L3, L4, and so on for the computer-controlled figures. An even better practice might be to use arrays: L(*n*) is the address of the screen

location of each of the ten figures; $CL(n)$ is their address in color memory; $FG(n)$ is the character to be POKEd to the screen; and $C(n)$ is the color to be POKEd into color memory.

You can even use $M(n)$ to hold the number that should be added to $L(n)$ for each computer-controlled character's movement. The variable $S(n)$ can hold each figure's new starting address. And $E(n)$ can hold the ending address.

Think how easily and economically you can program all the movements then. Line 300 starts the loop and erases each figure at the old location in screen and color memory:

```
300 FOR I = 0 TO 9:POKE L(I),32:POKE CL(I),0C
```

Line 310 changes the locations:

```
310 L(I) = L(I) + M(I):CL(I) = CL(I) + M(I)
```

Line 320 checks for collisions with the player-figure and checks to see if the figure has reached the end of its journey:

```
320 ON -(PEEK(L(I)) = FG) - 2*(L(I) = E(I)) GOSUB 400,490
```

Line 330 actually carries out the movement by POKeing the figures onto the screen in the new locations:

```
330 POKE L(I),FG(I):POKE CL(I),C(I):NEXT
```

Vital parts of this are the subroutines at 400 and 490. Since we haven't created an entire program here, I won't try to create the collision subroutine, but the routine at 400 would need to change the score, decide whether the player-figure should win or lose the encounter, and assign new addresses, if necessary, to both the player-figure and the computer-controlled figure.

We can be more specific with the miniroutine at 490. Remember that the program only reaches this line if the figure's screen location matches its ending location. Therefore, all we need is:

```
490 CL(I) = CL(I) - (L(I) - S(I)):L(I) = S(I):RETURN
```

And that's it. By giving all your onscreen figures the same name, with subscripts, you can assign all their values in loops using DATA statements, and you can handle their movements and their collisions in a single routine. You have saved memory, you have probably saved running time, but above all you have kept the routine simple and easy to follow.

Label your work. Just because you're using a computer doesn't mean you can't write things down on paper. It's a good idea to keep a list of what each variable is used for in each program you work on. That way you won't accidentally use a variable

you've already got doing something else somewhere else in the program.

You can write down where each major subroutine is. You can write down key line numbers or areas where you plan to put subroutines.

Most important of all, however, is simply to label your tapes and disks, both externally and internally. There's nothing more frustrating than working for hours on a program only to realize that the version you're working on is *not* the most recent one, that all of the improvements from your last programming session are saved somewhere else. I've made it a habit to put the date at the beginning of my program. Some people put the date in REM statements, so you'll see it when you LIST the program. I put it right in a PRINT statement, so the date of this version will be flashed on the screen. That way I can keep track of which version of my game I am working on.

Programming for Speed

With game playing, it is not *real* speed that is important, but the *illusion* of speed. The game has to *feel* fast. A game with figures that only inch along the screen can feel fast to the player as long as his own player-figure responds quickly and moves fairly fast. This means that your program, to feel fast, should check for player input as often as possible, and should have as few things as possible happen between player-figure movements.

The main loop. The key is to keep the main loop as tight as possible. The fewer things the program has to do every time it goes through the loop, the better. The main loop is really there just to perform tests, to see what should happen next.

You keep the main loop tight by testing for minimal conditions and then jumping to subroutines to check the details. Many things can be kept out of the main loop entirely. For instance, collisions should almost never be checked in the main loop. Why? Because collisions can only happen when something moves into the same space as something else. Therefore, why check for collisions unless something has moved? Collision checking always belongs in movement loops.

Main loops generally need to do the following:

- Check the timer for all timed events (a timer countdown, regular screen changes, etc.).
- Check the keyboard or joystick for the player's instructions.
- Control the computer-controlled figures' actions.
- Control any continuing background sound.

Be selective. That's a short list, but it can still be far too long. To keep up the illusion of speed, you need to be selective — not all those things need to happen *every* time through the loop.

For instance, suppose you wanted to control the ten computer-figures whose movement we just programmed. Instead of putting them in a FOR-NEXT loop and moving all ten of them every time, why not access 300 as a subroutine with a random value? If the main loop contained this line, you'd get a very interesting effect:

```
120 I = INT(RND(9)*10):GOSUB 300
```

With this line as the only access to the computer-controlled figures, only *one* computer-figure will move each time the player-figure has a chance to move. There won't be such a long delay between the player-figure's movements. Even though each computer-controlled figure moves in the same pattern every time, you never know which one will move next. The computer-figures will actually move much more slowly, but because the player-figure will move faster and the computer-figures will be more random, the game will *feel* much faster and more exciting.

Use a single timer. If you try to control lots of events using a separate timer, you'll fill up your main loop with statements like `A = A + 1:IF A > 10 THEN A = 0:GOSUB 500`. Every arithmetic operation slows down your program, and so does every IF statement. Why not share a single timer?

Let's say you have three events to control. Subroutines 500 and 700 should happen only rarely; subroutine 800 must happen often; and subroutine 900 has to happen every other time through the main loop. That timer variable A can do triple duty:

```
120 A = A + 1:ON A GOSUB 900,700,900,800,900,500,900,800:IF  
A = 8 THEN A = 0
```

There it is — one mathematical operation and one GOSUB each time through the loop. Subroutines 700 and 500 will be executed only once in every eight passes through the main loop. Subroutine 800 will be executed twice as often, and subroutine 900 will be executed every second pass. Yet a single handler accesses them all.

This kind of timer routine, however, is not regular. Anything that stops the main loop from executing will also delay those subroutines. A missile firing or a collision that stops the action will also keep line 120 from executing, so it will be longer before the next action occurs. Regular movements (like the scrolling of the

screen in our Mission: Nova! game) and countdowns that should reflect realtime must be controlled using the TI\$ function. Remember, though, not to use equal signs with TI\$. For instance, this line could be a disaster:

```
IF VAL(TI$) = 7 THEN TI$ = "000000":GOSUB 500
```

What's wrong with it? Well, TI\$ will have a value of 7 for only a second. What if this line executes once when TI\$ = 6, but then the program happens to carry out a time-consuming collision routine before the next pass through the main loop? TI\$ will have a value of 9 or 10 by then, so this line will never execute again. Instead, you'll want to use this statement:

```
IF VAL(TI$) > 6 THEN TI$ = "000000":GOSUB 500
```

Now, whether TI\$ is 7 or 70, this line will snag the program and send it out to 500 at nearly the right time.

Stop the action. Sometimes you can stop the action completely to carry out some task, and the player won't feel that the game has been slowed down at all. For instance, when the player-figure collides with something, you can stop for an elaborate collision routine, with animation and sounds, and the player won't feel like it slows down the game — it will actually make it more exciting, and it will feel faster. Likewise, when the player fires a missile, or when an opposing figure first comes on the screen, you can take some time with it.

Stopping the action is fine; what you must avoid is a long time-lag between the player giving an instruction and the player-figure carrying it out. If the player is getting a quick response during the action phases of the game, you can take as long as you like in the other sections.

Memory: The Upper Limit

It's going to happen to you: you'll be working at your game, you'll type in a line, and all of a sudden the OUT OF MEMORY error will flash on the screen. What can you do?

