CREATING ARCADE GAMES ON THE COMMODORE 64

A step-by-step guide to creating an arcade game on the 64,® plus six finished games to learn from and play.

Robert Camp

A **COMPUTE! Books** Publication

$14.95

CREATING ARCADE GAMES ON THE COMMODORE 64

COMPUTE!™ Publications, Inc.
One of the ABC Publishing Companies
Greensboro, North Carolina

Printed in the United States of America

ISBN 0-942386-42-6

10 9 8 7 6 5 4 3 2 1

COMPUTE! Publications, Inc., Post Office Box 5406, Greensboro, NC 27403, (919) 275-9809, is one of the ABC Publishing Companies and is not associated with any manufacturer of personal computers. Commodore 64 is a trademark of Commodore Electronics Limited.

Contents

Foreword

Many people who play arcade games take them for granted. But when you try to program such a game on your 64, you quickly learn to appreciate the amount of work and creativity behind every game. Characters and sprites on the screen, movement, collisions, sounds, and screen displays must be fit together into a complex, yet complete, whole. All this can be rather intimidating, especially if you're just starting.

This book takes you step by step through the process of designing and then programming a game. The principles and techniques of writing arcade games are outlined clearly and in detail, including examples from the six complete games provided in the book. The 64's sound, graphics, and sprite capabilities are fully demonstrated, and are shown in actual programming situations which you can duplicate.

You'll see how to develop a game from an idea and, using examples from successful arcade games, you'll learn what makes a good game. Techniques of setting up a playing field for your game are shown, as well as how to create the figures that will move across the screen. You'll even learn how to make the computer control game opponents while the player moves a figure with the keyboard or joystick controls. And the extras—such as sounds and graphic title displays—are not forgotten. As you read through the book, you'll actually create your own arcade-style game.

As with all COMPUTE! books, this guide to creating arcade games will be useful not just while you're writing your first game, but again and again. You'll refer to it for more information and more complex game ideas and techniques on the Commodore 64 as your skills improve. With this book and your creativity and imagination, you'll quickly be developing and writing games of a quality you never thought possible.

1
The 64 Is a Game Machine

The 64 Is a Game Machine

The best use of computers is playing games. If they can pay their own way by doing useful programs like accounting or word processing or math, so much the better. But I wouldn't care if they couldn't do any of those things, as long as they can create videogames, worlds that never before existed, where we can do things that we might never do anywhere else.

Flying on giant birds and jousting with lance-bearing enemies.

Saving the world from dozens of species of alien invaders.

Burrowing through the ground in pursuit of dragons.

Piloting starships, balloons, biplanes.

And really crazy things, like wearing a kangaroo suit and punching out monkeys, escaping from cute ghosts, climbing a skyscraper populated by homicidal clowns, rescuing a girl from a gorilla and then rescuing the gorilla from the rescuer—

After dumping a few hundred quarters into video arcade games, I decided it would be a better investment of my money 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 would end up in the arcade, too, among the others that I had enjoyed so many times.

I started with a VIC-20, and learned the hard way. Trying things out, finding out what worked and what didn't. I kept wishing for a single book that would tell me the things I needed to know to program games. I didn't find it. So when I finally figured it out, I wrote it myself, and called it *Creating Arcade Games on the VIC.*

Then I got a Commodore 64. Suddenly I had elbow room—enough memory to hold several character sets, sprites to make animation and movement easy, new features like multicolor characters and sprites, and a screen 40 columns wide.

The basic principles of game design are the same on all

computers. But the techniques for carrying them out differ from one machine to the next. Commodore 64 videogamers needed their own book on game programming, and with so many great features to work with, I had to remind myself that I was *working* as I wrote this book.

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 in creating a game is planning how you want the game to play. So in Chapter 2, you'll 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. How to make characters move on the screen and how to animate them. How to create sprites and move them with the joystick. 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.

Finally, since no one book can possibly include everything there is to know, there'll be a list of other books you can use to learn even more.

How to Use This Book

Try It Out

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 that you've already learned. But there are many things that 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.

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 learned only 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. Some things you'll grasp faster than I did; other things may take more effort. One thing is for sure—unless you're already an expert programmer, programming a game is going to take *time*. Not only the time of typing 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, you can'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, you'll have more than a few times 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 64 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 that readers can type them in and use them. At first

those programs look like a lot of gibberish to a beginning programmer. But the more you learn, the more those programs 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 redefines 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 that 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 that has slipped 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. Then 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 on the screen collide; how many points each goal should be worth; how many seconds or fractions of seconds each activity should take; and so on.

When it's all there, on paper, then 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 there are some ideas that 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, and takes as much creativity.

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 run-of-the-mill games and great games that people 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, then 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.

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? The surest sign of a successful game is that they don't want to quit. The surest proof that a game has 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.

What You Should Have on Hand

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

1. A Commodore 64.
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. *Commodore 64 User's Guide.*It came with your computer.
4. *Commodore 64 Programmer's Reference Guide* (published by Sams). It's available at most places that sell the computer, and extremely valuable in problem solving. (Other recommended books are listed at the end of this book.)
5. Plenty of cassettes or disks to save your programs. If you're using cassettes, 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 but 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 8×8 groups of tiny squares, like the example in Chapter 4.
7. A 6-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 another 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 64 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 Commodore 64

If you just got your 64 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.

Memory

The Commodore 64 comes with 64K. The *K* stands for kilobytes, which means 1000 bytes. But that's a rounded-off number. A kilobyte is actually 1024 bytes, and 64K is actually 64 of those, for a total of 65,536 bytes.

Human beings count memory in kilobyte units because "1000 bytes" sounds pretty close to our usual base 10 numbering. But the computer only rarely thinks in kilobytes. Instead, it naturally groups memory in *pages*. Each page consists of 256 bytes. And a 64K machine contains 256 pages: 256 times 256 equals 65,536.

Why is 256 the magic number? It comes from the size of a byte. Each byte consists of eight bits. Each bit is either a 0 or a 1. To the computer, the number 56 is a byte that looks like this: 00111000. In the decimal number 56, the 6 is the rightmost column, the ones column. The 6 means 6. But the 5 is in the *tens* column. The 5 doesn't mean 5, it means 50. The computer also uses columns, but the columns mean different things. The rightmost column is the ones column. The next one to the left is the twos column. Then come the 4s, 8s, 16s, 32s, 64s, and 128s. And the numbers you can find in those columns are pretty simple: either a 1 or a 0. Either there *is* a 64 in the number or there isn't. If there is, there'll be a 1 in the 64s column. If there isn't, there'll be a 0.

So the number 00111000 has no 1s, no 2s, no 4s. It has an 8 and a 16 and a 32. It has no 64s or 128s. When you add up the numbers 8, 16, and 32, you get 56.

Of course, the computer doesn't have to add up the digits of 00111000, any more than you ever think "56 is 6 plus 50." You see 56 and you understand what it means. The computer

sees 00111000 and it understands. It has as much trouble figuring out 56 as we have figuring out 00111000.

What is the lowest number a byte can hold? 00000000. And what is the highest number? 11111111. That's 128 + 64 + 32 + 16 + 8 + 4 + 2 + 1. It equals 255.

That's the range of possible numbers that can be contained in any one byte of memory. The numbers from 0 to 255.

Notice that this means a byte can have 256 possible values. 0 is as much a number as 255. So even though 255 is the *highest* value that can be contained in a byte, the byte can hold a *total* of 256 possible numbers, including 0.

Back to the reason why the computer thinks in pages. Every byte in memory has an address. That address must be expressed in the same numbering system. Fortunately, the 6510 microprocessor in your Commodore 64 uses *two* bytes to number the addresses.

The first address in the computer is the two-byte address 00000000 00000000. The byte on the left is the high byte; the one on the right is the low byte. The next address is 00000000 00000001. (Notice that the first address is 0, and address 1 is the *second* byte in memory. Computer users quickly learn to start counting everything with 0. Ask a programmer to count off the first eight numbers, and he'll say "0, 1, 2, 3, 4, 5, 6, and 7.")

Address 255 is binary 00000000 11111111. (Remember, address 255 is the 256th memory location.) Those first 256 bytes all had the same high byte: 00000000. Therefore, we say that they are all in the same *page* in memory. It is as if computer memory were a book with 256 pages (from 0 to 255), and each page has exactly 256 words (or *bytes*), and each word has exactly eight letters (or *bits*).

That's 256 pages of 256 words each. 65,536 bytes of memory, each with its own unique address. Your computer can't find any more addresses than that—it can't think of an address higher than 65,535 (remember, we started numbering at 0). Your 64 plays a few tricks, however, to squeeze in more memory.

RAM and ROM. But before getting to those tricks, let's distinguish between ROM and RAM. RAM is random access memory, and ROM is read only memory. You can use the PEEK function to find out what value is stored in the byte of memory at any address in the computer. You just say some-

thing like PRINT PEEK(65535) and BASIC goes and looks at address 65535 and finds out what number is stored there. It will always be a number from 0 to 255, because even though that location has a two-byte *address* (11111111 11111111), it is only one byte of *memory*.

PEEKing a byte doesn't change what's there—you look, but you don't change it, any more than your reading this book changes the letters on the page. So ROM and RAM can both be PEEKed. However, ROM cannot be POKEd. POKEing changes the value stored in a byte. You can use POKE to change any value in RAM: POKE 53281,5. But the ROM memory locations are permanent. They contain programs, and if you changed them you would wreck the programs. BASIC, for instance, is a program stored in ROM. So POKE doesn't work with ROM, except in the case of certain hardware registers, which you can POKE in order to give the 64 special instructions. For instance, the location 53281 is the register that controls the screen color. You can POKE any number from 0 to 15 into that location, and the screen color will change accordingly.

ROM sits on top of RAM. It uses up part of it. The character set, for example, is 16 pages of ROM starting at address 53248. You can't change the numbers stored there. So those addresses are effectively closed to you. The area occupied by BASIC, the Kernal, the SID sound chip, and the VIC-II video chip are likewise unavailable. But don't worry; you still have plenty of memory left to work with. And on special occasions, you can even switch out all the ROMs and get at every single byte in memory. However, that also switches out BASIC—you have to be working in machine language to use all 64K in your computer.

Blocks. The ROM, which you can't change, is busy carrying out all the business of the computer. But there are also areas of RAM that are involved with the operating system of the computer. For instance, there is an area called *screen memory*, where codes are kept to tell the VIC-II what to display in each location on the screen. Chapter 3 is all about how to use screen memory. What matters right now is the concept of *blocks*.

A block of memory is a group of bytes that have a temporary special meaning to the computer. Screen memory is a block of 1K that maps the screen (and performs some other functions with sprites and bitmapped screens). When you turn

on the computer it is located starting at address 1024, continuing to address 2047. (Only the first 1000 bytes are used to map the screen.) However, by POKEing a certain code into a certain register, you can tell the VIC-II video chip to look for screen memory at another location. In effect, you have moved the block. You haven't moved the *data* in that block, but you have moved the place where the computer will *look for* that data.

The character set is another block. You can tell the computer to look for that block in a different place, too.

However, blocks have certain restrictions. You can't just put them anywhere. For instance, screen memory has to be located on an even 1K boundary. That is, the starting address of screen memory has to be at an address that is evenly divisible by 1024. Obviously, the address 1024 is a valid address for the screen memory block. So is 0. So is 2048. And 3072, 4096, 5120, and so on.

The character set has to start on a 2K boundary. That means its starting address must be evenly divisible by 2048. Valid starting addresses include 0, 2048, 4096, and 6144.

The bitmap block must start on an even 8K boundary—valid addresses are 0, 8192, and 16384.

Each sprite pattern block must start on a 64-byte boundary, so there are a lot more possibilities: 8192, 8256, 8320, and 8384 are all legal.

There are even blocks within blocks. For instance, the video memory block has to start on an even 16K boundary. There are only four such boundaries possible in the 64: 0, 16384 (16K), 32768 (32K), and 49152 (48K). The video memory block must contain all of the blocks used by the VIC-II video chip. This is because the VIC-II, unlike the 6510, can handle only 14-bit addresses instead of 16-bit (two-byte) addresses. The two highest bits of a two-byte address mean nothing to the VIC-II. You can select which one of the four 16K blocks in the computer the VIC-II will use, but you can select only one at a time.

Within that 16K video memory block, you must locate all the other blocks related to video. If you move the character set, you must put that block on an even 2K boundary *within* the video memory block. Screen memory must be in that block. If you use a bitmap, all 8K of it must be in the video memory block.

When you turn on your 64, the video memory block is located at 0. Screen memory is within that block, at 1024. (Character memory is *not* actually in the block, since the ROM is at 53248. But the operating system is really fooling itself into thinking that the character set is at 10240, which is within the block. Don't worry about why.) However, a lot of other things are there. The locations from 0 to 1023 are heavily used by the operating system—you can't play with them or you'll crash the computer. Your BASIC program starts at 2048, and starts eating up memory fast. If you wanted to put an 8K bitmap there, that block could get awfully crowded. So if it gets *too* crowded, you can always select a different block of memory to serve as the video memory block, and then use it almost exclusively for video.

If this seems complicated, don't worry. It *is* complicated. But the more you use the 64, the more natural it'll seem. You'll be doing some pretty fancy things with memory before you get very far into this book. And you'll be doing it almost painlessly. I always figure that if it works, it's fine with me—whether I understand *why* or not!

Screen

The 64 screen is 40 columns wide and 25 rows from top to bottom. That gives you exactly 1000 individually controllable locations where you can put characters. There is also a bitmapped mode, in which every *dot* is individually controllable—all 64,000 of them! However, it is impossible to get any kind of speed on the bitmapped screen in BASIC, and since speed is vital to the arcade-style videogame, we won't be working with bitmapped screens in this book.

Sprites

The 64 allows eight sprites, which can be moved on the screen independent of screen memory. Each sprite takes its shape from a completely relocatable sprite shape block—which means you can change the sprite's shape instantly, with a single POKE, and its location almost as fast, with two or, sometimes, three POKEs. And when it moves across the background, the rest of the display is unaffected. This makes game programming far easier and more powerful than is possible on machines without sprites. However, we'll still go over character and screen movement and animation. As you'll see, sprites

don't *replace* character and screen animation—they merely enhance it.

Colors

With a color TV or monitor, the 64 can produce 16 different colors. However, just as important as the number of possible colors is the number of colors you can have on the screen at the same time. In standard color mode, you can select different colors for every character position on the screen, and a different color for each of eight sprites. Also, the background and border colors are separately controlled. This means that *thousands* of color combinations are possible.

In addition, sprites and screens can also be set up in *multicolor* mode, in which your program can display three different colors within the *same* character or sprite, in addition to the background color. You won't lack for dazzle in your games, if you want it!

Programmable Characters

You can redefine the shapes of characters if you want to, and sprites *have* to be defined in order to use them. By careful design and planning, you can get almost any shape you want to appear on the screen.

Sound

The 64's SID chip is one of its most notable features. You get the same kind of sound control available in music synthesizers, so that 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 64 gives you plenty of sound resources to work with.

Languages

The 64 has two languages easily available, BASIC and machine language.

Machine language consists of eight-digit binary codes that give instructions to the 6510 chip, which is the central processing unit (CPU), the real brain of the machine. The binary number 10000101 (decimal 133), when it's read as a machine language instruction, tells the CPU to store a certain value in a certain location—it works very much like a POKE.

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 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 slow only 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 with your BASIC programs, so that you can get the speed of machine language in a few places where you really need it. In fact, there are some machine language routines later in this book. But actually teaching machine language is beyond the reach of this book. Once you learn how to make the 64 play good games, it's up to you to decide what language to use.

The 6510

The 64 has a brain made of silicon. The CPU (central processing unit) is the 6510, which is very similar to the famous 6502 microprocessor that serves as the brains of the Apple, Atari, and other microcomputers. However, Commo-

dore isn't just *using* the 6510—they *own* both the 6502 and the 6510, which means they can offer the 64 at a lower price than if they had to buy the processor 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 64.

For one thing, Commodore BASIC is different in small but important ways from the BASIC used by the other machines. It's even different from the BASIC used by the VIC and the PET, though in fact it's often possible to take programs written for the VIC or PET and, by changing a few numbers, make it run on the 64.

Why can't BASIC programs and programs for the 6502 run without changes on all the computers that use that language and that processor? Because 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 your computer stores a 4, for instance, at location 1 in RAM, that tells the computer to switch on the input/output (I/O) register. That same location on the Atari is used to store a timer value. You'll be using location 53272 often with your 64, to tell the VIC-II chip where to find screen memory and character memory—but in the Atari, that same location is used to set one of the playfield colors. The equivalent playfield color on the 64 is set at location 53283. You can see why you can't just pick up a program from one computer and run it on another!

The only way for a BASIC program to be portable (easy to translate for another computer) is to stick to generally accepted BASIC commands and never PEEK and POKE into specific memory locations. Unfortunately for game designers, computers usually differ the most in graphics and sound handling, which are at the heart of 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

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 players understand what they're 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, rock-like 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*. The distinction is important in programming—you will use one kind of routine to animate, or change the *shape* of the character, and another kind of routine to move, or change the *location* of the character.

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. If you are doing that when you reach the gap between the two close-together islands on the right side of the screen, you'll be squished down through the gap and ejected on the lower side in an instant, not a bit the worse for wear—quite a remarkable feat!

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 awhile 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 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 top for rebounding—they fly quickly upward and then rebound downward so they come at you faster than you expect and from a direction you weren't prepared for. If you imitate 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, almost 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 a new 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.

Also, 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 said, "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. Then the islands come on all over again, but by then everything is moving so fast that they don't make it much easier.

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 (Gladiator Wave), in which you get bonuses for hitting each other, and cooperative play (Survival Wave), where the bonus comes 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. If 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 50 points for getting killed. I think this was inserted in the game so novices wouldn't get depressed at a score of 0 when the game ends.

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 coming 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. And as you scan the rest of the screen, the wing-flapping sound helps you keep track of your own knight's rhythm. You may not be conscious of the sound, but it's as much a part of the game as the video.

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 fantasty 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 (though we *will* see how to move and animate many characters at once). 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 by 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 hard-working hero Mario from *Donkey Kong,* or the magnificent flying mounts from *Joust.*

Characters

On your 64, 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 that you designed. To the computer, the letter A is just the screen code 1. It will treat the character the same no matter what shape you give it. Or you can combine several new characters to make part of a larger picture on the screen.

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. You have a lot of memory on the 64—but animation eats up memory fast.

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 only rolls barrels to one side, the programmer had to create only the shape for 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 or sprite definitions 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 causes a movement on the screen only 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 delays or 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. We'll go through several techniques for doing this in the chapter on animation.

And as you design your program, move as much of the program 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. There are some machine language routines in this book, too, that will help you achieve real speed. Still, don't despair if your first version of a program runs more slowly than you want. You'll soon learn tricks to let you 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.

There are other methods of checking for collisions, too—PEEKing into screen memory and checking the sprite collision register. 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 *best* 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 altitude 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 also you 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 that 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 the creatures 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 that they have to make decisions more quickly. And if opponents throw rocks or shoot bullets, those projectiles can move faster so that it's 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 practically have to *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 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 acomplish 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 be only 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 it feels 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.

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 64 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 doe saves her joey. In these games, you'll notice that the objective is often achieved—the girl, Papa, and the joey 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 only be done in machine language—you'll have to keep in mind that every use of sound 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 64, 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 some things in your plan that 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 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

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.

Visualizing what the video display will look like will help you as you design your own game. What will the player first see? Is the screen colorful; does it immediately draw the player into the game world? Does it create the impression of actually being underwater, in space, or tunneling beneath the ground? Imagining yourself as a first-time player of your game makes it easier to decide what should be included in a screen display.

Changing Colors

Let's start getting control of the screen display. There are 256 possible combinations of background and border colors on the Commodore 64. Instead of simply describing them, I'll show them to you. Just type in this simple program.

```
5 PRINT "{CLR}"
10 FOR X=0 TO 255:POKE 53281,X:POKE 53280,X:NEXT
20 GOTO 10
```

What does this program do? Location 53281 controls the background color, and location 53280 controls the border colors. This program changes the number stored at these locations by POKEing 256 different numbers there—all the values from 0 to 255.

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

Anything between braces—"{" and "}"—is the name of a special Commodore 64 character. In line 10, you type the word PRINT, then a quotation mark. But you don't then type {CLR}. Instead, you press the SHIFT key and the CLR/HOME key. In line 60, you don't type {2 DOWN} {RIGHT}. Instead, you type the CURSOR DOWN key twice and the CURSOR RIGHT key one time. Special characters will appear on your screen as you type these in when they are used within quotation marks. See Appendix B, "How to Type in Programs," for a full explanation of this.

All of the programs and many of the one-line listings include a checksum number at the end of each line. For example, in the program below, line 10 has the characters *:rem 197* at its end. That's the checksum number. These have been included to make it easier for you to enter the programs correctly. As you type in the programs in this book, you won't actually enter the checksum numbers; you'll simply use them to make sure the line was typed in correctly. Before you enter any of the programs in this book, read through Appendix J, "The Automatic Proofreader," for the short checksum program and description of how it operates.

Program 3-1. Screendemo

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 PRINT"{CLR}"                                        :rem 197
20 FOR BR=0 TO 15                                       :rem 83
30 FOR BG=0 TO 15                                       :rem 73
40 POKE 53280,BR                                        :rem 85
50 POKE 53281,BG                                        :rem 76
60 PRINT "{HOME}{2 DOWN}{RIGHT}BORDER COLOR="BR;"
   {LEFT} {2 RIGHT}BACKGROUND COLOR="BG"{LEFT} "
                                                        :rem 243
70 FOR T=0 TO 1000:NEXT                                 :rem 236
80 NEXT:NEXT                                             :rem 32
```

Now the screen changes more slowly, so you can see what's happening. Also, line 60 shows you the numbers being POKEd into both the border (BR) and background (BG) locations. Some of the color combinations are more attractive than others, while still others lend themselves better to text display. If you see a particular combination you like, just hit the RUN/STOP key and check the values on the screen. If you can't

make them out, you can press the RUN/STOP-RESTORE keys and then type:

PRINT BR

and/or

PRINT BG

and the last-used values will appear.

Notice that when the background value is 14, the numbers on the screen disappear. The numbers are still there, but they're invisible because they're the same color as the background. 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 53281—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 screen background seems to disappear. That means that you can put invisible objects on the screen, and then make them suddenly reappear by changing either their color or the background color. Magic—with a single POKE.

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

Program 3-2. OnOff

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 PRINT "{CLR}"                                    :rem 197
20 POKE 53281,14+13*(PEEK(53281)=254):GOSUB 30:GOT
   O 20                                             :rem 114
30 FOR I=0 TO 19:PRINT CHR$(94)"{DOWN}{RIGHT}";:NE
   XT:RETURN                                        :rem 211
```

Line 20 acts like an on-off switch. It POKEs 53281 with 14, unless the location already contains a value which will create that color, 254. In that case it POKEs 53281 with 1. But where does the 1 show in that line?

On-off switch. A good thing to remember with the 64, and many other computers, is that *false* has a value of zero and *true* has a value of negative one. So the expression (PEEK(53281) = 254) has a value of 0 if the location does not contain 254, and a value of -1 if 53281 *does* contain 254.

13 times 0 is 0—so if 53281 holds any number other than

254, the program POKEs it with 14+0, or 14.

13 times −1 is −13, however. So if 53281 *does* contain 254, line 20 POKEs it with 14−13, or 1.

You could have done it another way, using two program lines:

```
20 IF PEEK(53281)=254 THEN POKE 53281,1:GOTO 30
25 POKE 53281,14
```

In all programming, there's usually more than one way to do the job.

Default color What color does the screen usually display? Type RUN/STOP-RESTORE, to get the screen back to normal, and then type in PRINT PEEK(53281). The number 246 appears on the screen. That is the normal or default value of 53281—the number the 64 puts into that memory location once you turn it on. You'll notice that the value in the memory location is quite different than the numbers you POKE into that location to effect changes. Although you can use only the numbers from 0 to 15 to POKE new colors onto the border and background sections of the screen, the computer stores the values differently. To return a value between 0 and 15, you need to use the PEEK command like this:

PRINT PEEK(53281) AND 15

which will then eliminate all other values except those between 0 and 15.

Hide and Seek

Why did POKEing 53281 change the background color? Can one POKE do that much?

No. That POKE didn't actually change the colors. Color changes are complex operations, since the 64 has to change its entire signal to the television or monitor screen. That signal is handled by the Video Interface Chip II, which does all the communication between the computer and the screen. The VIC-II chip updates the screen display 60 times a second. You don't have to worry about it—it's done automatically. The important thing to remember is that the VIC-II chip looks at certain memory locations in order to find out what the screen should look like. 53281 is the location where the chip looks—every sixtieth of a second—to find out what the background color should be. 53280 is the location it looks at to find out the

border color. By putting a different number there, you are telling the chip what colors you want. The VIC-II chip takes care of the rest.

Screen Memory

This principle applies to everything about screen display and graphics. The VIC-II chip in the Commodore 64 looks at a number of locations to find out what you want on the screen. By changing what is in those locations, you can tell the computer to display exactly what you want. You only have to follow the rules.

Besides checking 53281 and 53280 to get the background and border colors, the chip also scans an area called *screen memory* to find out what characters to show on the screen, another area called *color memory* to see the foreground and background color of those characters, and still another area, called the *character set,* to find out what the character should actually look like. It checks other things, too, but we won't worry about them now. And remember, it checks all these areas 60 times a second.

By changing what the computer finds when it scans through these memory areas, you control what gets shown on the screen. You can change these locations using PEEKs and POKEs.

PEEKs and POKEs

If you're already familiar with the rules about using PEEKs and POKEs, 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 53281,15
```

Legal POKEs. POKE always has to specify an address that actually exists. That means that you can't POKE to a negative address or to an address higher than 65535. Also, you can only POKE into an address a number from 0 to 255. Negative numbers or numbers above 255 will give 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 in the number you're POKEing. If you told the 64

```
POKE 53281.5,14.2
```

it would act as if you had entered

```
POKE 53281,14
```

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 in your program:

```
POKE ADDR,NU
POKE C(I),N(I)
```

You can even use expressions, such as:

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

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

PEEK. You use this function to find out what is stored in a certain location, without changing the value already there. PEEK is not a command like POKE—it does nothing alone. If you enter PEEK(648), the computer won't know what to do. You also have to have a command that tells the computer to do something with the value it finds when it PEEKs at a certain memory location. The location you are PEEKing must always be in parentheses right after the word PEEK.

```
X=PEEK(648)
```

You can use PEEK with many different commands:

```
PRINT PEEK(1024)
IF PEEK(1024) = PEEK(2023) THEN N = PEEK(648)
FOR A = 0 TO PEEK(53281)
```

PEEK also works with expressions, like:

```
A = PEEK(INT(N/256)*256+X+5)
```

Whatever number results from the calculation becomes the address which is PEEKed.

PEEKing and POKEing in Screen Memory

POKE is the command you'll use to directly change one spot on the screen. You'll use the PEEK function to find out what is stored in a particular memory location in screen memory. Because the computer can't actually see what is on the screen,

this is the only way your program can find out, and know, what is being displayed at any given spot.

A sample program can show how this works. The value ADDR will be 1024. The variable X will be added to ADDR to select different locations on the screen:

Program 3-3. Screenfill

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 ADDR=1024                                    :rem 128
20 POKE 53281,1                                 :rem 241
30 FOR X=0 TO 1000                              :rem 115
40 POKE ADDR+X,INT(RND(X)*256):NEXT             :rem 102
50 FOR X=0 TO 1000                              :rem 117
60 POKE ADDR+X,32:NEXT                           :rem 61
```

The program is simple, yet it has quite an effect on the screen display.

Lines 30 and 40 put randomly selected characters on the screen. Lines 50 and 60 fill the screen with the same character you get when you press the space bar, a blank.

You'll notice that wherever there was writing on the screen before you typed in RUN, the characters are blue, and the inverse characters have a blue background. This is because you've done nothing to change color memory. That will come a bit later in this chapter.

Getting the Screen under Control

Changing the screen display is easy. The hard part is changing it exactly the way you want, to get the effect you want. We're getting there.

Codes. Notice that the screen doesn't display the *numbers* that resulted from the expression in line 40. Instead, it treats those numbers as codes for certain characters, such as letters, numbers, and symbols, and then prints the characters on the screen. These are not the same as the ASCII codes that you use with the CHR$ function. You get quite different results from these two statements, for instance:

```
PRINT CHR$(94)
POKE 1024,94
```

The CHR$ function uses the ASCII code values. Almost every computer understands the ASCII character codes—that's why

BASIC uses them, so that one program can be easily transported to another computer. But your 64's operating system has a harder time with them.

So you have two systems of code. This sample program shows the differences between them:

Program 3-4. Screencodes

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 ADDR=1024:S$="THIS IS A TEST"                    :rem 7
20 PRINT "{CLR}"S$                                 :rem 61
30 FOR X=1 TO 14                                   :rem 24
40 A=PEEK(ADDR+X-1)                                :rem 84
50 A$=MID$(S$,X,1)                                :rem 174
60 PRINT A;TAB(5);A$;TAB(8);ASC(A$)               :rem 207
70 NEXT X                                         :rem 254
```

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

But the differences are not purely random. You'll notice that the screen code is always 64 less than the ASCII code, except for the blank space, which is 32 on both lists. To be sure when you use either code in your game program, however, refer to the ASCII Code and Screen Code tables in Appendices F and G of this book.

Organization of Screen Memory

Computer memory is one long chain of addresses, one right after another. One section of this chain is used as screen memory. The 1000 bytes from 1024 to 2023 are used as a map of the TV or monitor screen. Each of the 1000 bytes represents one of the 1000 pixels on the screen. To see just one of these pixels, type this in:

{RVS} Space bar

and then hit RETURN two times. The small square now in the top-left corner is one pixel of the screen.

To convert that memory into 25 rows of 40 characters each, the VIC-II chip reads screen memory as if it were cut every 40

addresses. At every forty-first address, the VIC-II chip begins a new row on the screen. See the Screen Location Table in Appendix C.

In reading screen memory, the VIC-II chip starts at the upper-left corner, 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 gets to the lower-right corner of the corner of the screen, it jumps back up to the top-left corner and starts over.

The upper-left corner is the *lowest* address of screen memory (1024). The lower-right corner is the *highest* address of screen memory (2023). It may seem confusing that the top of the screen is lower in memory than the bottom of the screen, but it just means that the lowest-numbered address is at the top of the screen, and the highest-numbered address is at the bottom. If you remember that the 64 reads from left to right, from top to bottom, just as we do, and that the memory address follows the same order, it shouldn't be confusing. Appendix C shows the address of each screen location.

To change the screen display, the POKE command is used. A simple program, using a FOR/NEXT loop, can show each of the 1000 memory locations in screen memory:

```
10 POKE 53281,1
20 FOR X=1024 TO 2023:POKE X,83:NEXT
30 GOTO 30
```

As soon as you type in RUN, the screen will fill with the character whose screen code is 83, the heart-shaped figure. Each of the 1000 memory locations of screen memory will be filled with this character. You can place any of the 64's characters, or your own custom characters, into any of the memory locations in the computer's screen memory.

Setting Up a Screen

Let's say we want to draw a cross in the middle of the screen. The screen's center is character 19 of row 12 (counting from 0 for both). To find the exact address in memory of any location, including this one, multiply the row number (12) by 40, the total number of addresses per row. The answer is 480. Then add 19, since we want the twentieth character in that row (the first character is numbered 0). The answer is 499.

Program 3-5. Border

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 SC=1024:PRINT"{CLR}"                          :rem 153
20 POKE 53281,1                                  :rem 241
30 ROW=12:COLUMN=20:GOSUB 150                    :rem 242
40 COLUMN=COLUMN-1:GOSUB 150                     :rem 235
50 COLUMN=COLUMN+2:GOSUB 150                     :rem 235
60 ROW=ROW-1:COLUMN=COLUMN-1:GOSUB 150           :rem 178
70 ROW=ROW+2:GOSUB 150                            :rem 65
80 ROW=0:FOR COLUMN=39 TO 0 STEP -1:GOSUB 150:NEXT
                                                 :rem 155
90 COLUMN=0:FOR ROW=0 TO 24:GOSUB 150:NEXT:rem 252
100 ROW=24:FOR COLUMN=0 TO 39:GOSUB 150:NEXT
                                                  :rem 96
110 COLUMN=39:FOR ROW=24 TO 0 STEP -1:GOSUB 150:NE
    XT                                           :rem 251
120 GOTO 30                                       :rem 47
150 POKE SC+COLUMN+ROW*40,102:RETURN             :rem 222
```

If you type in only lines 10, 20, 30, 120, and 150, then RUN it, a small checkered square will appear at the center of the screen. (RUN this program each time you enter a new line to 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 one space to the left, unless you're already at the leftmost column.

```
40 COLUMN=COLUMN-1:GOSUB 150                     :rem 235
```

To add a character to the right, add 2 to the value of COLUMN. Why 2? You changed the value of COLUMN in line 40 to equal 19 (20−1=19), so to get to column 21, you must add 2 (19+2=21).

```
50 COLUMN=COLUMN+2:GOSUB 150                     :rem 235
```

Moving up a *row* is almost as simple. Remember that each row is 40 characters long. So the spot directly above the original character on the screen is 40 addresses earlier. However, in this sample program, the subroutine at line 150 takes care of multiplying by 40. So to move up, you only have to subtract 1 from ROW. You also need to get COLUMN back to its original value.

```
60 ROW=ROW-1:COLUMN=COLUMN-1:GOSUB 150           :rem 178
```

Now, to move down a row, you need only add:

```
70 ROW=ROW+2:GOSUB 150                            :rem 65
```

Borders

Putting a border around the screen is also quite easy, using the POKE command. First, to fill up the top row, you should type:

```
80 ROW=0:FOR COLUMN=39 TO 0 STEP -1:GOSUB 150:NEXT
                                              :rem 155
```

Now the left-hand margin:

```
90 COLUMN=0:FOR ROW=0 TO 24:GOSUB 150:NEXT:rem 252
```

The bottom row:

```
100 ROW=24:FOR COLUMN=0 TO 39:GOSUB 150:NEXT
                                              :rem 96
```

And the right-hand margin:

```
110 COLUMN=39:FOR ROW=24 TO 0 STEP -1:GOSUB 150:NE
    XT                                        :rem 251
```

(We used line 120 to keep the program running, so that the READY message didn't spoil the screen display.)

```
120 GOTO 30                                   :rem 47
```

You'll have to press RUN/STOP to end the program. Notice in lines 80 and 110 that the STEP command is used to draw the border from right to left, and from bottom to top, respectively. The STEP command is very useful when POKEing in new values to the screen memory when you want to make it create screen displays. Instead of drawing always from left to right, or top to bottom, the STEP command makes it seem as if the border, for instance, is racing around the edge of the screen.