There are three primary solutions:

1. Buy more memory.
2. Chain your program.
3. Crunch your program.

Buying memory. This is a financial decision, of course. I can promise you that your life will be easier with more memory in your VIC, but you can still program some excellent games with no

additional memory. Remember, though, that when you get anything more than the 3K expander, certain key memory locations change. In this book I've almost always used relocatable code: the program does not assume that screen and color memory are in a certain place, but rather finds out where they are. Even if you don't plan to get more memory anytime soon, it's a good idea to write programs that will adapt themselves to whatever memory setup they might find. After all, you may change your mind six months from now and buy more memory, and then you'd have to go back and change all your programs to fit.

Chaining programs. Chaining means that you LOAD and RUN only part of your program, and that part of the program contains instructions to automatically LOAD and RUN the next part.

When a program is being LOADED, any old program is completely erased. However, any values POKEd by the old program will stay in place until the power is turned off or the new program changes them. That means that you could write your game in two parts. Part 1 would contain the instructions and all the character set DATA. Once the instructions were through and the character set was in place, you would end Part 1 with this command:

```
PRINT "{CLEAR}LOADING PART 2":POKE 631,131:POKE
198,1
```

Be sure, though, that you don't put anything in Part 1 that you will want to RUN again in Part 2 — a screen-drawing subroutine in Part 1 just won't be there where you can get to it during Part 2.

Crunching. Without chaining or buying more memory, you can still pack a lot more program into the unexpanded VIC than you might suppose. All you need to do is crunch your program — remove excess bytes.

Which bytes are excess?

- Spaces between words. Almost the only places where you need to leave spaces are at the ends of numeric variable names and within messages that will be PRINTed on the screen. To the VIC, these two lines are exactly the same:

```
FOR I =9 TO 16 STEP 5: GET A$: IF A$ <> 45 THEN GOSUB 500:
NEXT
FOR I =9TO16STEP5:GETA$:IFAS <> 45THENGOSUB500:NEXT
```

Each space you leave out is one byte saved. The trouble is, it makes the program harder to read.

- **REM statements.** REMs are nice because they help you label what's going on in sections of a program — they make it easier for you or someone else to understand what your program is doing. But when you're dealing with limited memory, REMs are all expendable.

- **Multiple statements per line.** Each new line number is extra bytes used up in memory. You can save some memory by putting several statements per line, separated by colons (:). However, remember that everything on a line after an IF statement will be executed only if the condition is true. However, anything after an ON-GOSUB statement will be executed whether the condition is true or not. Anything on a line after an unconditional GOTO will never be executed at all.

- **Use short variable names.** Since the first two characters of a variable name are all that count, you might as well use only those two characters and save bytes.

- **Change numbers to variables.** Numbers take several bytes to store in memory, because the computer reserves enough memory for each number to store large numbers in that space. Thus a 0 uses up as much memory as 33999. However, a short variable takes up less space. Remember that you have to use up some bytes assigning a value to the variable — $A = 55$. That statement takes up some bytes, too, so you'll only save memory by assigning variable names to numbers that are used more than once. The most commonly used numbers are 0 and 1 — it's often a good idea to use Z0 and Z1 or N0 and N1 as the variable names for these numbers.

- **Use arrays, loops, and ON.** The program used as an example in the discussion of speed also saves memory because of the use of an array. You can do ten operations in a single FOR-NEXT loop instead of using ten separate lines. You can make five tests on a single ON statement instead of using five separate IF statements. By programming carefully, you can develop tighter code.

- **Trim introductions and farewells.** Important as they are, introductions, instructions, and farewell comments do eat up bytes. You may have to choose between having a really explicit introduction and having some important features in a game. Remember that you can often give the same amount of information in fewer words.

- **Use subroutines for any operation performed more than once.** If you have line after line that does almost the same thing except that a few numbers are different, you might consider using

a subroutine for the meat of the operation, and having each of the other lines set variable values and access that subroutine in order to perform the operation. This can sometimes slow down a game by adding extra GOSUBs and implied LETs, but if you're up against the end of memory, you sometimes have no choice.

- Don't use redefined characters. If you can make do with the built-in character set, you will save hundreds of bytes.

- Delete features. Sometimes a feature just isn't going to fit in an unexpanded VIC. If you can't or don't want to expand memory, you may just have to accept a limitation on what your game can include.

After all this talk of crunching memory, it's good to remember this: there are a lot of fantastic games that run on the unexpanded VIC. Yours can be one of them.





10
Missiles and
“Moonraker”



10 Missiles and "Moonraker"

There is a kind of figure that is partly under the player's control and partly under the computer's. When a baseball player in a videogame throws the ball, the player might control when the ball is released and the angle at which it is thrown, but rarely will the player control the ball's actual trajectory, its path across the screen. That is almost always controlled by the computer.

The same thing happens with missiles in *Asteroids*, bombs and bullets in *Scramble*, the juggler's ball in *Mr. Do*, and the air hose in *Dig-Dug*. They all seem to emerge from the player-figure, but once they are launched, their path and the distance they travel are usually controlled by the computer.

The principle of firing missiles is simple enough: when the player presses the joystick button or a designated key on the keyboard, the program jumps to a missile subroutine. A missile-figure is put on the screen one character away from the player-figure in the direction in which the player is shooting. Then the missile is moved across the screen as far as you want it to go.

In machine language games, the missile is moved in steps along with the player-figure, so that the player-figure can keep going after it shoots. In BASIC, however, it is usually better to let the whole game freeze until the shot is completed. That way when the other objects on the screen move at all, they move as quickly as ever. Trying to move a missile along with everything else will tend to slow down the program.

Add a Laser

Here are some lines to add to Program 6-2, "Mission: Nova!" If you have added in the new lines for sound routines in Chapter 7 and for introductions and farewells in Chapter 8, this missile routine won't fit in an unexpanded VIC. Just delete the introductory display and you'll have no problem. If you have an expanded VIC, you can add these lines directly to the full program.

**10
Missiles
and "Moonraker"**

Program 10-1. Starshot

```
40 NH=7:NV=7:C=NH+NV*22:W=32:SS=102:CS=2:TS=0:DI=-  
1  
100 A=PEEK(653)AND7:B=PEEK(197):IF B<55 THEN GOSUB  
300  
285 IF NH<>OH THEN DI=H  
300 IF (OH=0 AND DI=-1)OR(OH=21 AND DI=1) THEN RET  
URN  
305 MH=OH:MV=22*OV:BB=32:EM=0:IF DI>0 THEN EM=21  
310 FOR I=MH+DI TO EM STEP DI:MM=SC+I+MV:X=PEEK(MM  
) :IF X<>32 THEN EM=I-DI:GOTO 330  
315 POKE MM,70:NEXT  
320 FOR I=MH+DI TO EM STEP DI:MM=SC+I+MV:POKE MM,B  
B:NEXT I:RETURN  
330 FOR I=0 TO 7:POKE MM,77:POKE MM,93:POKE MM,78:  
POKE MM,67:NEXT  
340 IF X=46 THEN NN=81:BB=46:GOTO 370  
350 IF X=81 THEN NN=46:P=P+(8*NV)*INT(QQ-Q)  
360 IF X<>46 AND X<>81 THEN NN=104:P=P-1000  
370 POKE MM,NN:GOTO 320
```

Line	Function
40	Set the initial value of DI to -1, for a leftward shot.
100	If a key with a value less than 55 (f5) has been pressed, jump to the subroutine at 300.
285	If a horizontal move has been made, set DI to show the direction of the move. DI will therefore always show the direction of the last horizontal movement — and this will be the direction of the next laser shot.
300	If the player-figure is at the edge of the screen, shooting off the edge, return without doing anything.
305	Set MV and MH to the current vertical and horizontal location of the player-figure, plus one step in the direction of the shot, so that the laser beam will appear beside, not on top of, the player-figure. Then set EM to 0 for leftward shots, 21 for rightward shots.
310-315	If the next character in the direction of the shot is a blank (32), POKE the laser character (70) into that place. If the next character is not a blank, set EM to the last location where a 70 was POKED and jump to the explosion subroutine at line 330.

- 320 Replace the laser with blanks (BB = 32) and return to the main loop of the program.
- 330 Show an animated, twirling explosion.
- 340 If the laser hit a small star, turn it into a nova, but set BB to 46, so that all the spaces between the player-figure and the new nova are small stars. We wouldn't want the player to rack up a million points by changing small stars to novas and moving right over to them without any obstacles.
- 350 If the laser hits a nova, add the appropriate number of points and replace the nova with a small star.
- 360 If the laser hit anything else, change it to a used-up space station (104) and subtract 1000 from the score. This is because the only thing the laser can hit, besides a blank, a small star, and a nova, is the space station itself. Not the sort of thing that earns many points.
- 370 POKE the new character into place and go back to line 330 to erase the laser (or replace it with small stars).

A Missile Version

If you don't like the continuous beam of the laser, it's a simple matter to replace it with a missile. Basically, all you have to do is erase the last missile character as soon as you POKE in the new one. The tricky part with missiles is how to handle the *end* of their flight. It has to be different if the flight ends with a collision or simply with the edge of the screen. The virtue of this routine is that it's very quick, very dartlike in its movement, in spite of being in BASIC. This version still preserves the feature of not allowing the player to change small stars right next to the ship into novas. However, if you want to remove that feature, just replace line 340 with

```
340 IF X=46 THEN 380
```

Otherwise, use lines 40, 100, and 285 from Program 10-1, and replace lines 300-370 with this program:

Program 10-2. Starshot with Missiles

```
300 IF (OH=0 AND DI=-1)OR(OH=21 AND DI=1) THEN RET  
URN  
305 MH=OH: MV=22*OV: BB=32: EM=0: IF DI>0 THEN EM=21
```

10 Missiles and "Moonraker"

```
310 FOR I=MH+DI TO EM STEP DI:MM=SC+I+MV:X=PEEK(MM
   ):IF I<>MH+DI THEN POKE MM-DI,32
315 IF X<>32 THEN EM=I:GOTO 330
320 POKE MM,70:NEXT:POKE MM,32
325 RETURN
330 FOR I=0 TO 7:POKE MM,77:POKE MM,93:POKE MM,78:
   POKE MM,67:NEXT
340 IF X=46 THEN NN=46:IF EM<>MH+DI THEN 380
350 IF X=81 THEN NN=46:P=P+(8*NV)*INT(QQ-Q)
360 IF X<>46 AND X<>81 THEN NN=104:P=P-1000
370 POKE MM,NN:GOTO 325
380 NN=81:IF EM<>MH+DI THEN FOR I=EM TO MH+DI STEP
   {SPACE}-DI:POKE SC+MV+I,46:NEXT:GOTO 390
385 POKE MM,46
390 GOTO 370
```

Your First Game — and Mine

We've been through all the principles of programming arcade games in BASIC on the VIC. Using other reference books, you'll be able to pick up some more sophisticated programming techniques, and if you're serious about game design, you'll eventually turn to machine language. But for now, you're certainly ready to write a game. Your first attempt might be simple, but if you program it carefully, leaving plenty of line numbers for adding new subroutines to improve the game, your first game will turn out pretty well. And with practice, you'll gain skill and confidence.

But no matter how good you eventually become, you'll probably always feel kind of proud of your first game. I know I feel that way about mine. "Moonraker" was the result of my first struggle with game programming. It went through a lot of versions, but I finally settled on this one.

There are some features of the game that reflect my situation at the time. For one thing, I used POKE 36864,4 to move the screen display to the right. I had to — my old TV didn't scan correctly, and without that POKE I couldn't see the whole screen. In fact, you might enjoy playing around with POKES at 36864 to see the effect it has. Try entering this one-line program and RUNNING it:

```
10 FOR X=0 TO 20:POKE 36864,X:NEXT:FOR
   X=20 TO 0 STEP -1:POKE 36864,X:NEXT:GOTO 10
```

To get your screen back to normal, press RUN/STOP and then POKE 36864,5.

You'll notice that this program doesn't skip any line numbers.

No, I didn't write it that way. I left plenty of extra line numbers while I was programming. But when I was through, I used a line-renumbering utility to tighten down my program.

Since I was using an unexpanded VIC, I used 7680 directly as the starting address of screen memory. I also found that I was running out of memory with all my title screens. So I used a fairly sophisticated technique for fooling BASIC into erasing the introductory part of the program after it was entered. But that's a programming technique, not a game technique — the sort of thing you'll want to start learning as you get better at game programming, but certainly not essential when you're just starting out.

So here's my first full-fledged game. I hope you like it. Who knows? Maybe I'll get to play yours someday.

Program 10-3. Moonraker

```

1 PRINT"{CLR}{8 DOWN}{6 SPACES}MOON RAKER":PRINT
2 PRINT"{7 DOWN}{RVS}HIT ANY KEY TO CONT.{OFF} )
3 GETA$: IFA$="" THEN 3
4 GOTO 87
5 PRINT"{CLR}{HOME}{2 DOWN}{6 RIGHT}CONTROLS:"
6 PRINT:PRINT"{CYN}{4 RIGHT}UP= {RVS}@{OFF}":PRINT
7 PRINT"{4 RIGHT}DOWN= {RVS}SPACE BAR{OFF}":PRINT
8 PRINT"{4 RIGHT}RIGHT= {RVS}={OFF}
9 PRINT:PRINT"{4 RIGHT}LEFT={RVS}:{OFF}":PRINT:PRI
10 PRINT"{4 RIGHT}FIRE= {RVS}A{OFF}":PRINT:PRINT"
11 {2 RIGHT}YOU HAVE 2 MINUTES"
12 PRINT:PRINT:PRINT"{3 RIGHT}HIT A * TO BEGIN"
13 GETA$: IFA$="" THEN 10
14 POKE 52, 28: POKE 56, 28: POKE 36864, 7
15 CS=7168: X=0
16 FOR I=CSTOCS+511: POKE I, PEEK(I+25600): NEXT
17 FOR I=0 TO 7: READ J: POKE CS+I, J: NEXT
18 FOR I=16 TO 39: READ J: POKE CS+I, J: NEXT
19 FOR I=48 TO 87: READ J: POKE CS+I, J: NEXT
20 FOR I=96 TO 103: READ J: POKE CS+I, J: NEXT
21 FOR I=128 TO 143: READ J: POKE CS+I, J: NEXT
22 FOR I=152 TO 255: READ J: POKE CS+I, J: NEXT
23 DATA 102, 153, 153, 102, 102, 153, 153, 102
24 DATA 254, 60, 29, 15, 15, 29, 60, 254, 0, 56, 124, 198, 198,
25 124, 56, 0
26 DATA 15, 3, 1, 0, 0, 1, 3, 15, 224, 195, 215, 252, 252, 215, 1
27 95, 224
28 DATA 0, 128, 192, 96, 96, 192, 128, 0, 0, 0, 0, 0, 254, 60, 29
29 , 15
30 DATA 0, 0, 0, 0, 0, 56, 124, 198, 15, 29, 60, 254, 0, 0, 0, 0, 1
31 98, 124, 56, 0, 0, 0, 0, 0