Random Screen Displays

The screen you just created will appear the same every time you run the program. To design a screen that will be different every time, something very useful when designing a game, you need to do things a bit differently. For example, let's take a look at a screen which shows a different starfield each time it runs.

Program 3-6. Starfield

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 ADDR=1024:CL=55296:POKE 53280,0:POKE 53281,0:PR
   INT"{BLK}{CLR}"                            :rem 78
20 Q=100*RND(9)+20:Q1=6*RND(9)+2             :rem 202
30 FOR I=0 TO Q:X=1000*RND(9):N=46:GOSUB 150:NEXT
                                              :rem 20
```

```
40 FOR I=0 TO Q1:X=1000*RND(9):N=81:GOSUB 150:NEXT
                                                :rem 69
50 FOR I=0 TO 999:NEXT:PRINT "{CLR}":GOTO 20
                                                :rem 60
150 POKE ADDR+X,N:POKE CL+X,N:RETURN          :rem 236
```

This program creates a field of distant (small) and close (large) stars. It waits about a second before drawing an entirely different star pattern.

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

Line 20 is where the program decides how many far stars (Q) and near stars (Q1) there will be. There will always be at least 20 and never more than 120 distant stars, while there will be between 2 and 8 close stars. 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 screen address where a star will be placed. So the number of near stars and far stars and their placement on the screen are all random.

Controlling the random numbers. How do these random numbers work, and how are they created? The RND(n) function generates a random *fraction* between 0 and 1. The number in parentheses (called the *argument)* doesn't matter. You don't have to use the same number as in the examples. Your program then multiplies the random fraction to get a random number in the range you want.

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

Minimums. You can establish minimums to the range of random numbers by adding to it. A = INT (5*RND(5) + 3) will give you a number between 3 and 7.

A simple program like this will let you see this generation of random numbers:

```
100 A=INT(5*RND(9)+55):PRINT A,:GOTO 100
```

Hold the CTRL key down to slow the display. You can type RUN/STOP to end the program. Changing the 5 and 55 to other values will let you see other random numbers.

Changing the 9 has no effect, however. It is considered the *argument*, remember, and it has no effect on the function. Yet it must be there, because all functions (as opposed to commands such as POKE or PRINT) have to have a number in parentheses.

Random numbers are vital to most games, because they are the easiest way of making sure the game never plays the same way twice. You only need to set the appropriate minimums and maximums for the RND function to make it work for you and your game program.

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

Program 3-7. Fillin

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 ADDR=1024:POKE 53280,1:PRINT"{CLR}":POKE53281,1
   2:N=160                                    :rem 61
20 FOR X=0 TO 39 STEP 2:FOR Y=0 TO 23 STEP 2:GOSUB
    150:NEXT:NEXT                             :rem 42
140 END                                      :rem 108
150 POKE ADDR+X+40*Y,N:RETURN                 :rem 9
```

Variety. Just because you use a regular pattern in the playfield doesn't mean it has to be dull. You can make slight changes in your screen setup that will make a big difference on the TV or monitor. Adding these four lines to the program you just typed does that:

Program 3-7. Fillin, Cont.

```
30 READ N:IF N=0 THEN RESTORE:N=160           :rem 0
40 GOTO 20                                   :rem 255
150 POKE ADDR+X+1+(Y+1)*40,N:POKE ADDR+X+40*Y,N:RE
    TURN                                     :rem 165
200 DATA 102,81,78,77,86,127,255,79,80,0     :rem 189
```

Now the program READs a table of information in the DATA statement in line 200. Each new number causes a new character to print to the screen. You can experiment by adding new numbers to the table, as long as you make sure that the only 0 in the list comes at the very end.

You'll notice that the program keeps recycling through the characters, even though each is listed only once in line 200.

Line 30 is the key to this. The program READs the value of *N* from the list, remembers where it left off in the table and then READs the next value in order. When it sees a zero, the IF THEN statement notices and the RESTORE statement is executed. RESTORE simply puts the pointer back to the beginning of the list. The value of *N* is set to 160 again, and the entire loop starts over.

Many of the patterns conceal the fact that the screen is made up of 1000 little rectangles. Just because the 64 uses characters for its graphics doesn't mean you have to have rectangular-looking screens.

You can see how much variety comes from changing the screen color. Change the 12 in line 10 to a 1, or 7, and see how much difference it makes. Variety can come from many different sources when you're designing screens for your game.

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 new character there.

Here's a variation on the checkerboard program that shows how the PEEK function can be used. Consider it carefully, for this technique will be vital later, when you're figuring out whether a figure has bumped into something on the screen.

Program 3-8. Checkerboard

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 ADDR=1024:POKE53281,6:POKE 53280,1:PRINT"{CLR}"
   :POKE53281,1 :N=160:P=81                    :rem 9
20 FOR X=0 TO 38 STEP 2:FOR Y=0 TO 23 STEP 2:GOSUB
    150:NEXT:NEXT                              :rem 41
30 FOR X=0 TO 959:IF PEEK(ADDR+X)<>N THEN POKE ADD
   R+X,P                                       :rem 118
40 NEXT:READ N,P:IF N=0 THEN RESTORE:N=160:rem 246
50 GOTO 20                                      :rem 0
150 POKE ADDR+X+40*Y,N                         :rem 239
160 POKE ADDR+X+1+(Y+1)*40,N:RETURN             :rem 19
200 DATA 86,78,77,127,255,95,105,79,80,88,74,113,1
    12,32,0,81                                 :rem 159
```

You'll notice in line 30 that the program looks at the screen, from address 1024 to location 1983, but the *P* character is POKEd in only when that spot on the screen doesn't contain the current character *N*.

Multiple READs. In line 40, the program READs two numbers, *N* and *P*. This means that in the DATA table, the numbers for *N* alternate with the numbers for *P*. If you change this DATA statement, you need to make sure that your list has an even number of values, and that there is a zero in the second-to-last position.

Combining characters. You can see how the alternating characters can combine to create new shapes that do not exist in the 64's standard graphic character set. That's how you can make a larger picture for your game screen—combining several shapes to make one design. The graphics characters built into the 64 character set—screen codes 64–127 and 192–255, and ASCII codes 96–127 and 161–191 are designed so that many of them fit together this way to let you make continuous drawings easily.

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 at once. Setting up the PRINT statements takes more programming, and more typing, but once they're set up they can be PRINTed instantly on the screen. That is one of the values of the PRINT statement—its speed of execution. This program shows how quickly PRINTing a string can change the screen. Think of it as a wiring diagram, and imagine that you're controlling a spark of electricity trying to escape from the corner of the screen.

Program 3-9. Spark

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM A$(6),S$(8):POKE 53281,1:POKE 53280,1:PRINT
    "{CLR}{BLK}":GOSUB 500                    :rem 97
20 PRINT"{HOME}{DOWN}":FOR I=0 TO 8:PRINT A$(0):NE
   XT I                                       :rem 120
30 PRINT"{HOME}":GOSUB 150:PRINT A$(N+3):PRINT S$(
   8)+A$(6-N)                                 :rem 25
40 PRINT"{HOME}{DOWN}"+S$(S)+A$(0):GOSUB 160:GOSUB
    170:PRINT "{HOME}{DOWN}"+S$(S)+A$(N)      :rem 247
```

```
50 FOR I=0 TO 2000:NEXT:GOTO 30                    :rem 182
150 N=INT(2*RND(9)):RETURN                          :rem 65
160 N=INT(3*RND(9)):RETURN                          :rem 67
170 S=INT(7*RND(9)+1):RETURN                       :rem 169
500 FOR X=1 TO 38:A$(0)=A$(0)+CHR$(221):NEXT
                                                    :rem 30
510 FOR I=1 TO 5 STEP 4:FOR X=1 TO 19:A$(I)=A$(I)+
    CHR$(106)+CHR$(107):NEXT:NEXT                   :rem 0
520 FOR I=2 TO 6 STEP 4:A$(I)=CHR$(221):FOR X=1 TO
     18                                            :rem 207
525 A$(I)=A$(I)+CHR$(106)+CHR$(107):NEXT:A$(I)=A$(
    I)+CHR$(221):NEXT                              :rem 121
530 FOR I=0 TO 2:A$(I)=A$(I)+CHR$(32)+CHR$(32):NEX
    T                                              :rem 188
540 FOR X=1 TO 38:A$(0)=A$(0)+CHR$(221):NEXT
                                                    :rem 34
550 FOR I=1 TO 3 STEP 2:FOR X=1 TO 19:A$(I)=A$(I)+
    CHR$(117)+CHR$(105):NEXT:NEXT                   :rem 0
560 FOR I=2 TO 4 STEP 2:A$(I)=A$(I)+CHR$(221):FOR
    {SPACE}X=1 TO 18                               :rem 249
565 A$(I)=A$(I)+CHR$(117)+CHR$(105):NEXT:A$(I)=A$(
    I)+CHR$(221):NEXT                              :rem 125
570 S$(0)="{2 DOWN}":FOR I=1 TO 8:S$(I)=S$(I-1)+"
    {2 DOWN}":NEXT:RETURN                           :rem 5
```

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

Moving the Cursor with PRINT

If you're using PRINT statements to create your own 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 64. Once you've typed the quotation mark to start the string, pressing the HOME key or the cursor control keys will cause that character to be included in the string, instead of simply moving the cursor. When the string is PRINTed, the cursor will then move just as if you were pressing the cursor control keys yourself.

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 using CHR$(19). This moves the cursor to the upper-left corner of the screen. You can then 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 the cursor is on a 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 screen and counts *n* columns to the right. If the *n* in TAB(n) is a number greater than 39, TAB will move to the next line. A combination of HOME and TAB(n) can locate the cursor anywhere on the screen.

Another function, SPC(n), is sometimes used to move the cursor. The SPC(n) function starts counting from the current cursor position, and then moves *n* spaces to the right. This is not as valuable a function as TAB(n), since it is a relative, rather than an absolute, cursor-positioning command.

This program demonstrates several PRINT functions, including the HOME and TAB(n) functions, as well as the cursor control character within PRINT strings. Things such as game titles, scores, time remaining, and other game indicators lend themselves well to the use of the PRINT command. This program creates a simple game screen, complete with title, total score, and time remaining.

Program 3-10. Printscreen

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 PRINT"{CLR}":POKE 53281,0:TR=99          :rem 28
20 FOR X=1024 TO 1063:GOSUB 200:NEXT X      :rem 47
30 FOR X=1063 TO 2023 STEP 40:GOSUB 200:NEXT
                                            :rem 120
40 FOR X=2023 TO 1983 STEP -1:GOSUB 200:NEXT X
                                            :rem 214
50 FOR X=1984 TO 1024 STEP -40:GOSUB 200:NEXT X
                                            :rem 11
60 FOR X=1184 TO 1224:GOSUB 200:NEXT X      :rem 57
70 PRINT "{HOME}{DOWN}{WHT}";TAB(15) "SPACE GAME"
                                            :rem 244
```

```
80 PRINT "{DOWN}{2 RIGHT}{YEL}TIME REMAINING:"TR;"
   {5 RIGHT}{GRN}SCORE:"SC;"{WHT}"          :rem 201
90 TR=TR-1:IF TR=0 THEN TR=99               :rem 176
100 SC=SC+5:IF SC>205 THEN SC=0             :rem 192
110 FOR T=0 TO 1000:NEXT                     :rem 23
120 GOTO 70                                  :rem 51
200 POKE X,42:POKE X+54272,3                :rem 250
210 FOR T=0 TO 10:NEXT                      :rem 184
220 RETURN                                  :rem 116
```

Notice that the character colors were changed in lines 70 and 80. You've already seen how to alter screen colors using the POKE commands. Colors can also be changed with the PRINT statement, simply by entering a color key as part of the string to be PRINTed. Pressing CTRL and a color key, on the top row of the keyboard, changes all the following characters PRINTed to that color, until another color key is used. In line 80, for example, there are two color changes, from yellow to green.

Let's take a closer look at color on the 64, and especially color memory. It does have an important effect on what you see displayed on the screen.

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-II chip interprets the numbers in color memory as *color codes*.

Color memory is a perfect shadow of screen memory. For each of the 1000 locations in screen memory, there is a matching location in color memory that controls the color of whatever character is displayed on the screen. For example, the tenth byte of color memory controls the tenth character in screen memory. The color memory locations begin at 55296 and end at 56295. Appendix D shows each of the color memory locations in the 64.

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

Although you can change character color with the PRINT statements, as you've seen, it is usually better to POKE colors directly into color memory. Refer to the Screen Color Codes in

Appendix E for the numbers to POKE into color memory. You'll notice that they're the same values used to change the screen's background and border colors.

PEEKing and POKEing Color Memory

Color memory is a peculiar area of memory in the 64. 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 the color memory, the upper four bits of that number are chopped off. POKEing 255 into a color memory location would be the same as using the number 15.

However, if you PEEK at locations in color memory, you can easily get numbers higher than 15. The 64 is putting garbage in those upper four bits. To use the PEEK function correctly, then, you should AND the value you get with 15, to erase the garbage in the upper four bits:

```
COLOR=PEEK(LOCATION) AND 15
```

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

Finding Screen Memory

Relocatable programs. So far, we've been using the number 1024 as an *absolute* location for screen memory. (Remember that color memory always begins at location 55296.) But when you write a program, you need to be able to RUN it, no matter where screen memory begins. This is especially true when you switch banks on the 64, for instance when you're using bitmapping to create a high-resolution screen.

Instead of using 1024 as the screen memory, then, we'll write a short program line that finds out where screen memory is. Fortunately, the 64 always stores the address of the start of screen memory in the upper four bits of location 53272. To find screen memory, you could use a program line like this:

```
SC=(PEEK(53272)AND 240)*64
```

From then on, the variable SC (for SCreen) will contain the address of the upper-left corner of the screen. You could include this line at the beginning of a program to show this address.

If you've relocated screen memory (for whatever purpose), remember that you'll have to do a few more things

to make the computer work correctly. If you've switched video banks, you'll have to add the starting bank address to the value SC in order to show the true location of screen memory. For example, if you switched to the video bank which begins at location 32768, you would need to use:

```
SC=((PEEK(53272)AND 240)*64)+32768
```

to show screen memory's true location.

You'll also have to POKE a new value into location 648, where the 64's operating system looks to find where screen memory is. You do this by:

```
POKE 648,SC/256
```

This sets a pointer to let the 64 know where to look for screen memory.

On the Commodore 64, color memory is *always* at location 55296. It never changes. If you use the formula CM=55296, CM will always equal the address of the upper-left corner of color memory. To show both screen memory and color memory in one program line, then, you could use:

```
SC=(PEEK(53272)AND 240)*64:CM=55296
```

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.

Program 3-11. Colorpeek

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
900 SC=(PEEK(53272)AND 240)*64:CM=55296     :rem 69
910 FOR J=0 TO 15:POKE 53281,J:POKE 53280,1:PRINT"
    {CLR}"                                  :rem 137
920 FOR I=0 TO 1000:N=INT(RND(9)*8):POKE SC+I,N+48
    :POKE CM+I,N                            :rem 216
930 NEXT:NEXT:GOTO 910                      :rem 97
```

Line 900 establishes the values of CM and SC. Line 910 sets the background color to each of the possible colors. 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 appropriate location in color memory (CM+I). The same variable, *I*, leads to the corresponding locations in both screen and

color memory.

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

To see this working with graphics characters, try adding these lines to Program 3-8, Checkerboard.

Program 3-12. Checkerboard, Cont.

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 ADDR=(PEEK(53272)AND 240)*64:CM=55296  :rem 146
15 POKE53281,1:POKE 53280,1:PRINT"{CLR}":N=160:P=8
   1                                      :rem 231
25 GOSUB 300                              :rem 122
40 NEXT:READ N,P:IF N=0 THEN RESTORE:N=160:rem 246
45 GOSUB 300                              :rem 124
300 Q=INT(RND(9)*8):IF Q=1 THEN 300       :rem 119
310 IF Q=PEEK(CM) AND 7 THEN 300          :rem 131
320 FOR I=0 TO 959:POKE CM+I,Q:NEXT I:RETURN
                                          :rem 66
```

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

Or you can start again with the original version of Checkerboard, and add these lines instead:

Program 3-13. Checkerboard, Again

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 ADDR=(PEEK(53272)AND 240)*64:CM=55296  :rem 146
15 POKE53281,1:POKE 53280,1:PRINT"{CLR}":N=160:P=8
   1:GOSUB 300                            :rem 52
40 NEXT:READ N,P:IF N=0 THEN RESTORE:N=160:rem 246
300 FOR I=0 TO 959:N=INT(RND(9)*8):POKE CM+I,N:NEX
    T I:RETURN                            :rem 14
```

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. Characters that used to fit together to make combined patterns are now clearly separate.

Moving On

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 color and color memory and how to use them, it will be harder to make use of the more advanced techniques shown in the rest of the book.

4 Custom Characters

Custom Characters

The 64 comes with a lot of shapes you can put on the screen just by pressing keys or using PRINT statements in your program. So far, the only characters we've used to make shapes on the screen are graphics characters. Graphics characters are shapes that don't mean anything. They aren't letters, numbers, or symbols like %, *, $, or <. The only reason Commodore included them with your computer was so you could make shapes. Good games can be programmed using those shapes alone.

Sometimes, however, the graphics characters just aren't enough. There is no single character that looks like a human being, for instance. Or a dog. Or a train. Fortunately, the 64 is flexible enough to let you create your own characters.

How Characters Get Their Shapes

When you press a key on the 64, a character (letter, number, or symbol) appears on the screen. Press another key, and another character appears just to the right of it, or on the next line.

On a typewriter, the same thing happens. Press a key, and there's a character on the paper. Whenever you press *A*, you get an *A* on the paper. And it's always the same *A*, because the shape of that character is always there, stored as a pattern of raised metal, ready to be slammed through the ribbon into the paper.

The 64 doesn't use all that noisy machinery, but it *does* store the character pattern, and when you press a key, the 64 goes to the right pattern, takes it to the right location on the screen, and displays it there.

Character Memory

The area in memory where the character patterns are stored is called *character memory* or the *character set*. The built-in character set takes up 4K of memory, from 53248 to 57343.

How big is it really? Character sets don't really take that much space. The built-in character set is actually *two* character sets. You switch from one to the other when you press SHIFT-Commodore. Each of those character sets is only 2K, and 2K is the largest character set that the 64 can address at any one time.

That 2K set is even smaller than it looks. The first 1K is the

regular character set; the second 1K consists of the *reversed* character set—the characters that are displayed when you press keys after pressing CTRL-9. So the basic character set consists of 1K, 1024 bytes of memory.

Finding the pattern you want. The first 512 bytes are the patterns for the letter, number, and symbol characters—the characters with screen codes from 0 to 63. The next 512 bytes are the patterns for the graphics characters, with screen codes from 64 to 127.

The character patterns are organized in the same order as screen codes. The pattern for the character with screen code 0, the @ character, is the first one in character memory; the pattern for A, with screen code 1, is next; then the pattern for B, with screen code 2, and so on.

This means you can use the screen code as an *index* into the character set. In fact, that's just what the VIC-II chip does. When it reads a number in screen memory (in a text mode, of course, not in a bitmapped mode), it goes to the starting address of character memory, counts in the correct number of bytes, and displays whatever pattern it finds there.

This relationship between screen codes and character patterns holds up even when you have changed the pattern for a character. The 64 doesn't care whether an *A* looks like an *A*. If it finds screen code 1, it will display character pattern 1, even if you have redefined it to look like an alien invader.

How far to count. Each character pattern is eight bytes long. This means that when the VIC-II is looking for the character pattern, it looks for the pattern to start at the start of character memory plus *eight times* the screen code. This means that the pattern for *D*, with screen code 4, starts at byte 8*4, or 32, in character memory. The pattern for the question mark, screen code 63, starts at 8*63, or byte 504 of character memory.

The Character Patterns

Each character pattern is eight bytes long. The character pattern is an exact map of what the character is shaped like on the screen.

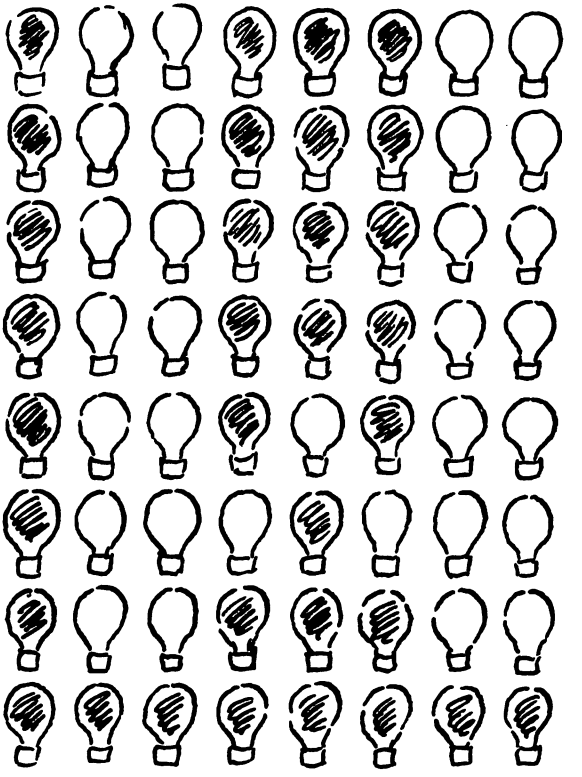
If you look closely at the TV screen (and if you have good eyes and your TV has good resolution), you'll see that each individual character is made up of little dots, arranged in rows and columns. Each character is eight dots wide and eight dots high. That makes exactly 64 dots. By an amazing coincidence,

each byte of the eight-byte pattern contains eight bits. That makes 64 bits in the pattern.

A lighted billboard. Think of the character on the screen as a signboard with 64 light bulbs, as shown in Figure 4-1. Some of the bulbs are on, and some are off.

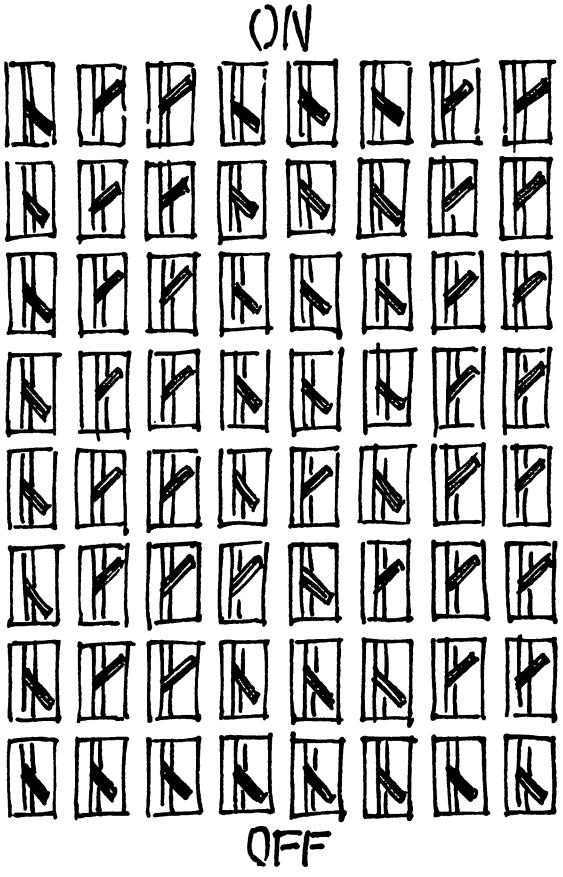
Think of the character pattern in character memory as a series of switches, as shown in Figure 4-2. When a switch is on, the corresponding light bulb is turned on. When a switch is off, the bulb is off.

Figure 4-1. Character Patterns



Each byte of the character pattern contains eight switches, corresponding to a row (horizontal line of dots) in the character pattern. The top row is controlled by byte 0 of the character pattern. The bottom row is controlled by byte 7.

Figure 4-2. On-Off Switches



So to find the byte controlling row 3 (the fourth row from the top) of the *D* character (screen code 4), with the built-in character set starting at 53248, you would PEEK at location 53248 + 8*4 + 3. The formula is:

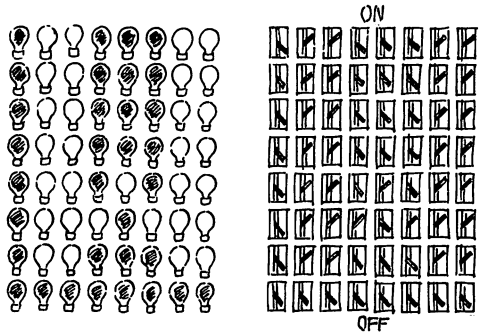
character memory + 8 * screen code + byte number

***On* bits and *off* bits.** How are the switches turned off or on? Each bit in the byte can be either a 1 or a 0. If it is a 0, the switch is off, and the background color is displayed at that dot in the character's pattern. If the bit is a 1, the switch is on, and the foreground color (the color assigned for that position by its

byte in color memory) is displayed at that dot. Figure 4-3 shows the bit patterns (and their decimal equivalents) for each byte of our lighted signboard.

Figure 4-3. Bits of the Character Pattern

Decimal	Binary
99	01100011
99	01100011
99	01100011
99	01100011
107	01101011
119	01110111
99	01100011
0	00000000



Here's a program that lets you look at the bit pattern for any character in character memory:

Program 4-1. Patterns

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
5 DIM A$(7),X$(1),A(7)                          :rem 27
10 VM=0:CH=53248                                :rem 121
15 X$(0)="P":X$(1)="{RVS} {OFF}":C$="{8 T}"
                                                :rem 75
97 REM                                          :rem 84
98 REM MAIN LOOP                                :rem 180
99 REM                                          :rem 86
100 GET A$:IF A$="" THEN 100                    :rem 69
110 A=ASC(A$):Q=A:GOSUB 200:GOSUB 300           :rem 63
120 PRINT B$:PRINT A$:FOR I=0 TO 7:PRINT A$(I);A(I
    ):NEXT:PRINT C$                             :rem 230
130 GOTO 100                                    :rem 94
197 REM                                         :rem 133
198 REM CONVERT ASCII TO SCREEN CODE            :rem 142
199 REM                                         :rem 135
200 IF A>63 AND A<96 THEN A=A-64:RETURN         :rem 103
210 IF A>95 AND A<128 THEN A=A-32:RETURN        :rem 148
220 IF A>159 AND A<192 THEN A=A-64:RETURN       :rem 204
230 IF A>127 AND A<160 THEN A=A+64:RETURN       :rem 193
240 IF A<32 THEN A=A+128:RETURN                 :rem 213
250 IF A>191 AND A<255 THEN A=A-128:RETURN:rem 252
```

```
260 IF A=255 THEN A=94:RETURN                          :rem 117
297 REM                                                :rem 134
298 REM MAKE STRINGS TO PRINT PATTERN                   :rem 29
299 REM                                                :rem 136
300 B$="SCREEN CODE "+STR$(A)                           :rem 47
310 IF A<128 THEN T=A:GOSUB 400:A$=A$+"{2 SPACES}C
    HR$("+STR$(T)+")"                                  :rem 232
320 IF A>127 THEN T=A-128:GOSUB 400:A$="REVERSE-"+
    CHR$(T)+"{2 SPACES}CHR$("+STR$(Q)+")"              :rem 168
330 X=8*A:GOSUB 800:GOSUB 700                          :rem 113
390 RETURN                                             :rem 124
397 REM                                                :rem 135
398 REM CONVERT SCREEN CODE TO ASCII                   :rem 144
399 REM                                                :rem 137
400 IF T<32 THEN T=T+64:RETURN                         :rem 219
410 IF T>63 AND T<96 THEN T=T+32:RETURN                :rem 175
420 IF T>95 THEN T=T+64:RETURN                         :rem 232
430 RETURN                                             :rem 119
697 REM                                                :rem 138
698 REM TAKE APART PATTERN BYTES                       :rem 205
699 REM                                                :rem 140
700 FOR I=0 TO 7:A$(I)="":NEXT                          :rem 65
710 FOR I=0 TO 7:W=A(I):FOR J=0 TO 7                   :rem 106
720 W=2*W:IF W>255 THEN W=W-256:A$(I)=A$(I)+X$(1):
    GOTO 740                                           :rem 106
730 A$(I)=A$(I)+X$(0)                                  :rem 253
740 NEXT J:NEXT I:RETURN                                 :rem 0
797 REM                                                :rem 139
798 REM GET PATTERN BYTES                               :rem 17
799 REM                                                :rem 141
800 POKE 56334,PEEK(56334)AND 254:POKE 1,PEEK(1)AN
    D 251                                              :rem 185
810 PT=CH+X:FOR I=0 TO 7:A(I)=PEEK(PT+I):NEXT
                                                       :rem 146
820 POKE 1,PEEK(1) OR 4:POKE 56334,PEEK(56334)OR 1
                                                       :rem 137
830 RETURN                                             :rem 123
```

Changing Shapes

Before you can change the shape of a character, you have to move character memory. This is because the built-in character set is ROM, read only memory. The bits and bytes are permanently burned into the chip. You could POKE there all day and not a shape would change. You have to locate character memory somewhere else.

Moving Character Memory

Fortunately, a single POKE will tell the 64 to look for the char-

acter memory block in a different place. The operating system uses location 53272 as a pointer to the start of the character memory block. There are certain rules to follow in locating character memory.

Character memory must be within the video block. The video block is the 16K that the VIC-II chip can address at any one time. Just like screen memory, the bitmap, and the sprite shape tables, character memory must be located within that block. When you first switch on your 64, the video block is the 16K from 0 to 16383. The VIC-II will look for character memory within that block.

The only exception is the ROM character set. The 64 is designed so that even though the character ROM is at 53248, the VIC-II *thinks* that it's finding character memory at 4096 when the video block starts at 0. You can't fool the machine that way. Your character patterns have to be where the VIC-II expects to find them, or it won't see them at all.

Character memory starts on a 2K boundary. Since a full character set (including reversed characters) takes 2K, there are only eight possible blocks within the 16K video block where character memory can be located—at byte 0 of the video block, or byte 2048, 4096, 6144, 8192, 10240, 12288, or 14336. In other words, character memory has to start at a 2K boundary—the 0K, 2K, 4K, 6K, 8K, 10K, 12K, or 14K boundary.

If the video block starts somewhere other than address 0, then the above addresses must be *added to* the video block address to find the actual address of character memory. For instance, if the video block starts at 32768 (the 32K boundary), and the 12K boundary is selected for the start of character memory, then the real address of character memory is 32768 plus 12288, or 45056. That's where you would POKE the patterns for our custom character set. But you would still POKE the number 12 into 53272, since character memory is still at the 12K boundary *within* the video block.

Use only the lower four bits of 53272. Only bits 0-3 (the lower, or rightmost, four bits) of the byte at 53272 control character memory. The upper four bits control screen memory. So when changing the location of character memory, you must protect the screen memory information.

If you wanted to tell the VIC-II to find character memory at the 12K boundary, you would not just POKE 53272,12. The number 12 in binary is 00001100. The lower nybble (four bits) is

1100, the code for the 12K boundary for character memory. But the upper nybble is 0000, and that tells the VIC-II to find screen memory at block 0. That's fine, if screen memory is actually at block 0. But when you turn on the 64, screen memory is at 1024, screen memory block 1. And if you want it to stay there, you can't POKE 53272,12.

You can protect the high (leftmost) nybble by using this formula to POKE a character memory code into 53272. To put character memory at the 6K block, you would use this statement:

```
POKE 53272, (PEEK(53272)AND 240) OR 6
```

Likewise, when you are relocating screen memory and you don't want to change the location of the character set, you must protect the lower nybble. If you wanted to locate screen memory at block 11 (screen memory can be located at 1K boundaries), you would use this statement:

```
POKE 53272,(PEEK(53272)AND 15) OR 176
```

(176 is 11*16. To turn a number from 0 to 15 into a high nybble value, just multiply it by 16.)

Be careful where you put character memory. If the video block is at 0, you've got to be aware of all the other things using that section of RAM. For instance, you wouldn't want to locate the character set at 0, because 0–1023 is used by the operating system and BASIC for vital operations, and 1024–2047 is screen memory. You wouldn't want to put character memory at block 2, either, because that's the start of your BASIC program. In fact, the best place for a character set when the video block starts at 0 is at the 14K boundary, so that there'll be a full 12K, from 2048 to 14335, for your BASIC program.

Moving the Video Block

If your BASIC program is going to run longer than 12K, you will need to move the video block. This is a bit trickier than moving character memory. First, you have to tell the VIC-II which 16K block to look at. Then you have to relocate screen memory, character memory, and, if you're using them, the bitmap and sprite shape blocks within the video block.

The video block codes. The VIC-II gets its video block instruction from location 56576. The video block is assigned with the following codes:

code: 0—video block starting location: 49152
code: 1—video block starting location: 32768
code: 2—video block starting location: 16384
code: 3—video block starting location: 0

It takes two steps to tell the VIC-II to locate the video block at 32768 (code 1). First, turn on bits 0 and 1 of location 56578:

```
POKE 56578,PEEK(56578) OR 3
```

Then POKE the video bank code into 56576. Since location 56576 also controls other functions, the rest of the byte needs to be protected by using this routine.