```

**10
Missiles
and "Moonraker"**

```
25 DATA48,72,180,207,207,180,72,48,0,0,0,255,255,0
,0,0
26 DATA136,73,74,42,165,90,189,126,34,132,88,61,60
,154,33,72
27 DATA36,36,24,24,60,60,36,24,6,136,152,124,62,25
,17,96
28 DATA0,0,0,0,36,36,24,24,8,8,28,34,62,62,85,85
29 DATA60,60,36,24,0,0,0,0,0,0,0,0,0,0,255,0,255,0
,0,0,0,0,0,0
30 DATA128,64,32,16,8,4,3,0,1,2,4,8,16,32,192,0,0,
128,127,0,0,0,0,0,0,1,254,0,0,0,0,0
31 PRINT"{CLR}":POKE36869,255:CLR:POKE36879,8
32 PRINT"{RED}":{20 DOWN}Z]↑←£ZZZ]↑←£][[↑←[[£Z"
33 L=7900:V=36878:S=36877:L1=7788:L2=7723:L3=7718
:L4=8112:L5=8116:TIS="000000":U=32
34 M=22:O=23
35 PRINT"{HOME}{RVS}{YEL}{3 RIGHT}MOONRAKER{WHT}
{RIGHT}";X
36 POKEL3,21:POKEL3-M,U:POKEL1,0:POKEL2,M:POKEL4,2
4:POKEL5,24:POKEL,2:POKEL+1,3
37 POKEL2,U:N=255:IFL2=>8054THENL2=L2-M
38 IFL2<=7702THENL2=L2+M
39 P=INT(RND(1)*4):IFP=0THENL2=L2-44
40 IFP=1THENL2=L2+44
41 IFP=2THENL2=L2+2
42 IFP=3THENL2=L2-2
43 IFL3=L4ORL3=L5THENX=X-500:GOTO75
44 IFL3=>8098THEN75
45 POKEL3,O:POKEL3+M,25:L3=L3+M:F=L1+21:POKEF,0:PO
KEL1,U:L1=F
46 IFF=>8098THENPOKEF,U:L1=INT(RND(1)*O)+7702
47 IFL=L1ORL=L2ORL=L3THENX=X-250:GOTO77
48 IFL+1=L1ORL+1=L2ORL+1=L3THENX=X-250:GOTO77
49 IFL>8076THENPOKEL,U:POKEL+1,U:L=L-M:GOTO35
50 IFL<7724THENPOKEL,U:POKEL+1,U:L=L+M:GOTO35
51 IFPEEK(197)=46THEN59
52 IFPEEK(197)=45THEN60
53 IFPEEK(197)=53THEN63
54 IFPEEK(197)=UTHEN61
55 IFPEEK(197)=17THEN65
56 IFE<0THEN80
57 IFTIS>"000120"THEN79
58 GOTO35
59 POKEL,4:POKEL+1,6:POKEL+2,7:POKEL,U:POKEL+1,2:P
OKEL+2,3:L=L+1:GOTO35
60 POKEL-1,4:POKEL,6:POKEL+1,7:POKEL-1,2:POKEL,3:P
OKEL+1,U:L=L-1:GOTO35
61 POKEL,8:POKEL+1,9:POKEL+M,10:POKEL+O,12
62 POKEL,U:POKEL+1,U:POKEL+M,2:POKEL+O,3:L=L+M:GOT
O35
```

```

63 POKEL,10:POKEL+1,12:POKEL-M,8:POKEL-21,9
64 POKEL,U:POKEL+1,U:POKEL-M,2:POKEL-21,3:L=L-M:GO
   TO35) 56
65 POKEL+2,16:POKEV,15:POKE36877,N
66 FORF=0TO18:POKEL+3+F,17:POKEC+F+3,2:POKEL+2+F,U
   :POKEL+2,U:POKE36877,N
67 L6=L+3+F:IFL6=L1THEN X=X+150:GOTO71
68 IFL6=L2THENX=X+250:GOTO73
69 IFL6=L3ORL6=L3-MTHENX=X+50:GOTO75
70 N=N-12:NEXT:POKEL+21,U:POKEV,0:GOTO35
71 POKEV,15:POKES,130:POKEL1,20:FORT=0TO600:NEXT
72 POKEL1,U:POKEV,0:L1=INT(RND(1)*O)+7702:GOTO35
73 POKEV,15:POKES,130:POKEL6,20:FORT=0TO600:NEXT
74 POKEL6,U:POKEV,0:GOTO35) 36
75 POKEV,15:POKES,130:POKEL3,20:FORT=0TO600:NEXT
76 POKEL3,U:POKEL3-M,U:L3=INT(RND(1)*O)+7702:POKEV
   ,0:GOTO35
77 POKEL,20:POKEL+1,20:POKEV,15:POKES,130:FORT=0TO
   600:NEXT
78 POKEV,0:GOTO35
79 POKEL198,0:POKE36869,240:PRINT"{CLR}{5 DOWN}TIME
   IS UP:YOUR SCORE IS: ";X
81 PRINT:PRINT"HIT THE * TO TRY AGAINOR THE RUN/ST
   OP TO END
82 GETA$:IFA$=""THEN82
83 GOT011
84 A=PEEK(61)+256*PEEK(62)+3:POKE2,INT(A/256):POKE
   1,A-256*PEEK(2)
85 A=PEEK(61)+256*PEEK(62)+3:POKE2,INT(A/256):POKE
   1,A-256*PEEK(2)
86 IFERTHENPOKEA-2,0:POKEA-1,0:POKE45,PEEK(1):POKE
   46,PEEK(2):CLR:GOTO6
87 PRINT"{CLR} AS COMMANDER OF MOON-{DOWN}RAKER YO
   U ARE TO{2 SPACES}PRO-{DOWN}TECT THE 2 RESEARCH
88 PRINT"{DOWN}PODS ON THE MOON'S{4 SPACES}{DOWN}S
   URFACE. HITS BY ALIEN{DOWN}TO YOU OR YOUR
   {2 SPACES}PODS
89 PRINT"{DOWN}CAUSES LOSS OF POINTS.{DOWN}BLAST T
   HE ALIENS WITH {DOWN}YOUR LASER FOR POINTS
90 PRINT"{3 DOWN}{RVS}HIT ANY KEY TO CONT.{OFF}
91 GETA$:IFA$=""THEN91
92 PRINT"{CLR}BEWARE OF THE RANDOM{2 SPACES}{DOWN}
   MOVING TWIRLER WHICH{2 SPACES}{DOWN}STRIKES WIT
   HOUT NOTICE
93 PRINT"AND ROBS YOU OF POINTS{DOWN}YOU WILL HAVE
   120 SEC-{DOWN}ONDS TO GET AS MANY
94 PRINT"{DOWN}POINTS AS YOU CAN.{4 SPACES}
   {2 DOWN} GOOD LUCK COMMANDER!!
95 PRINT"{2 DOWN}{RVS}HIT ANY KEY TO CONT.{OFF}
96 GETA$:IFA$=""THEN96
97 ER=1:GOTO84

```





Appendices



A Beginner's Guide to Typing In Programs

What Is a Program?