```
POKE 56576,(PEEK(56576)AND 252) OR 1
```

The video bank is now set to start at 32768.

Reconfiguring video memory. Now that the video block has been moved, you'll want to make sure that screen memory, character memory, and other blocks are in safe locations. For instance, if the video block starts at 32768, you have to avoid the entire upper 8K of the block, from 40960 to 49151, because that's where the BASIC ROM sits. That's no problem in text modes—the remaining 8K gives you room for two character sets (2K each), two screen memory blocks (1K each), and 32 separate sprite shape blocks (64 bytes each). The ability to have multiple screen memory blocks and character memory blocks is very useful, for reasons that will be explained in Chapter 5, on character animation.

However, if you're using a bitmap, you're in trouble because the bitmap alone uses 8K. Also, any additional cartridge will use 32768 to 40959—wiping out the rest of the block. (You'll almost never have a cartridge in place when running a BASIC program.)

If you use the block starting at 49152 (code 0), you have even more ROMs to cope with—everything above 53247 is used for ROMs (I/O, Kernal, Character ROM, Color RAM). But that still leaves you 4K from 49152 to 53247—enough room for one character set, one screen memory, and 16 sprite patterns.

The block at 16384 is completely free of ROMs, so you can use any of it—when you're using a bitmap, you almost have to put it here. However, you still have the problem of large BASIC programs creeping up from their starting point at 2048 until they start spilling into this block. If your BASIC program is

large enough that you have to move the video block at all, you'll probably want to move the video block even higher than this just to make sure it doesn't get in the way again.

Conventional variable names. For the rest of this chapter and the next, we'll assume that the variable VM contains the starting address of the video block, and the variable CH contains the starting address of character memory *within the video block*, and SC contains the starting address of screen memory *within the video block*. That is, if the video block starts at 32768 and character memory starts at the 6K boundary within the block, the variable VM would contain 32768 and the variable CH would contain 6144. To find the *absolute* address of character memory, you would add VM and CH.

If there are two or more character sets, the array variable CH(*n*) will be used to express their addresses.

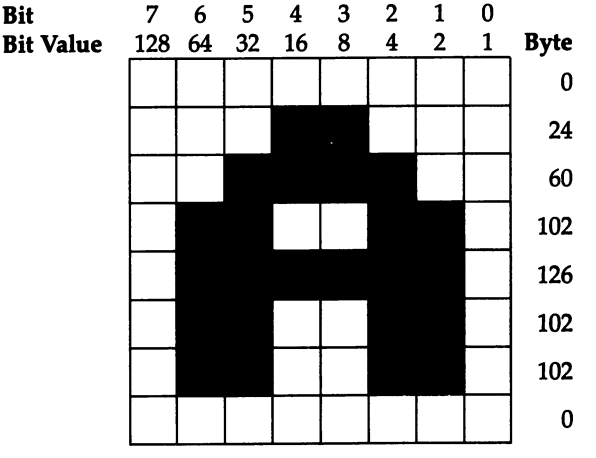
This is so that all the sample programs will work regardless of how you have configured video memory. If you are using the video block at 0, you can omit the variable VM from these routines.

New Shapes

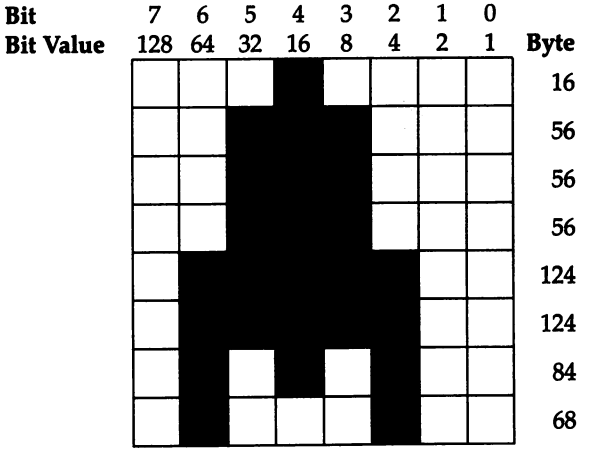
Now that you have relocated character memory (and, if necessary, the video block), you're ready to start putting in your own character shapes. In fact, you *have* to put in your own character shapes. If you have tried out these POKEs, you've already discovered that when you move character memory, the VIC-II starts reading whatever's there, even if it's garbage or nothing at all—it doesn't care whether there's a sensible pattern or not; it just counts in the right number of bytes from the start of character memory and displays whatever bit pattern it finds there.

The shape matrix. So let's get a few shapes ready to display. Figure 4-4 shows four characters on character grids. The shaded squares represent 1 bits; the unshaded squares represent 0 bits. The numbers above the columns on the grids represent the value of an *on* (1) bit in that position. To find the decimal value of each byte, just add up the values of the *on* bits in each row. Those are the numbers to put in DATA statements, in order from top to bottom.

Figure 4-4. Patterns on the Character Grid



Letter A



Rocket

Bit	7	6	5	4	3	2	1	0	
Bit Value	128	64	32	16	8	4	2	1	Byte
	☐	☐	☐	☐	☐	☐	☐	☐	0
	☐	☐	☐	☐	☐	☐	☐	☐	0
	☐	☐	☐	☐	☐	☐	☐	☐	0
	☐	☐	☐	■	■	☐	☐	☐	24
	☐	☐	■	☐	☐	■	☐	☐	36
	■	■	■	■	■	■	■	■	255
	■	■	■	■	■	■	■	■	255
	☐	■	■	☐	☐	■	■	☐	102

Car

Bit	7	6	5	4	3	2	1	0	
Bit Value	128	64	32	16	8	4	2	1	Byte
	■	☐	☐	■	■	■	☐	☐	156
	☐	☐	■	■	☐	■	■	☐	54
	☐	■	■	☐	☐	☐	■	■	99
	■	■	☐	☐	■	☐	☐	■	201
	■	☐	☐	■	■	■	☐	☐	156
	☐	☐	■	■	☐	■	■	☐	54
	☐	■	■	☐	☐	☐	■	■	99
	■	■	☐	☐	■	☐	☐	■	201

Bent Stripes

Figure 4-5. Blank Character Grid

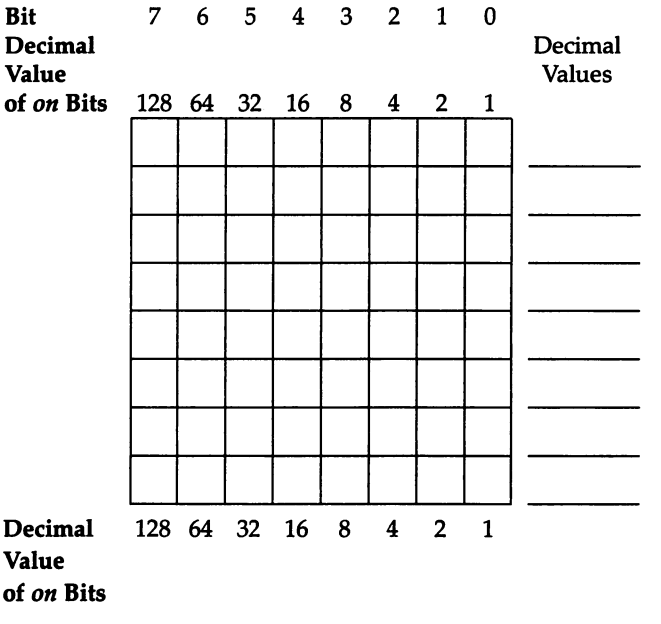


Figure 4-5 is a blank character grid that you can photocopy and use to create your own characters. Just shade the dots you want to have *on;* then calculate the sums of the *on* bits for each byte and arrange them in DATA statements.

From DATA statements to character memory. Once you have the DATA statements for each character, your program must READ the values and POKE them into the right place in memory.

What is the right place? It depends on which screen code you want the custom character to have. For instance, if you want the new character to replace @, which has a screen code of 0, you would POKE the pattern at VM+CH+0. If you want the new character to replace the left bracket ([), which has a screen code of 27, you would POKE the pattern at VM+CH+8*27. If the variable CS contains the screen code, this formula will always find the right starting address for the character pattern:

POKE VM+CH+8*CS, *character pattern data*

Of course, a single POKE won't do the job—it takes eight POKEs, since there are eight bytes in a character pattern. You will use a loop to put the whole character pattern in place:

```
FOR I=0 TO 7: READ N: POKE VM+CH+8*CS+I, N: NEXT
```

This loop will run slowly, since you are performing three additions and one multiplication each time through the loop. The same thing will go faster if your loop looks like this:

```
X=VM+CH+8*CS: FOR I=X TO X+7: READ N: POKE I, N: NEXT
```

If you are creating patterns for several characters in a row— for instance, the left bracket, British pound, right bracket, up arrow, and left arrow (screen codes 27–31)—you can use a nested loop:

```
X=VM+CH:FOR I=X+27*8 TO 31*8: FOR J=I TO I+7:
READ N: POKE J, N: NEXT: NEXT
```

Of course, you must make sure your DATA statements are in the right order. That is, that DATA for screen code 27 must come before the DATA for screen code 28.

Keeping part of the ROM character set. What if you want some custom characters, but you also want to have the regular letters and symbols? Once you move character memory, you have to provide the patterns for every character you use. But you *don't* have to have all the character patterns in DATA statements. You can also copy some or all of the built-in character set into your new character memory. You just PEEK at each byte of the pattern in the character ROM and POKE that byte, in the same order, into the new character memory.

Since copying hundreds of bytes from one location to another takes a long time in BASIC, you probably don't want to copy more of the ROM character set than you're actually going to use. You *can* copy the entire 2K of one set, or even copy both built-in sets—but the person playing your game will have a long wait while your program PEEKs at the number in the built-in set and POKEs it into the new character memory location. Remember, this operation is carried out eight times for each character, and there are 256 characters in each complete character set. That's 2048 PEEK and POKE pairs, plus the time it takes to calculate the different addresses within the loop.

So you'll usually want to copy only the portion of the character set that you want to use. If you aren't going to use

reversed characters, that eliminates the upper half of the character set right away. If you aren't using any of the graphics characters, you don't need to copy any of the next quarter. To get all the letters and symbols from ROM into your new character memory block, you need to copy only the first 64 character patterns. That's only 512 times through the loop—a lot faster.

And once you have those 64 characters copied, you still have room for 192 custom characters.

Copying part or all of the character ROM requires a couple of operations, however. Usually the character ROM is switched out, so that you can't PEEK into it. This is because it uses the same addresses as the I/O ROMs. It saves you 4K of memory, but it means that you have to turn off the I/O and switch in the character ROM, then copy what you want to copy, and finally switch out the character ROM and turn the I/O back on. These statements perform those operations:

```
POKE 56334, PEEK(56334) AND 254: POKE 1, PEEK(1)
AND 251
loop to copy character set
POKE 1, PEEK(1) OR 4: POKE 56334, PEEK(56334) OR 1
```

It is very important that you perform these operations in the order shown.

Here is a complete program that copies the first 64 characters of the character set and locates them at the 14K boundary in the video block, which is assumed to begin at 0.

Program 4-2. Move Character Set

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 VM=0:CC=14:CH=VM+CC*1024:RM=53248-CH:GOSUB 1100
                                          :rem 106
1100 POKE 53272,(PEEK(53272)AND 240)OR CC :rem 122
1110 POKE 56334,PEEK(56334)AND 254:POKE 1,PEEK(1)A
     ND 251                               :rem 228
1120 FOR I=CH TO CH+511:POKE I,PEEK(I+RM):NEXT
                                          :rem 140
1130 POKE 1,PEEK(1) OR 4:POKE 56334,PEEK(56334)OR
     {SPACE}1                             :rem 180
1140 RETURN                               :rem 166
```

And here is a modification of the same program, which copies the first 64 characters, but then replaces characters 27

through 31 with five custom characters. After you run this program, you will be able to see the custom characters on the screen by typing the keys for the left bracket, British pound, right bracket, up arrow, and left arrow.

Program 4-3. Move and Customize

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 VM=0:CC=14:CH=VM+CC*1024:RM=53248-CH:GOSUB 1100
   :GOSUB 1200                                    :rem 231
99 PRINT "{CLR}":FOR I=32 TO 95:PRINT CHR$(I);:NEX
   T:END                                           :rem 26
1100 POKE 53272,(PEEK(53272)AND 240)OR CC :rem 122
1110 POKE 56334,PEEK(56334)AND 254:POKE 1,PEEK(1)A
     ND 251                                        :rem 228
1120 FOR I=CH TO CH+511:POKE I,PEEK(I+RM):NEXT
                                                   :rem 140
1130 POKE 1,PEEK(1) OR 4:POKE 56334,PEEK(56334)OR
     {SPACE}1                                      :rem 180
1140 RETURN                                        :rem 166
1200 FOR I=CH+8*27 TO CH+8*31 STEP 8:FOR J=I TO I+
     7:READ N:POKE J,N:NEXT:NEXT                    :rem 70
1210 RETURN                                       :rem 164
1300 DATA 12,12,9,118,8,24,36,72                   :rem 27
1310 DATA 16,16,56,124,254,146,56,40              :rem 232
1320 DATA 16,124,248,248,124,147,127,62           :rem 131
1340 DATA 170,0,242,146,159,255,126,109           :rem 127
1350 DATA 0,0,28,254,28,126,129,126               :rem 180
```

Using Custom Characters

Once the new character patterns are POKEd in and character memory has been reconfigured, you are ready to use the custom characters in your programs. You can do anything with custom characters that you could do with regular characters. You can POKE them on the screen using the regular screen codes. You can PRINT them to the screen using the ASCII codes or regular PRINT statements.

For instance, if you have programmed the left bracket character to be a person running, you can put it on the screen with any of these methods:

POKE SC+N, 27
PRINT CHR$(91)
PRINT "["

Complex Custom Characters

Often the small character size just won't be enough for your needs. You'll want a much larger picture. Of course, sprites can be used for that purpose—but sprites are not particularly useful when you want to be able to locate many versions of the same figure in many locations on the screen. For instance, if you wanted a fleet of a dozen large ships in random locations on the screen, your eight sprites would be used up quickly. It is much better in such cases to program complex characters.

Complex characters are large figures made up of smaller characters. The next short program creates pictures of ships on the screen. Each ship is made up of 12 custom characters. Individually, the characters are meaningless. But we won't use them individually. Instead, we'll combine them in a string. This program redefines the characters @, A, B, C, D, E, F, G, H, I, J, and K. The complex character has @, A, and B in the top row, C, D, and E in the next row, F, G, and H in the third row, and I, J, and K in the bottom row.

POKEing a complex character to the screen is slow and ragged. PRINT works much better. However, you must be sure to include the cursor movement characters between the rows. Since the top row of the character is @, A, and B, we will begin the string that prints the complex character with @AB. Now we must get the cursor from the position on the screen directly to the right of the B to the position directly under the @. This will take one cursor down and three cursor left characters. In quote and insert modes you can enter those characters in the string just by pressing the appropriate cursor movement keys. The whole complex character string becomes the variable A$. Wherever you PRINT A$, the complex character appears.

The string array S$(*n*) contains all the possible vertical positions for the start of the ship character. The numeric array S(*n*) contains the valid horizontal values to be used with the TAB function. Since we don't want the ships overlapping, the legal positions start at every fifth line and every fourth column. This will leave a blank column or row between any two adjacent ships.

Notice how colors are assigned using the string arrary CL$(*n*). The values of CL$(*n*) are color key combinations entered in quote mode. When they are PRINTed along with the string holding the complex character, they force the ships to have different colors.

Then the program puts a random number of randomly colored ships in random locations on the screen. The last ship is always white, and it is the only white ship on the screen, but its location is random, too. This could be the flagship of the fleet or a treasure ship.

Once the first display has been created, you can press any key and the program will automatically generate another random screen. This would be a good screen display generator for a game in which the player controls a small sailboat trying to make its way through a fleet of ships to reach the treasure ship.

Program 4-4. Ships

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM S$(4),S(7),CL$(5)                          :rem 163
20 CC=14:CH=CC*1024:SC=1024                       :rem 215
30 PRINT "[2]{CLR}":POKE 53272,(PEEK(53272)AND 2
   40)OR CC                                        :rem 78
40 GOSUB 1100:GOSUB 1200                          :rem 35
50 CL$(0)="[1]":CL$(1)="{BLK}":CL$(2)="[5]":CL
   $(3)="{YEL}":CL$(4)="[3]":CL$(5)="{WHT}"
                                                  :rem 182
100 GOSUB 900                                     :rem 170
110 GET C$:IF C$=""THEN 110                       :rem 75
120 PRINT "{CLR}":GOTO 100                        :rem 251
900 FOR I=0 TO INT(RND(9)*12+5):H=INT(RND(9)*8):V=
    INT(RND(9)*5)                                 :rem 17
910 CL=INT(RND(9)*5)                              :rem 111
920 PRINT CL$(CL)S$(V)TAB(S(H))A$:NEXT            :rem 203
930 V=INT(RND(9)*5):H=INT(RND(9)*8):PRINT CL$(5)S$
    (V)TAB(S(H))A$                                :rem 154
940 RETURN                                        :rem 125
1100 FOR I=0 TO 11:T=I*8:FOR J=CH+T TO CH+T+7
                                                  :rem 155
1110 READ N:POKE J,N:NEXT:NEXT                    :rem 76
1120 FOR J=CH+256 TO CH+263:POKE J,0:NEXT J:rem 75
1130 RETURN                                       :rem 165
1200 A$="@AB{DOWN}{3 LEFT}CDE{DOWN}{3 LEFT}FGH
     {DOWN}{3 LEFT}IJK"                           :rem 163
1210 S$(0)="{HOME}":FOR I=1 TO 4:S$(I)=S$(I-1)+"
     {5 DOWN}":NEXT                               :rem 51
1220 FOR I=0 TO 7:S(I)=I*5:NEXT                   :rem 193
1230 RETURN                                       :rem 166
1300 DATA 0,0,0,0,0,3,0,0                         :rem 149
1310 DATA 6,26,3,2,214,106,66,194                 :rem 78
```

```
1320 DATA 0,128,0,0,0,48,208,17                :rem 221
1330 DATA 1,0,0,0,3,7,6,15                     :rem 220
1340 DATA 66,74,115,246,250,226,226,242        :rem 131
1350 DATA 18,85,155,23,31,31,22,30             :rem 124
1360 DATA 11,15,22,55,112,112,128,64           :rem 219
1370 DATA 250,222,226,71,122,66,66,66           :rem 33
1380 DATA 30,22,255,31,23,31,16,16             :rem 120
1390 DATA 32,16,200,126,59,15,3,1               :rem 74
1400 DATA 66,66,66,66,242,95,245,255             :rem 0
1410 DATA 16,31,27,126,86,254,120,248           :rem 27
```