A computer cannot perform any task by itself. Like a car without gas, a computer has *potential*, but without a program, it isn't going anywhere. Most of the programs published in COMPUTE! Books are written in a computer language called BASIC. BASIC is easy to learn and is built into all VIC-20s.

BASIC Programs

Computers can be picky. Unlike the English language, which is full of ambiguities, BASIC usually has only one right way of stating something. Every letter, character, or number is significant. A common mistake is substituting a letter such as O for the numeral 0, a lowercase l for the numeral 1, or an uppercase B for the numeral 8. Also, you must enter all punctuation such as colons and commas just as they appear in the book. Spacing can be important. To be safe, type in the listings *exactly* as they appear.

Braces and Special Characters

The exception to this typing rule is when you see the braces, such as {DOWN}. Anything within a set of braces is a special character or characters that cannot easily be listed on a printer. When you come across such a special statement, refer to "How to Type In Programs."

About DATA Statements

Some programs contain a section or sections of DATA statements. These lines provide information needed by the program. Some DATA statements contain actual programs (called machine language); others contain graphics codes. These lines are especially sensitive to errors.

If a single number in any one DATA statement is mistyped, your machine could lock up, or crash. The keyboard and STOP key may seem dead, and the screen may go blank. Don't panic —

no damage is done. To regain control, you have to turn off your computer, then turn it back on. This will erase whatever program was in memory, *so always SAVE a copy of your program before you RUN it.* If your computer crashes, you can LOAD the program and look for your mistake.

Sometimes a mistyped DATA statement will cause an error message when the program is RUN. The error message may refer to the program line that READs the data. *The error is still in the DATA statements, though.*

Get to Know Your Machine

You should familiarize yourself with your computer before attempting to type in a program. Learn the statements you use to store and retrieve programs from tape or disk. You'll want to save a copy of your program, so that you won't have to type it in every time you want to use it. Learn to use the VIC's editing functions. How do you change a line if you made a mistake? You can always retype the line, but you at least need to know how to backspace. Do you know how to enter reverse characters, lowercase, and control characters? It's all explained in your VIC-20's manual, *Personal Computing on the VIC.*

A Quick Review

- 1) Type in the program a line at a time, in order. Press RETURN at the end of each line. Use the INST/DEL key to erase mistakes.
- 2) Check the line you've typed against the line in the book. You can check the entire program again if you get an error when you RUN the program.
- 3) Make sure you've entered statements in braces as the appropriate control key (see "How to Type In Programs").

How to Type In Programs

Many of the programs listed in COMPUTE! Books contain special control characters (cursor control, color keys, reverse characters, etc.). To make it easy to know exactly what to type when entering one of these programs into your computer, we have established the following listing conventions.

Generally, any VIC-20 program listings will contain words in braces which spell out any special characters: {DOWN} would mean to press the cursor down key. {5 SPACES} would mean to press the space bar five times.

To indicate that a key should be *shifted* (hold down the SHIFT key while pressing the other key), the key would be underlined in our listings. For example, S would mean to type the S key while holding the shift key. This would appear on your screen as a heart symbol. If you find an underlined key enclosed in braces (e.g., {10 N}), you should type the key as many times as indicated (in our example, you would enter ten shifted N's).

If a key is enclosed in special brackets, [**<**>], you should hold down the *Commodore key* while pressing the key inside the special brackets. (The Commodore key is the key in the lower-left corner of the keyboard.) Again, if the key is preceded by a number, you should press the key as many times as necessary.

Rarely, you'll see a solitary letter of the alphabet enclosed in braces, such as {A}. You should never have to enter such a character on the VIC-20, but if you do, you would have to leave the quote mode (press RETURN and cursor back up to the position where the control character should go), press CTRL-9 (RVS ON), the letter in braces, and then CTRL-0 (RVS OFF).

About the *quote mode*: you know that you can move the cursor around the screen with the CRSR keys. Sometimes a programmer will want to move the cursor under program control. That's why you see all the {LEFT}'s, {HOME}'s, and {BLU}'s in our programs. The only way the computer can tell the difference between direct and programmed cursor control is the quote mode.

Once you press the quote (the double quote, SHIFT-2), you are in the quote mode. If you type something and then try to change it by moving the cursor left, you'll only get a bunch of

B
How to Type
In Programs

reverse-video lines. These are the symbols for cursor left. The only editing key that isn't programmable is the DEL key; you can still use DEL to back up and edit the line. Once you type another quote, you are out of quote mode.

You also go into quote mode when you INSerT spaces into a line. In any case, the easiest way to get out of quote mode is to just press RETURN. You'll then be out of quote mode and you can cursor up to the mistyped line and fix it.

Use the following table when entering cursor and color control keys:

When You Read:	Press:	See:	When You Read:	Press:	See:
{ CLEAR }	SHIFT CLR/HOME		{ GRN }	CTRL 6	
{ HOME }	CLR/HOME		{ BLU }	CTRL 7	
{ UP }	SHIFT ↑ CRSR ↓		{ YEL }	CTRL 8	
{ DOWN }	↓ CRSR ↑		{ F1 }	f1	
{ LEFT }	SHIFT ← CRSR →		{ F2 }	f2	
{ RIGHT }	→ CRSR ←		{ F3 }	f3	
{ RVS }	CTRL 9		{ F4 }	f4	
{ OFF }	CTRL 0		{ F5 }	f5	
{ BLK }	CTRL 1		{ F6 }	f6	
{ WHT }	CTRL 2		{ F7 }	f7	
{ RED }	CTRL 3		{ F8 }	f8	
{ CYN }	CTRL 4		←	←	
{ PUR }	CTRL 5		↑	SHIFT ↑	

Screen Location Table

Row	Address (VIC)	Grid
0	7680 (4096)	
	7702 (4118)	
	7724 (4140)	
	7746 (4162)	
	7768 (4184)	
5	7790 (4206)	
	7812 (4228)	
	7834 (4250)	
	7856 (4272)	
	7878 (4294)	
10	7900 (4316)	
	7922 (4338)	
	7944 (4360)	
	7966 (4382)	
	7988 (4404)	
15	8010 (4426)	
	8032 (4448)	
	8054 (4470)	
	8076 (4492)	
	8098 (4514)	
20	8120 (4536)	
	8142 (4558)	
22	8164 (4580)	

0
5
10
15
20

Column

Note: Numbers in parentheses are for VICs with 8K or more of memory expansion.

Screen Color Memory Table

Row	Column
0	38400 (37888) 38422 (37910) 38444 (37932) 38466 (37954) 38488 (37976)
5	38510 (37998) 38532 (38020) 38554 (38042) 38576 (38064) 38598 (38086)
10	38620 (38108) 38642 (38130) 38664 (38152) 38686 (38174) 38708 (38196)
15	38730 (38218) 38752 (38240) 38774 (38262) 38796 (38284) 38818 (38306)
20	38840 (38328) 38862 (38350)
22	38884 (38372)

Note: Numbers in parentheses are for VICs with 8K or more of memory expansion.

Screen Color Codes

Color:	BLK	WHT	RED	CYN	PUR	GRN	BLU	YEL
Code:	0	1	2	3	4	5	6	7

Screen and Border Colors

Screen	Border							
	Black	White	Red	Cyan	Purple	Green	Blue	Yellow
Black	8	9	10	11	12	13	14	15
White	24	25	26	27	28	29	30	31
Red	40	41	42	43	44	45	46	47
Cyan	56	57	58	59	60	61	62	63
Purple	72	73	74	75	76	77	78	79
Green	88	89	90	91	92	93	94	95
Blue	104	105	106	107	108	109	110	111
Yellow	120	121	122	123	124	125	126	127
Orange	136	137	138	139	140	141	142	143
Light Orange	152	153	154	155	156	157	158	159
Pink	168	169	170	171	172	173	174	175
Light Cyan	184	185	186	187	188	189	190	191
Light Purple	200	201	202	203	204	205	206	207
Light Green	216	217	218	219	220	221	222	223
Light Blue	232	233	234	235	236	237	238	239
Light Yellow	248	249	250	251	252	253	254	255

ASCII Codes













ASCII	CHARACTER	ASCII	CHARACTER
5	WHITE	50	2
8	DISABLE	51	3
	SHIFT COMMODORE	52	4
9	ENABLE	53	5
	SHIFT COMMODORE	54	6
13	RETURN	55	7
14	LOWERCASE	56	8
17	CURSOR DOWN	57	9
18	REVERSE VIDEO ON	58	:
19	HOME	59	;
20	DELETE	60	<
28	RED	61	=
29	CURSOR RIGHT	62	>
30	GREEN	63	?
31	BLUE	64	@
32	SPACE	65	A
33	!	66	B
34	"	67	C
35	#	68	D
36	\$	69	E
37	%	70	F
38	&	71	G
39	'	72	H
40	(73	I
41)	74	J
42	*	75	K
43	+	76	L
44	,	77	M
45	-	78	N
46	.	79	O
47	/	80	P
48	0	81	Q
49	1	82	R

G
ASCII
Codes

ASCII	CHARACTER	ASCII	CHARACTER
83	S	120	
84	T	121	
85	U	122	
86	V	123	
87	W	124	
88	X	125	
89	Y	126	
90	Z	127	
91	[133	f1
92	£	134	f3
93]	135	f5
94	↑	136	f7
95	↑	137	f2
96		138	f4
97		139	f6
98		140	f8
99		141	SHIFTED RETURN
100		142	UPPERCASE
101		144	BLACK
102		145	CURSOR UP
103		146	REVERSE VIDEO OFF
104		147	CLEAR SCREEN
105		148	INSERT
106		156	PURPLE
107		157	CURSOR LEFT
108		158	YELLOW
109		159	CYAN
110		160	SPACE
111		161	
112		162	
113		163	
114		164	
115		165	
116		166	
117		167	
118		168	
119		169	

ASCII	CHARACTER	ASCII	CHARACTER
170		207	
171		208	
172		209	
173		210	
174		211	
175		212	
176		213	
177		214	
178		215	
179		216	
180		217	
181		218	
182		219	
183		220	
184		221	
185		222	
186		223	
187		224	SPACE
188		225	
189		226	
190		227	
191		228	
192		229	
193		230	
194		231	
195		232	
196		233	
197		234	
198		235	
199		236	
200		237	
201		238	
202		239	
203		240	
204		241	
205		242	
206		243	

**G
ASCII
Codes**

ASCII	CHARACTER
244	
245	
246	
247	
248	
249	
250	
251	
252	
253	
254	
255	

1. 0-4, 6-7, 10-12, 15-16, 21-27, 128-132, 143, and 149-155 have no effect.
2. 192-223 same as 96-127, 224-254 same as 160-190, 255 same as 126.

Screen Codes

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

H Screen Codes

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase	POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
62	>	>	95		
63	?	?	96	--space--	
64			97		
65		A	98		
66		B	99		
67		C	100		
68		D	101		
69		E	102		
70		F	103		
71		G	104		
72		H	105		
73		I	106		
74		J	107		
75		K	108		
76		L	109		
77		M	110		
78		N	111		
79		O	112		
80		P	113		
81		Q	114		
82		R	115		
83		S	116		
84		T	117		
85		U	118		
86		V	119		
87		W	120		
88		X	121		
89		Y	122		
90		Z	123		
91			124		
92			125		
93			126		
94			127		

128-255 are reverse video of 0-127.