Character Editors

Creating characters on graph paper works, but it is slow and requires a lot of calculation. The easiest way to create characters is using the computer. There are quite a few good character editor programs available. (One good character editor is "Ultrafont," by Charles Brannon, in *Compute!'s First Book of Commodore 64 Sound and Graphics*.)

Also, in case you'd like to use it, here are 32 custom characters, which you can use however you'd like. All the lines before 1300 load the character set and demonstrate it. To keep the demonstration going, press a key.

Program 4-5. Thirty-two Custom Characters

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
20 CC=12:VM=0:CH=CC*1024:CM=53248               :rem 88
30 GOSUB 1100:POKE 53272,(PEEK(53272)AND 240)OR CC
                                                :rem 151
40 PRINT "{CLR}"                                :rem 200
100 FOR I=0 TO 159:POKE 1024+I,I:POKE 55296+I,0:NE
    XT                                          :rem 70
120 GET A$:IF A$=""THEN 120                     :rem 73
130 POKE 53281,15:PRINT "{BLU}"                 :rem 130
140 FOR I=64 TO 95:PRINT "{HOME}{12 DOWN}"TAB(19)C
    HR$(18)CHR$(I)                              :rem 79
150 FOR J=0 TO 200:NEXT                         :rem 226
160 IF PEEK(197)=64 THEN 160                    :rem 170
170 NEXT:GOTO 140                               :rem 223
1100 POKE 56334,PEEK(56334)AND 254:POKE 1,PEEK(1)A
     ND 251                                     :rem 227
1110 RM=CM-CH:FOR I=CH TO CH+1023:POKE I,PEEK(RM+I
     ):NEXT                                     :rem 24
1120 POKE 1,PEEK(1)OR 4:POKE 56334,PEEK(56334)OR 1
                                                :rem 179
```

```
1130 FOR I=CH+1024 TO CH+1280:READ N:POKE I,N:NEXT
     :RETURN                                    :rem 54
1299 REM STARSHIP UP, LEFT, DOWN, RIGHT         :rem 49
1300 DATA 24,24,217,255,219,153,129,195         :rem 132
1310 DATA 63,25,16,252,252,16,25,63             :rem 179
1320 DATA 195,129,153,219,255,153,24,24         :rem 133
1330 DATA 252,152,8,63,63,8,152,252             :rem 185
1339 REM ROCKET UP, LEFT, DOWN, RIGHT           :rem 134
1340 DATA 16,56,56,56,124,124,84,84             :rem 196
1350 DATA 0,0,15,124,255,124,15,0               :rem 61
1360 DATA 42,42,62,62,28,28,28,8                :rem 42
1370 DATA 0,240,62,255,62,240,0,0               :rem 65
1379 REM DECIDUOUS TREE                         :rem 141
1380 DATA 57,87,186,85,20,24,24,60              :rem 149
1389 REM EVERGREEN TREE                         :rem 140
1390 DATA 8,28,8,62,28,127,60,235               :rem 97
1399 REM RUNNING FIGURE LEFT, RIGHT             :rem 114
1400 DATA 48,48,28,20,112,30,18,48              :rem 131
1410 DATA 12,12,56,40,14,120,72,12              :rem 109
1419 REM MISSILE 1&2 LEFT, 1&2 RIGHT            :rem 176
1420 DATA 0,0,0,3,14,3,0,0                      :rem 208
1430 DATA 0,0,0,48,224,48,0,0                   :rem 118
1440 DATA 0,0,0,192,112,192,0,0                 :rem 211
1450 DATA 0,0,0,12,7,12,0,0                       :rem 5
1459 REM FLICKERING FIRE (2 CHARS)              :rem 175
1460 DATA 0,0,82,16,106,38,28,0                 :rem 230
1470 DATA 0,0,42,8,86,100,56,0                  :rem 178
1479 REM BICYCLIST (COMPLEX) LEFT               :rem 243
1480 DATA 1,1,28,19,24,55,39,24                 :rem 243
1490 DATA 128,128,128,128,176,200,72,48         :rem 141
1499 REM BICYCLIST (COMPLEX) RIGHT               :rem 72
1500 DATA 1,1,1,1,13,19,18,12                   :rem 114
1510 DATA 128,128,56,200,24,236,228,24           :rem 75
1519 REM ASTEROID, ROTATE 4 POSITIONS             :rem 6
1520 DATA 0,48,120,124,30,124,24,0              :rem 108
1530 DATA 0,44,46,126,124,56,16,0                :rem 75
1540 DATA 0,24,62,120,62,30,12,0                  :rem 7
1550 DATA 0,8,28,62,126,116,52,0                 :rem 27
1559 REM BOULDER                                :rem 197
1560 DATA 0,0,28,30,30,126,127,255              :rem 121
1569 REM FLICKERING STAR (2 CHARS)              :rem 197
1570 DATA 0,8,28,54,28,8,0,0                     :rem 88
1580 DATA 0,20,54,8,54,20,0,0                   :rem 122
1589 REM PIG                                    :rem 155
1590 DATA 0,0,0,32,62,255,126,34                 :rem 22
1599 REM SAILBOAT LEFT AND RIGHT                :rem 135
1600 DATA 16,24,44,126,111,16,126,62            :rem 222
1610 DATA 8,24,52,126,246,8,126,124             :rem 184
1620 DATA 0,0,0,0,0,0,0,0                       :rem 151
```

Multicolor Mode

One limitation of the standard character screen is that the background is all one color, and the foreground (the *on* dots in each character) can only be a single color within an individual character. This is usually enough, but sometimes you need more colors in each character, or you want to be able to individually select the background color of each character. To display up to four colors within each character, you can use multicolor mode.

Bit Manipulation

Before we get to the particulars of multicolor mode, let's briefly review how to turn individual bits or groups of bits on or off without disturbing the rest of the byte. We've already done this several times, using the AND and OR operations, and if you completely understand how AND and OR work on the bits within a byte, you can skip right on to the explanation of multicolor mode.

Bits and bytes. In Chapter 1 we went over how bits work within the byte. That is, the rightmost bit is the 1s column, the next one is the 2s column, the next is the 4s column, and so on until the leftmost bit is the 128s column. To figure the decimal value of the number, you just add up the value of every column that has a 1 in it.

When we talk about the bits, we number the bits from 0 to 7, in order from right to left. So bit 0 is the 1s column, bit 1 is the 2s column, bit 2 is the 4s column, and so on until bit 7 is the 128s column. These relationships are shown in Figure 4-6.

Figure 4-6. Bit Numbers

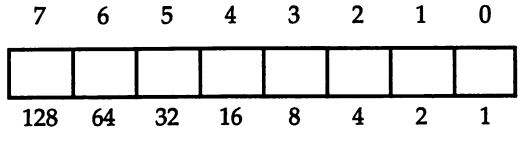


Figure 4-7. The Byte as a Row of Glasses

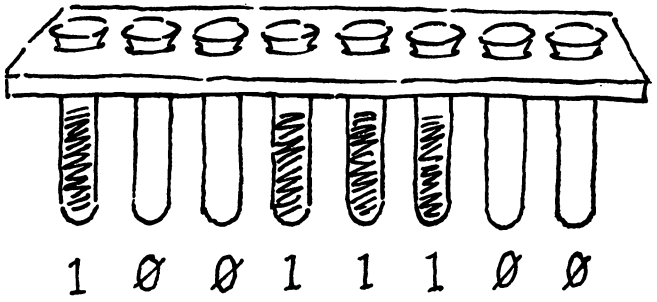


Figure 4-7 illustrates a byte as if it were a row of eight glasses linked together. Each glass is a bit. If it's full (shaded), that bit contains a 1. If it's empty, that bit contains a 0.

When you POKE a number into a byte, it has the effect of dumping out all the previous contents of the byte and then filling the appropriate bits to make the new number. Let's say a byte contains the decimal number 208, which is binary 11010000. If you POKE a 5 into that location, it is as if you emptied out the 11010000 and then filled up bits 0 and 2, so that the new contents of that byte are 00000101.

But what if you want the high nybble (left four bits) to stay 1101, and only want to change the low nybble (right four bits) to 0101?

In that case, you would PEEK the location, add 208 to 5, and POKE it back in. Now you have 213, the binary 11010101.

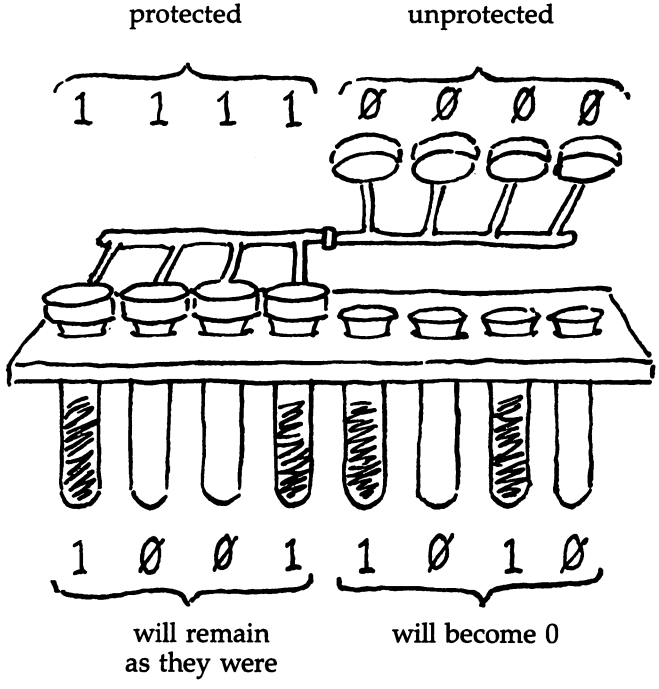
But what if the previous number were not 208, but 223, or binary 11011111. Now if you add 5 to it you get 228, or 11100100. Both the high and low nybbles are changed, and neither contains what you wanted it to contain.

What you want is a way to pour out whatever is in the low nybble, the rightmost four glasses, and then fill up just the ones that you want filled, while the high nybble is completely left alone—whatever is in there remains untouched.

To do this, you have two commands—AND and OR.

AND. In Figure 4-8, we visualize AND as a series of lids. Using AND is like covering the *on* (1) bits with a lid and then tipping the glasses. All the uncovered bits are emptied out—they become zeros. All the covered bits are left exactly the way they were—if they were full, they stay full, and if they were empty, they stay empty.

Figure 4-8. AND as Lids

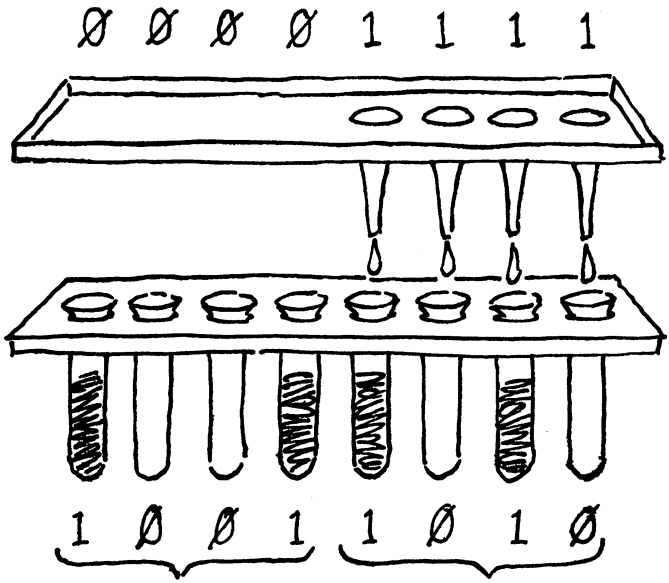


Notice that it doesn't matter which number is represented by the glasses and which is represented by the lids—the result is the same either way.

So you use AND to protect the condition of certain bits in a byte, and empty or zero out all the others. If you AND any number with 11000000, or 192, all the bits in the other number except bits 6 and 7 will become zeros, regardless of what they contained before. The two high bits, however, will stay as they were—if they were ones, they stay ones, and if they were zeros, they stay zeros.

OR. If AND is represented by lids, OR functions like funnels, designed to fill certain bits and leave the others alone. Figure 4-9 shows how OR works. Whichever bits are *on* are filled—all other bits are left alone.

Figure 4-9. OR as Funnels



AND and OR together. Many memory locations are used for many purposes at once. We've already seen how location 53272 points at once to the start of screen memory (high nybble) and the start of character memory (low nybble).

First, you need to pick up the value that is in location 53272 by PEEKing that location. Then, if you want to change just character memory, you must empty out the low nybble—get rid of whatever information is there. But you don't want to clear out the high nybble. So you AND the byte at 53272 with the byte 11110000 (decimal 240). This has the effect of emptying bits 0–3, while keeping a lid on whatever is in bits 4–7.

Then you put the correct value in the low nybble. You do this by ORing the byte with the code for the 2K boundary where character memory begins.

When these operations are finished, you're ready to POKE the result into location 53272. The next time the VIC-II checks location 53272, it will find the new values there. It will

continue to look for screen memory where it was before, because you didn't change that location, and it will start to look for character memory where your new instruction tells it to look.

All one way. Of course, you don't always need to use *both* OR and AND. For instance, if you are going to put a zero into a certain bit or group of bits, you only need the AND operation. Once the bits you want emptied are empty, they contain zeros—you don't have to OR it with a zero! Anytime you AND a number with 15, the high nybble (bits 4–7) will be all zeros. And when you AND a number with 240, the low nybble will be all zeros.

Likewise, if you are filling a bit or a group of bits, with none to be left as zeros, you don't need AND. ORing it will be enough. Anytime you OR a number with 240, the high nybble will be all ones, regardless of what was there before. And when you OR a number with 15, the low nybble will be all ones.

These formulas are always true:

nnnnnnnn AND 11110000 = *nnnn*0000
nnnnnnnn OR 11110000 = 1111*nnnn*

On and off. To turn on a particular bit (set it to 1), just OR the byte with the value of that bit. Bit 3 is the 8s column—so to set bit 3, PEEK the value already there, OR it with 8, and POKE it back into the same location. That bit will be set.

To turn off a particular bit (clear it to 0), just AND the byte with a number that consists of all ones except for a zero in that bit. For example, to turn off bit 5, you only need to AND the number with 223, or binary 11011111.

If you don't want to bother with calculating the decimal equivalents of binary numbers, there is an easy formula. To clear a bit, subtract the bit's value from 255. Bit 4 is the 16s column. To clear that bit in location 53270: POKE 53270, PEEK (53270) AND 255 – 16.

Reversed characters. This is a general rule with binary numbers. To find the complement of a binary number, just subtract it from 255. The opposite of 11000011 (decimal 195) is 00111100. The decimal value can be found with the operation 255 – 195.

You can reverse a character using this method. When you are copying a character from one location to another, just PEEK

the value from the original location, subtract it from 255, and POKE the result in the new location.

Using Multicolor Character Mode

Multicolor character mode is controlled at bit 4 of location 53270. If bit 4 is 1, then multicolor mode is enabled. If bit 4 is 0, then standard character mode is in force. To turn on multicolor mode:

POKE 53270, PEEK(53270) OR 16

To turn off multicolor mode:

POKE 53270, PEEK(53270) AND 239

(Notice that the number 239 is the result of 255 – 16.)

But what *is* multicolor mode?

Color instructions. Each bit in the character pattern can be regarded as a color instruction. Every *off* bit is an instruction to the VIC-II chip to display the *background* color at that dot on the screen. The background color is the color code (from 0 to 15) stored at location 53281. Every *on* bit is an instruction to the VIC-II to display the *foreground* color at that dot on the screen. The foreground color is the color code (from 0 to 15) found at the corresponding position in color memory (from 55296 to 56295).

Let's say that we've PRINTed an *X* at byte 9 of screen memory. At byte 9 of color memory, the color code is 7, for dark blue. This is the foreground color for that character position on the screen, regardless of what character is PRINTed there. At location 53281, the color code is 15, for light gray. This is the background color for the whole screen. Now the VIC-II reads the pattern for *X* (screen code 24) starting at byte 8*24. For every 0 bit, a light gray dot appears on the screen at the corresponding position within the character. For every 1 bit, a dark blue dot appears.

So in standard character color mode, each color instruction is one bit, and it can select one of two possible colors, the foreground color or the background color.

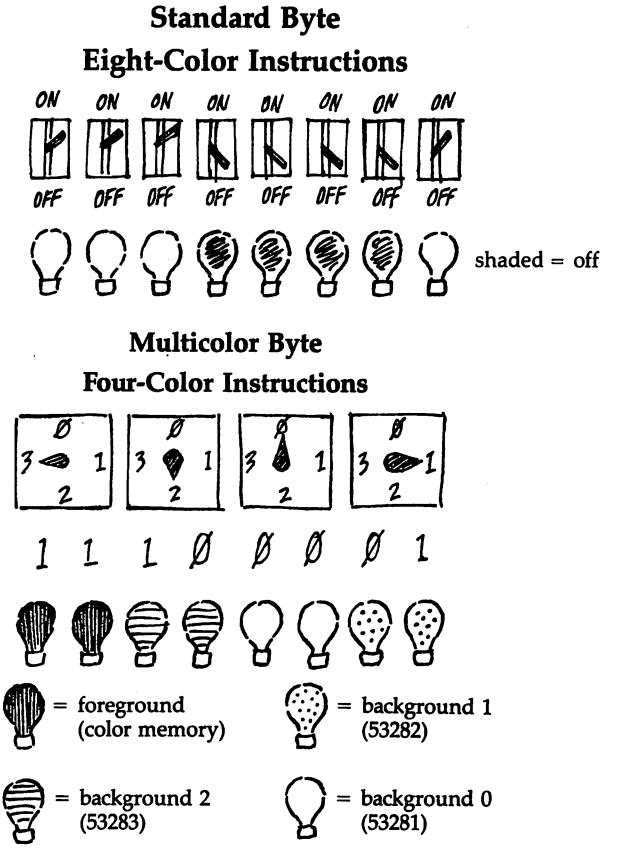
In multicolor character mode, each color instruction is *two* bits, a bit-pair. This means that it can specify one of four different colors. The bit-pair can be either 00, 01, 10, or 11. Now when the VIC-II reads a character pattern, it displays the color at 53281 when it finds the bit-pair 00, and it displays the

color at that position in color memory when it finds the bit-pair 11.

The other two bit-pairs are color instructions, too. Every 01 bit-pair displays the color code (from 0 to 15) stored at location 53282, and every 10 bit-pair displays the color code (from 0 to 15) stored at location 53283.

Bit-pairs and double dots. There is one problem: Each character pattern still has only eight bytes, and each byte of the pattern still has only eight bits. At the same time, the character position on the screen is still eight dots wide and eight dots high. But with two bits being used up by each color instruction, there can be only half as many different color instructions per byte, and therefore only half as many per character. (See Figure 4-10.)

Figure 4-10. Bit-pairs



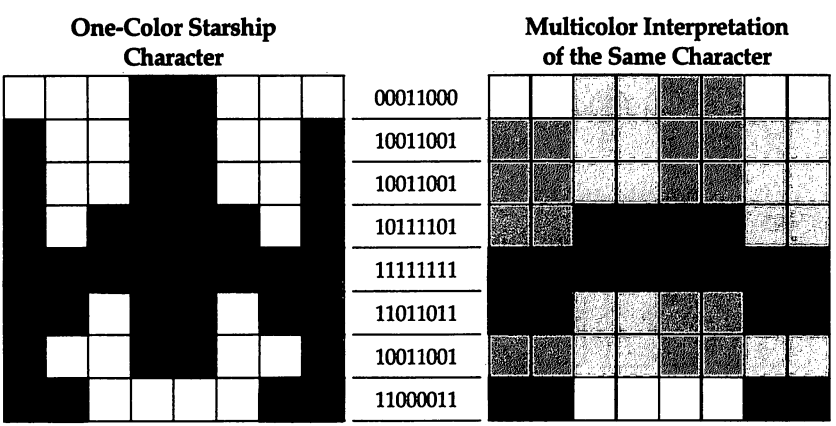
The solution is that each color instruction controls two dots. The two dots controlled by the same bit-pair always display the color called for by that instruction. The double dot acts like one dot when it comes to color selection.

Naturally, this does funny things to character patterns. You can see this by filling the screen with characters—an easy way to do this is to LIST a program you already have in memory. Then switch to multicolor mode with this instruction:

```
POKE 53270, PEEK(53270) OR 16
```

Watch the screen change. Suddenly the characters aren't too legible anymore! Figure 4-11 show why, using the pattern for a starship. When each bit controlled an individual dot, there were spaces of background color between individual lines of the foreground color. But now, in multicolor mode, the zeros no longer represent single dots of background color—now they are paired with the adjacent 1 to form a bit-pair with either a 01 or a 10 color instruction, and the double dot will display that single color.

Figure 4-11. Multicolor Patterns



Color memory switches. There's another feature with multicolor mode. Once it has been enabled at 53270, you can switch it on or off individually at each character position on the screen. In multicolor mode, color memory is used differently. In standard color mode, the low nybble contains one of 16 possible color codes, from 0000 to 1111 (decimal 0 to 15). The high nybble cannot be used.

In multicolor mode, however, only bits 0 to 2 are used for the color code. This means that you can select only eight colors, from 000 to 111 (decimal 0 to 7). Bit 3 is used differently now. It switches between multicolor and standard mode *for that character position*. If bit 3 is set (1), at byte 320 of color memory, the VIC-II will interpret the bytes of the character pattern as bit-pairs, and display them as double dots. If bit 3 is clear (0), the VIC-II will interpret each bit as an individual color instruction.

This means that by controlling the number in color memory, you can control whether or not multicolor mode is used. On the same screen you can have letters of regular text and custom characters in multicolor mode.

The easiest way to set color memory is to PRINT with the normal color keys embedded in a string. Now, however, the colors you get by pressing Commodore and a number key select multicolor mode for everything that is PRINTed to the screen until the next color key. However, it will now give you exactly the same foreground color as you would get by pressing CTRL and the same number key. The colors from 8 to 15 are not accessible for the foreground in multicolor text mode, though they can still be used for the colors controlled at 53281, 53282, and 53283.

This program creates a display using multicolor mode. It is essentially the same as the ship display we used before, except that the ships have been replaced by multicolored trees, and a single black-leafed tree has replaced the single white ship at the end of each new screen. Again, press any key to create a new screen.

Program 4-6. Multicolored Forest

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM S$(4),S(9),CL$(5)                          :rem 165
20 CC=14:CH=CC*1024:SC=1024                       :rem 215
30 PRINT "{GRN}{CLR}":POKE 53272,(PEEK(53272)AND 2
   40)OR CC                                       :rem 215
40 GOSUB 1100:GOSUB 1200                          :rem 35
50 CL$(0)="{6}":CL$(1)="{6}":CL$(2)="{3}":CL
   $(3)="{6}":CL$(4)="{8}":CL$(5)="{BLK}"
                                                  :rem 96
60 POKE 53282,9:POKE 53283,0:POKE 53281,11:POKE 53
   280,0                                          :rem 137
```

```
100 GOSUB 900                                    :rem 170
110 GET C$:IF C$="" THEN 110                      :rem 75
120 PRINT "{CLR}":GOTO 100                       :rem 251
900 FOR I=0 TO INT(RND(9)*12+5):H=INT(RND(9)*8):V=
    INT(RND(9)*5)                                 :rem 17
910 CL=INT(RND(9)*5)                             :rem 111
920 PRINT CL$(CL)S$(V)TAB(S(H))A$:NEXT           :rem 203
930 V=INT(RND(9)*5):H=INT(RND(9)*8):PRINT CL$(5)S$
    (V)TAB(S(H))A$                               :rem 154
940 RETURN                                       :rem 125
1100 FOR I=0 TO 11:T=I*8:FOR J=CH+T TO CH+T+7
                                                 :rem 155
1110 READ N:POKE J,N:NEXT:NEXT                    :rem 76
1120 FOR J=CH+256 TO CH+263:POKE J,0:NEXT J:rem 75
1130 RETURN                                      :rem 165
1200 A$="@AB{DOWN}{3 LEFT}CDE{DOWN}{3 LEFT}FGH
    {DOWN}{3 LEFT}IJK"                           :rem 163
1210 S$(0)="{HOME}":FOR I=1 TO 4:S$(I)=S$(I-1)+"
    {5 DOWN}":NEXT                                :rem 51
1220 FOR I=0 TO 9:S(I)=I*4:NEXT                  :rem 194
1230 RETURN                                      :rem 166
1300 DATA 12,60,60,240,255,61,61,1               :rem 114
1310 DATA 60,252,252,240,243,255,51,67            :rem 73
1320 DATA 0,0,0,0,0,192,240,64                   :rem 160
1330 DATA 0,3,3,0,0,3,3,3                        :rem 164
1340 DATA 125,127,252,236,236,233,253,189        :rem 235
1350 DATA 64,240,48,0,204,255,255,12             :rem 228
1360 DATA 3,0,3,3,63,53,1,240                    :rem 121
1370 DATA 189,125,248,200,184,52,84,16            :rem 90
1380 DATA 95,3,3,0,0,60,204,255                  :rem 230
1390 DATA 60,60,233,224,192,0,1,40               :rem 120
1400 DATA 20,23,89,150,42,169,170,136             :rem 25
1410 DATA 252,252,111,140,0,0,84,138             :rem 214
```

You can also POKE into color memory to turn multicolor mode on or off for particular character positions—or for the whole screen. To change a location LC in color memory to multicolor code without changing the color, use this statement:
POKE LC, (PEEK(LC)AND 7) OR 8

To turn off multicolor memory at the same location, again without changing the color, use this statement:
POKE LC, PEEK(LC)AND 7
(You'll notice that in both statements, we are in effect clearing the high nybble. It really doesn't matter much, because color memory is a unique area of RAM—each byte only contains four bits. In some 64s, the high nybble contains random garbage. In others, the high nybble contains zeros. It's usually

best just to erase those bits by ANDing with 15 or less, so there's no chance they can mess up your calculations.)

Since multicolor characters allow very little detail within an individual character, you will almost always use them as building blocks for larger images. You can design them to be either part of a larger complex character or pieces that can be combined many different ways.

5 Getting Things Moving

Getting Things Moving

Movement on the television screen is a series of still pictures. The pictures seem to move because each still picture is slightly changed from the one before, and each new picture comes on the screen so quickly that our 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 three 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 each movement 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 screen location to another. The other is making the figure seem natural in its movements. Even though the terms *movement* and *animation* are used almost interchangeably, in this book I'd

like to use them to mean different things.

Movement is changing the location of the figure.

Animation is changing the shape of the figure.

For instance, in *Donkey Kong*, *movement* is getting Mario from one place to another on the screen—up and down ladders, along ramps, jumping, etc. *Animation* is the way Mario's shape changes: his legs move, he seems to face the direction he's moving in, he moves the hammer up and down. All these things make his movement seem human and communicate what he's doing.

Chances are, since sprites are easy to use on the 64, that you'll want to use a sprite for the main player-controlled figure. However, you'll usually want a lot of other things to be moving, too, and in this chapter we'll see how to get fast and varied movement using character graphics on the screen.

Three Steps to Movement

Let's assume that the figure that 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.

Wraparound Movement with POKEs

Here's a program that will show you movement at it simplest. It uses POKE commands to make a single figure move in random patterns on the screen. When it reaches the edge of the screen, it wraps around—it disappears on the left side and reappears on the right side, or disappears at the top of the screen and reappears at the bottom.

It's important to understand, step by step, what's happening in this program. Most of your programming time—and debugging time, too—will be spent with getting exactly the movement effects you want on the screen. And most of the principles here will apply to sprites, too. So even though this program is quite short, let's go through it line by line to make absolutely clear what's happening. This program will be the basis of almost every other program in this chapter. If you type this in and SAVE it, most of the other programs can be entered just by LOADing this program; adding, editing, or replacing

lines; and then SAVEing the new version with another filename or on another tape.

How the Program Works:

Line	Function
20	Set up variables SM = screen memory start address CM = color memory start address EV = end of screen—vertical. (The last row number *minus 1*. Later, the program will know we are at the last row of the screen when the row number is *greater than* this number.) EH = end of screen—horizontal. (The last column number *minus 1*.) BV = beginning of screen—vertical. (The first row number *plus 1*. As always, the numbering really begins with 0.) BH = beginning of screen—horizontal. (The first column number *plus 1*.) HN = number of columns to return horizontally for wraparound. (This is the total number of columns available on the screen.) VN = number of rows to return vertically for wraparound. (This is the total number of rows available on the screen.)
30	FG = screen code for the figure BL = blank—the screen code for erasing the figure. (Usually 32, but if you have the screen filled with another character as a pattern, you would use that character's screen code for the erasure.) FC = figure color—the number to POKE into color memory for the figure BC = the color of the background XC = the foreground color of the BL character. (With the space character [screen code 32] this doesn't matter, but if the background character shows up, it does.)
40	POKE the background color and clear the screen.
50	Set initial character movement and position values. Unlike the values above, these variables will be changed often by the program. They are set now only to establish the starting position and direction of the characters.

H = horizontal direction of movement (1 = right, −1 = left)
V = vertical direction of movement (1 = down, −1 = up)
XH and XV = old horizontal and old vertical column and row numbers. (At the beginning of a particular movement, these hold the *current* position of the figure. This allows us to remember where the character *was* so we can erase it there.)
XP = old offset. This number is always XH + 40*XV. It represents the number of bytes to count in from the start of screen memory and/or from the start of color memory to find the byte where the figure is.
Then POKE the figure color into color memory plus XP, and the figure character into screen memory plus XP. The figure will now appear on the screen.

100-190 Main Loop

100 Check the timer (TI$). If enough time has passed, go get a random direction assignment at line 990.

190 If both directions are zero, no movement will take place—go back to 990 and get new movement values until at least one of the direction variables is not zero.

200-290 Movement Routine

200 Assign the new horizontal position. (See the explanation of this formula after the program listing.)

210 Assign the new vertical position. (See the explanation of this formula after the program listing.)

220 Set the new offset (NP) of the figure byte from the start of screen and color memory.

230 POKE the new color. Then erase the old character. Then POKE the new character. Then erase the old color. (The order is important. POKE commands take long enough that color changes will be visible, if only as a momentary flash. Therefore, no other statement should come between POKEing the figure into the new location and erasing it [POKEing BL] at the old character location.)

240 Set the old-value variables, XP, XH, and XV, to the new values. (When the next movement takes place, these variables will be used to erase the figure where we just put it.)

290 Close the movement loop.

990 Random Direction Subroutine
(This one-line routine randomly generates one of three numbers: −1, 0, and 1. The random number H then controls the direction of horizontal movement and the random number V controls the direction of vertical movement.)