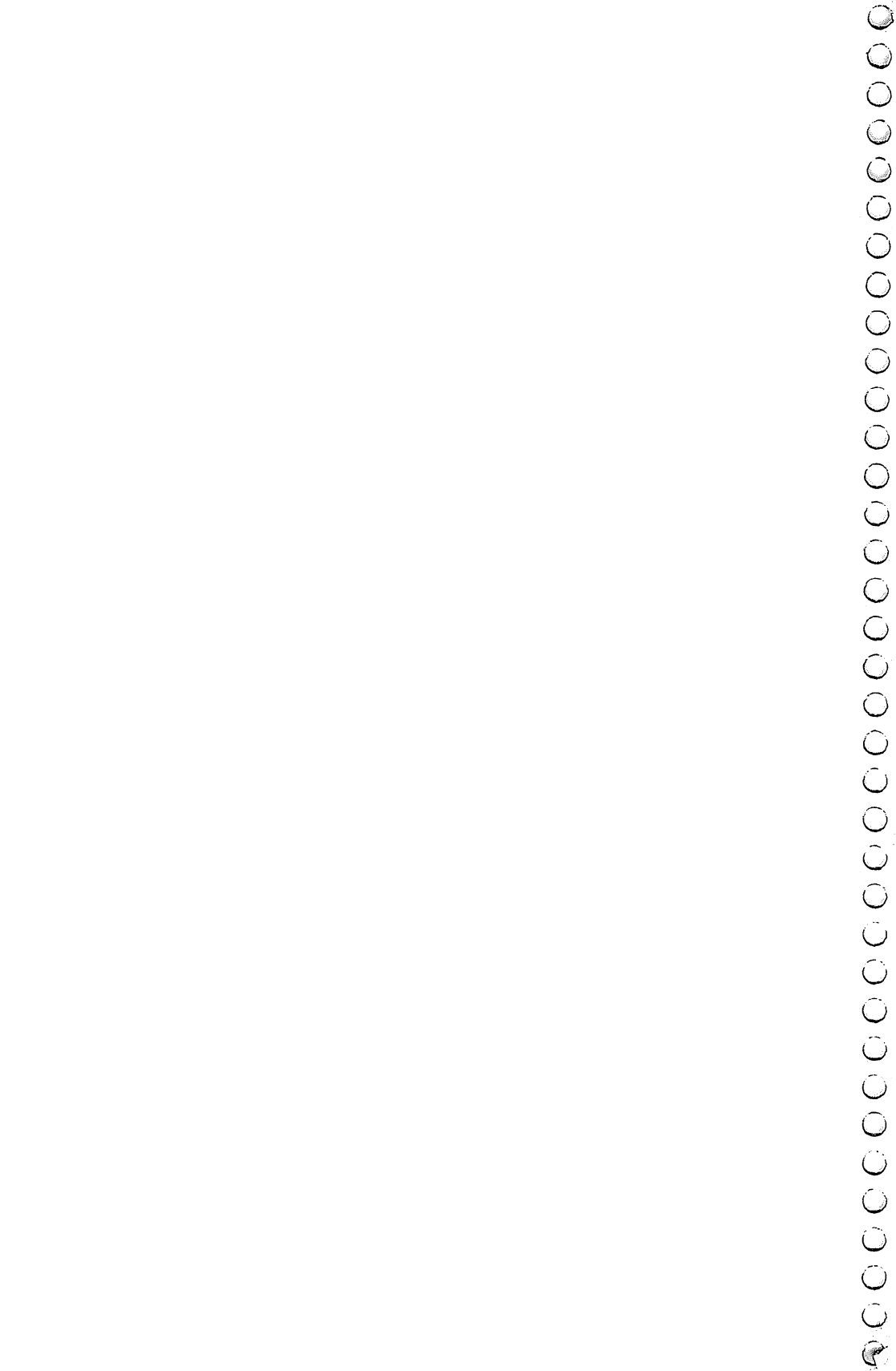
Index

- animation 4, 20
 - animation in place routine 94-95
 - defined 80
 - memory considerations 27
 - minimum speed for smooth 79
 - player-controlled 96-97
 - smooth 96
 - with movement 95
- Apple computer 14
- arrays, for program crunching 154
- artificial intelligence 30-31
- ASCII code 48, 49
 - table 177-80
- Asteroids* 25, 32, 36, 37, 159
- Atari computer 14
- "Athlete in Action" program 97-98
- background sound 122-23
 - in main loop of game 123
- BASIC 3
 - relatively slow 13-14
- Berserk* 29-30
- "Blip" program 120
- border colors 176
- Breakout* 26
- Burger Time* 24, 31
- cassettes, cheap ones work better 10
- Centipede* 25, 29
- central processing unit *see* CPU
- chaining programs 152-53
- character matrix 63-64
- character memory 65
- characters 26
 - replacing 71-73
 - sequential 73-74
- character set 45
 - copying into RAM for customization 68-75
 - stored in ROM 68
- character work grid 67
- children, and game testing 7-8
- CHR\$ function 48
- clock, in "Subroutine Sound" program 125
- collisions 101-14
 - detection of 101, 129
- color key 56
- color memory 45, 102
 - example program and discussion 57-59
 - finding 57
 - limitation on POKE 56
 - manipulating 56-59
 - table 174
- colors, border, manipulating 43-45
- colors, playfield, manipulating 43-45
- colors on VIC 12
- combining characters 54
- Commodore 14
- Commodore 64 Programmer's Reference Guide* 107
- complications, theory of 31
- COMPUTE!'s Second Book of VIC* 119
- control matrix 82
- "Copying Characters" program 69-71
- CPU 13
- "Creating Characters" program 69
- crunching programs 152, 153-55
- custom characters 27
 - creating 63-75
 - memory considerations 27
 - program crunching and 155
- custom character sets 4, 13, 26-27
- DATA statements 169-70
- defeat, psychology of 32
- Defender* 19, 25, 37
- delay loop 98
- Dig-Dug* 37, 159
- documentation 19-20, 38-39, 150-51
- Donkey Kong* 19, 24, 25, 26, 29, 31, 32, 35, 37, 80, 117
- Donkey Kong Jr.* 27, 30, 37
- existing games, improvement of 25-26
- experimentation 5-6
- extras 135-44
- farewells 135, 141-43
 - personalizing 144
- "Fill In" program 53
- Firebird* 25
- FOR-NEXT loops 3, 5
- FRE function 75, 136
- Galaga* 25, 35
- Galaxians* 25, 36
- game design, different from programming 6
 - discussion 19-40
- game story 25-26
 - importance of originality 25
- GOSUB 3, 5, 28, 136
- GOTO 3
- graph paper 10
- graphics characters 13
- habits, important in good programming 148
- homing pattern 29, 30
- horizontal movement 85
- ideas 6-7
- in-line logic 89
- "Inchworm" program 95

instructions 135, 138-41
 interest-maintaining techniques 23, 32-38
 bonus turns 37
 complication 34
 incentives 35
 increasing accuracy 34
 increasing difficulty 33-34
 in *Joust* 20
 new scenery 34-35
 scoring 35-36
 story rewards 37
 introductions 135-38
 invisible objects 44
Joust 19, 26, 30, 31, 34, 37, 117
 game analyzed 20-24
 joystick 19, 20, 28
 reading 92-93
 joystick button 93
Kangaroo 30, 35, 37
 keyboard reading 81-85
 key code 81, 83
 table 84
 "Keysound" program 120
 discussion 120-21
 "Keysound UFO" program 121
 discussion 121-22
 "Laser X Jet" program 91-92
 loops, for program crunching 154
 machine language 13, 28
 and animation 79
 magazines, programs in 5-6
 main loop, proper function of 150
 marketing 8-10
 memory, economical use of 135
 memory locations 14
 MID\$ function 105
 missile routines 159-65
Missile Command 25, 33, 35, 37
 "Mission: Noval" program 105-6
 discussion 106-9
 farewell code 143
 instructions code 139-40
 introduction code 136
 laser routine in 159-61
 missile routine in 161-62
 modified by GOSUBs 136
 sounds, in code and discussion, 128-31
Monopoly 32, 35
 "Moonraker" program
 discussion 162-63
 program 163-65
 movement 79-81
 defined 80
 demonstration program 80-81
 with animation 95
 with POKE program 86-87
Mr. Do 159
Ms. Pac-Man 19
 "Multicharacter PRINT Movement"
 routine 90
 noise register 118
 ON statement, for program crunching 154
 opponents, intelligent 29-30
 opponents, mindless 30
 originality, in game story 25
Pac-Man 24, 26, 30, 31, 32, 35, 37, 101, 117,
 118
 PEEK 14
 explained 46-47
Personal Computing on the VIC-20 1, 119
 pixel 12, 26, 64
 "Player-Controlled Animation" routine
 96-97
 player movement 20
 trade off speed and complexity 28
 point inflation 36
 POKE 14
 explained 46-47
 PRINT 3, 48
 combined with POKE 112-14
 dangers of 88-90
 faster than POKE 88
 moving cursor with 55-56
 screen displays with 54-55
 "PRINT Movement Routine" 87-88
 programming techniques 147-55
 program translation 14-15
Qix 25
Rally-X 31
 RAM 11
 5K basic option on VIC-20 11-12
 random access memory *see* RAM
 random screen displays 51-53
 raster 79
 READ statement 52
 in character replacement 72-73
 REM statement 154
 RESTORE statement 52
 RND function 51-52
Robotron 25
Scramble 159
 screen, VIC-20 12
 screen codes 48-49
 table 181-82
 screen color codes 175, 176
 screen color memory table 174
 screen design 4, 43-58
 screen location table 173
 screen memory 45, 102
 finding 57
 manipulating with PEEK and POKE 47
 organization 49

screen relationships 29
 scrolling 104-5
 6502 chip 14
 not all machines using it compatible
 14-15
 sound 4
 importance of 37-38
 in *Joust* 23-24
 in "Mission: Nova!" 128-31
 in subroutines 123-27
 turning off 120
 uses of 117-18
 "Sound Game" program 127
 discussion 127-28
 sound registers 118
Space Invaders 28, 30, 33, 117
 "Spaceship Collisions" program 101-2
 discussion 102-3
 "Spark" program 113-14
 discussion 112-13
 farewell code 42
 setting difficulty in 140-41
 SPC function 56
 speed, real and apparent 150-51
 "Star-Eater" program 110-12
 "Starfield" program 51-53
 STEP function 5
 submission to publishers 9-10
 subroutines, program crunching and
 154-55
 subroutines, sound 123-31
 halt game action 124
 no-stop subroutines 126-28
 "Subroutine Sound" programs 124, 126
 discussion 124-25, 126

Super Breakout 36
Surround 26
 TAB function 55
Tempest 25, 34, 35
 testing 7
 timer, common 151-52
 timer reset routine 109
 TI\$ variable 107, 109, 110
Tron 26
 true/false test 44
 unpredictability 33
 programming 33
 vanity board 36-37
 in *Joust* 23
 variables, take less space than numbers
 154
Venture 25, 35
 vertical movement 85
 VIC-20
 colors 12
 languages available 13
 RAM 11-12
 screen 12
 sound 13
 specifications 11-15
VIC-20 Programmer's Reference Guide 10, 68
VIC-20 User's Guide 10, 68
 Video Interface Chip (VIC) 45
 volume control register 118
 weaknesses, need for in computer
 opponents 31
 "Wire" program 54-55
 wraparound 89-90
 "Zap" program 119



If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!** Magazine. Use this form to order your subscription to **COMPUTE!**.

For Fastest Service,
Call Our **Toll-Free** US Order Line

800-334-0868
In NC call **919-275-9809**

COMPUTE!

P.O. Box 5406
Greensboro, NC 27403

My Computer Is:

- Commodore 64 TI-99/4A Timex/Sinclair VIC-20 PET
 Radio Shack Color Computer Apple Atari Other _____
 Don't yet have one...

- \$24 One Year US Subscription
 \$45 Two Year US Subscription
 \$65 Three Year US Subscription

Subscription rates outside the US:

- \$30 Canada
 \$42 Europe, Australia, New Zealand/Air Delivery
 \$52 Middle East, North Africa, Central America/Air Mail
 \$72 Elsewhere/Air Mail
 \$30 International Surface Mail (lengthy, unreliable delivery)

Name _____

Address _____

City _____

State _____

Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank; International Money Order, or charge card.

- Payment Enclosed VISA
 MasterCard American Express

Acc t. No. _____

Expires _____ / _____



If you've enjoyed the articles in this book, you'll find the same style and quality in every monthly issue of **COMPUTE!'s Gazette** for Commodore.

For Fastest Service
Call Our **Toll-Free** US Order Line
800-334-0868
In NC call **919-275-9809**

COMPUTE!'s GAZETTE

P.O. Box 5406
Greensboro, NC 27403

My computer is:

Commodore 64 VIC-20 Other _____
01 02 03

- \$20 One Year US Subscription
 \$36 Two Year US Subscription
 \$54 Three Year US Subscription

Subscription rates outside the US:

- \$25 Canada
 \$45 Air Mail Delivery
 \$25 International Surface Mail

Name _____

Address _____

City _____

State _____

Zip _____

Country _____

Payment must be in US Funds drawn on a US Bank, International Money Order, or charge card. Your subscription will begin with the next available issue. Please allow 4-6 weeks for delivery of first issue. Subscription prices subject to change at any time.

- Payment Enclosed VISA
 MasterCard American Express

Acct. No. _____

Expires _____

/

25-6

The *COMPUTE!'s Gazette* subscriber list is made available to carefully screened organizations with a product or service which may be of interest to our readers. If you prefer not to receive such mailings, please check this box .



COMPUTE! Books

P.O. Box 5406 Greensboro, NC 27403

Ask your retailer for these **COMPUTE! Books**. If he or she has sold out, order directly from **COMPUTE!**

For Fastest Service
Call Our **TOLL FREE US Order Line**

800-334-0868
In NC call **919-275-9809**

Quantity	Title	Price	Total
_____	Machine Language for Beginners	\$14.95*	_____
_____	Home Energy Applications	\$14.95*	_____
_____	COMPUTE!'s First Book of VIC	\$12.95*	_____
_____	COMPUTE!'s Second Book of VIC	\$12.95*	_____
_____	COMPUTE!'s First Book of VIC Games	\$12.95*	_____
_____	COMPUTE!'s First Book of 64	\$12.95*	_____
_____	COMPUTE!'s First Book of Atari	\$12.95*	_____
_____	COMPUTE!'s Second Book of Atari	\$12.95*	_____
_____	COMPUTE!'s First Book of Atari Graphics	\$12.95*	_____
_____	COMPUTE!'s First Book of Atari Games	\$12.95*	_____
_____	Mapping The Atari	\$14.95*	_____
_____	Inside Atari DOS	\$19.95*	_____
_____	The Atari BASIC Sourcebook	\$12.95*	_____
_____	Programmer's Reference Guide for TI-99/4A	\$14.95*	_____
_____	COMPUTE!'s First Book of TI Games	\$12.95*	_____
_____	Every Kid's First Book of Robots and Computers	\$ 4.95†	_____
_____	The Beginner's Guide to Buying A Personal Computer	\$ 3.95†	_____

* Add \$2 shipping and handling. Outside US add \$5 air mail; \$2 surface mail.

† Add \$1 shipping and handling. Outside US add \$5 air mail; \$2 surface mail.

Please add shipping and handling for each book ordered.

Total enclosed or to be charged.

All orders must be prepaid (money order, check, or charge). All payments must be in US funds. NC residents add 4% sales tax.

Payment enclosed Please charge my: VISA MasterCard
 American Express Acc't. No. _____ Expires ____/____

Name _____

Address _____

City _____ State _____ Zip _____

Country _____

Allow 4-5 weeks for delivery.





Creating Your Own Game

You want to create your own games, but you're not sure just how to begin. After all, game writing can be one of the most demanding programming tasks. Whether you already know programming or are just starting, it may seem almost impossible to create a successful game. This book — a guide for anyone who has thought of designing and writing a game program — shows you how.

Robert Camp's *Creating Arcade Games on the VIC* is a step-by-step guide through the process of writing your own game on a home computer. From developing a game design concept to writing a complete game, you'll find every method and technique you need for creating VIC games. Detailed and clear explanations make it easy to follow along.

Here are just a few of the topics covered:

- Sound
- Custom characters
- Animation
- Missile graphics
- How to develop an idea
- How to market a game program

Along the way, you'll be able to follow the step-by-step creation of four games, and see how the principles work out in real programming situations.

Using an unexpanded VIC-20, you'll see the power of a home computer as a game-writing tool. With this book, and your imagination, you'll soon be writing that game you only hoped you could write.