Program 5-1. POKE Movement Demonstration

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
20 SM=1024:CM=55296:EV=23:EH=38:BV=1:BH=1:HN=40:VN
   =25                                          :rem 99
30 FG=87:BL=32:FC=6:BC=15:XC=0                  :rem 224
40 POKE 53281,BC:PRINT "{CLR}"                  :rem 229
50 H=1:V=0:XH=19:XV=11:XP=XH+40*XV:POKE CM+XP,FC:P
   OKE SM+XP,FG                                 :rem 92
100 IF VAL(TI$)>1 THEN GOSUB 990:TI$="000000"
                                                :rem 113
190 IF H=0 AND V=0 THEN GOSUB 990:GOTO 190:rem 210
200 NH=XH+H+HN*((XH>EH AND H=1)-(XH<BH AND H=-1))
                                                :rem 147
210 NV=XV+V+VN*((XV>EV AND V=1)-(XV<BV AND V=-1))
                                                :rem 32
220 NP=NH+NV*40                                 :rem 98
230 POKE CM+NP,FC:POKE SM+XP,BL:POKE SM+NP,FG:POKE
    CM+XP,XC                                    :rem 134
240 XP=NP:XH=NH:XV=NV                           :rem 143
290 GOTO 100                                    :rem 101
990 H=SGN(INT(RND(9)*3)-1):V=SGN(INT(RND(9)*3)-1):
    RETURN                                      :rem 74
```

The logic in lines 200 and 210. A row of variables and calculations like this can be puzzling, and since this formula eliminates a lot of IF-THENs and GOSUBs or GOTOs, it's worth understanding it and using it.

The first part of line 200 is easy. We are setting the value of NH, the new horizontal position. It is equal to XH, the old position, plus H, the direction of movement.

The rest of the line is devoted to checking to see if the new position is beyond the edge of the screen and, if it is, moving it to the opposite side of the screen.

The math here depends on a quirk of the computer. When BASIC evaluates a relational expression (like X>Y or P=C or XX<25), the answer comes back as either true or false. If it is *true*, the answer is the number -1. If it is false, the answer is the

number 0. This is how BASIC checks for conditional statements like IF X=Y THEN 350. But by using the true= −1 and false=0 values, you can avoid the problems of using IF.

Look at the first of the tests: (XH>EH AND H=1). If the old horizontal position (XH) is greater than the edge-of-screen value (EH) *and* the direction of movement is to the right (H=1), this expression will evaluate to −1, or true. If either of the two conditions is false (either we are not beyond the right edge marker or the direction of movement is not to the right), the whole expression is false, and its value is 0.

The same process takes place in the other expression, only this time we are testing the left edge of the screen. The expression will be true (−1) if the old horizontal position is less than 1 *and* if the direction of movement is leftward (H= −1).

Now, these two expressions are within the larger expression, with a minus sign between them. We are subtracting the second expression from the first expression. It is important to remember that they can't *both* be true. However, they *can* both be false, and in fact usually will be—most of the time, the figure will not be crossing an edge. If neither expression is true, the entire expression will have a value of zero. Since 40*0 is 0, none of this has any effect on the rest of the line. NH will be equal to XH plus H plus zero. This is the usual result.

However, if the figure is at the right-hand edge, moving rightward, the *left* expression (XH>EH AND H=1) will be true (−1). The *right* expression will therefore be false (0). The large expression will then have a value of −1−0, or −1. We will multiply that by HN, which has a value of 40. Now the whole line is changed: NH now equals XH plus H *plus* 40 times −1 (which is, of course, the same as XH plus H *minus* 40). We will add 1 to the value of XH, but then we'll subtract 40 from it. The effect is to take the figure from column 39 to column 0 on the screen. It has wrapped around from the right-hand edge to the left-hand edge.

If the figure is at the left-hand edge, moving leftward, then the *right* expression is true (−1) and the left one is false (0). Now the large equation is 0−(−1). Subtracting a negative number is the same as adding, so that the whole expression evaluates to 0+1, or 1. Now we are multiplying HN (40) by 1, so the result is 40. Since in this case XH was 0 and H was −1, we are setting the value of NH to −1+40, or 39—the number of the rightmost column on the screen. Again, the figure has

wrapped around.

Line 210 uses exactly the same pattern. Only the actual values involved are different, since instead of columns 0 to 39, we are dealing with rows 0 to 24.

Changing the ranges. You don't have to use the full screen. You can put the figure you're moving inside a smaller area of the screen just by giving new values to the boundaries. Program 5-2 does this. It is almost identical to Program 5-1. So we'll only need to list and explain the few lines that are changed.

In the program setup, you'll notice that we eliminate wraparound by the simple method of changing HN and VN to 1. Now, if the edge test comes out true, instead of moving 40 columns or 25 rows in the opposite direction, we'll only move back *one* column or row. In effect, then, when the figure tries to move beyond an edge, it simply stays in the same row or column.

We change the ranges just as easily, by giving new values to BV, BH, EV, and EH. Now when the program tests for the edge, it thinks the edge is in a different place, and acts accordingly. When you RUN the program, you'll see that the character still moves around, but it often stops or slides along the edge of an invisible wall.

Program 5-2. A Shrinking Screen

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
20 SM=1024:CM=55296:EV=18:EH=28:BV=6:BH=11:HN=1:VN
   =1                                          :rem 51
100 IF VAL(TI$)>2 THEN GOSUB 990:TI$="000000"
                                              :rem 114
220 NP=NH+NV*40:IF NP=XP THEN GOSUB 990:GOTO 190
                                              :rem 12
```

Moving with PRINT

The next program again starts with Program 5-1 and makes changes. There are more changes this time, however, since we'll be moving using PRINT statements instead of POKE statements. Therefore, the whole listing is included, though we'll only talk about the changes.

Advantages of PRINT. The advantage of PRINT is speed. PRINT takes care of color assignments and cursor movements all at once. We've already seen how complex characters (trees

and ships) can be displayed instantly with PRINT statements. They can also be moved with PRINT statements, using the same principles.

We will still keep track of new and old horizontal and vertical positions. However, instead of combining them into an offset which is added to the start of screen and color memory (NP), we'll use the NH and XH numbers with the TAB function—the screen editor will TAB us to whatever column we specify. The NV and XV numbers will be indexes into an array V$(*n*), in which V$(0) is the HOME character (which takes us to the top row), V$(1) is HOME CURSOR-DOWN (which takes us to the second row), and each succeeding V$(*n*) string contains one more CURSOR DOWN. Then if we tell the computer to PRINT V$(9)TAB(11)FG$, it will PRINT the string FG$ at row 9, column 11.

Disadvantages of PRINT. The main disadvantage of PRINT is scrolling. The *can* be an advantage, too, of course, as we'll see with a later program, but when you don't want the screen to scroll, you just can't PRINT in the last line. PRINTing in the last column can also cause odd things to happen on the screen. So as a general rule, when you're moving with PRINT you have to pretend that the rightmost column and bottom row don't exist. In effect, you shrink the screen just as we did with Program 5-2, only not as much.

There are a couple of ways to hide the fact that the right column and bottom row aren't used. You can make the screen and border the same color—then the edge will be wherever you put it and no one will be the wiser. Or you can put a frame all the way around the screen—if you don't use column 0 and row 0 either, it won't look like a mistake that you aren't using column 39 and row 24.

Notice the changes in value for EV, EH, VN, and HN to eliminate the two unPRINTable columns. Line 30 sets up the array V$(*n*) to control vertical movement.

Program 5-3. Moving with PRINT

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM V$(23)                                    :rem 107
20 EV=22:EH=37:BV=1:BH=1:HN=39:VN=24:FG$="{BLK}X":
   BL$="{BLK} ":BC=7                             :rem 124
```

```
30 V$(0)="{HOME}":FOR I=1 TO 23:V$(I)=V$(I-1)+"
   {DOWN}":NEXT                                    :rem 200
40 POKE 53281,BC:POKE 53280,BC:PRINT "{CLR}":rem 1
50 H=1:V=0:XH=19:XV=11:PRINT V$(XV)TAB(XH)FG$
                                                   :rem 217
100 IF VAL(TI$)>1 THEN GOSUB 990:TI$="000000"
                                                   :rem 113
190 IF H=0 AND V=0 THEN GOSUB 990:GOTO 190:rem 210
200 NH=XH+H+HN*((XH>EH AND H=1)-(XH<BH AND H=-1))
                                                   :rem 147
210 NV=XV+V+VN*((XV>EV AND V=1)-(XV<BV AND V=-1))
                                                    :rem 32
220 PRINT V$(XV)TAB(XH)BL$V$(NV)TAB(NH)FG$:rem 242
240 XH=NH:XV=NV                                    :rem 210
290 GOTO 100                                       :rem 101
990 H=SGN(INT(RND(9)*3)-1):V=SGN(INT(RND(9)*3)-1):
    RETURN                                          :rem 74
```

Little Boxes

Program 5-4 introduces a new feature—multiple figure movement. Using the PRINT method, we'll put four figures on the screen and move them all. But they'll each remain in a separate area of the screen, as if each had its own little box.

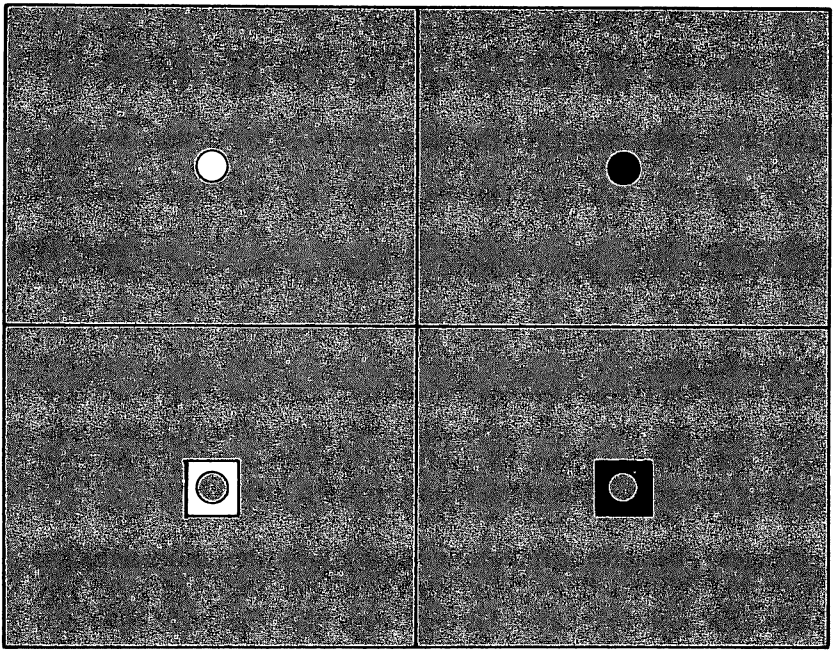
This program started with Program 5-3 and made changes—but enough changes that again the whole listing is included. Notice that all the movement control variables have been changed to arrays. The movement of figure FG$(0) is controlled by the variables NH(0), XH(0), H(0). The edges are also arrays, since each figure has its own miniscreen to work with: FG$(0) can move only within the area defined by EH(0), BH(0), EV(0), and BV(0); when it reaches one of those edges, it wraps around using the variables VN(0) and HN(0).

The arrays are all set up in the routine starting at line 300. Figure 5-1 illustrates the screen arrangement. FG$(0), the white circle, stays in the upper-left corner; FG$(1), the black circle, stays in the upper-right; FG$(2), the reversed white circle, stays in the lower-left, and FG$(3), the reversed black circle, stays in the lower-right.

Notice that pairs of figures have borders in common. The two white figures, for instance, have the same horizontal edges, and the two black circle figures have the same horizontal borders. The two regular circles (0 and 1) have the same vertical edges, and the two reversed circles (2 and 3) have the same vertical borders.

Therefore, the setup routine from 300 to 390 uses loops and sets the boundaries using these pairs.

Figure 5-1. Segmented Screen



Program 5-4. Little Boxes

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM V$(23),EV(3),EH(3),BV(3),BH(3),HN(3),VN(3),
   FG$(3),H(3),V(3)                          :rem 110
20 GOSUB 300:BL$="{BLK} ":BC=5               :rem 163
30 V$(0)="{HOME}":FOR I=1 TO 23:V$(I)=V$(I-1)+"
   {DOWN}":NEXT                              :rem 200
40 POKE 53281,BC:POKE 53280,BC:PRINT "{CLR}":rem 1
50 GOSUB 990:FOR I=0 TO 3:PRINT V$(XV(I))TAB(XH(I)
   )FG$(I):NEXT                              :rem 52
100 FOR I=0 TO 3:IF VAL(TI$)>1 THEN GOSUB 990:TI$=
    "000000"                                 :rem 30
190 IF H(I)=0 AND V(I)=0 THEN GOSUB 980:GOTO 190
                                             :rem 5
```

```
200 NH(I)=XH(I)+H(I)+HN(I)*((XH(I)>EH(I)AND H(I)=1
    )-(XH(I)<BH(I)AND H(I)=-1))              :rem 151
210 NV(I)=XV(I)+V(I)+VN(I)*((XV(I)>EV(I)AND V(I)=1
    )-(XV(I)<BV(I)AND V(I)=-1))               :rem 36
220 PRINT V$(XV(I))TAB(XH(I))BL$V$(NV(I))TAB(NH(I)
    )FG$(I)                                  :rem 244
240 XH(I)=NH(I):XV(I)=NV(I)                   :rem 58
290 NEXT:GOTO 100                            :rem 222
300 FORI=0 TO1:EV(I)=11:EV(I+2)=22:BV(I)=1:BV(I+2)
    =13:VN(I)=12:VN(I+2)=11:NEXT             :rem 200
310 FORI=0 TO 2 STEP 2:EH(I)=18:EH(I+1)=37:BH(I)=1
    :BH(I+1)=21                               :rem 5
320 HN(I)=1:HN(I+1)=1:NEXT                   :rem 224
330 FG$(0)="{WHT}Q":FG$(1)="{BLK}Q":FG$(2)="{WHT}
    {RVS}Q{OFF}":FG$(3)="{BLK}{RVS}Q{OFF}":rem 204
340 FOR I=0 TO 1:XV(I)=7:XV(I+2)=19:NEXT  :rem 253
350 FOR I=0 TO 2 STEP 2:XH(I)=10:XH(I+1)=30:NEXT
                                             :rem 115
390 RETURN                                   :rem 124
980 H(I)=SGN(INT(RND(9)*5)-3):V(I)=SGN(INT(RND(9)*
    5)-3):RETURN                             :rem 133
990 FORJ=0 TO 3:H(J)=SGN(INT(RND(9)*5)-3):V(J)=SGN
    (INT(RND(9)*5)-3):NEXT:RETURN            :rem 175
```

Multiple Movements

Single character animation can make a good game, without ever using a sprite—several of the games in this book work that way. However, usually in game programming on the 64, the player's figure will be a sprite. Individual character movement will then be used for computer-controlled opponents.

If there is a single opponent figure, it might as well be a sprite, too. But many arcade games use a different principle—instead of a single smart opponent, they use many dumb opponents. The player is overwhelmed by numbers, not by speed or accuracy. Think of *Asteroids*, *Space Invaders*, *Donkey Kong*, and *Donkey Kong Jr.* The asteroids and aliens, barrels and eaters are not particularly fast or smart—their movements are predictable and easy to beat. What makes the games challenging is that there are so *many* of them, coming at you from every direction.

Program 5-5 puts ten figures on the screen and moves them in fairly predictable patterns. However, they aren't all the *same* patterns. Some move up and down, some move left and right. The same movement and edge-checking program we've used all along is again the basis of this program.

There are five pairs of characters. Both members of each pair start out in the same place on the screen, but one will move only horizontally, and the other will move only vertically. Since each pair starts in a different place, and since the horizontally moving figures have farther to travel than the vertically moving figures, the regular movement wouldn't necessarily be obvious during the middle of a game. Each individual character moves slowly—but if you were controlling a large sprite that had to get through them without touching any of them to pick up certain objects scattered about the screen, you'd think they were moving pretty fast!

How the Program Works:

Line	Function
10-40	Standard Setup
50	Set initial screen positions.
100-110	Vertical Movement Loop Figures 0-4 move only vertically. Therefore, there is never a need to check their horizontal position or movement. This loop handles only vertical movement, and if an edge is reached the program jumps to the subroutine at 400, which changes the direction variable DI(*n*) so the figures bounce off the walls.
120-130	Horizontal Movement Loop Figures 5-9 move only horizontally, so only horizontal movements are checked. The direction variables are changed at 450.
300-390	Array Setup
400-450	Reverse Direction Subroutines

Program 5-5. Move Ten Figures

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM V$(23),FG$(9),XV(9),NV(9),XH(9),NH(9),DI(9)
                                                :rem 117
20 GOSUB 300                                    :rem 117
30 BL$=" ":BC=1                                 :rem 195
40 POKE 53281,BC:POKE 53280,BC:PRINT "{CLR}":rem 1
50 FOR I=0 TO 9:PRINT V$(NV(I))TAB(NH(I))FG$(I):NE
   XT                                           :rem 202
100 FOR I=0 TO 4:NV(I)=NV(I)+DI(I):IF NV(I)>23 OR
    {SPACE}NV(I)<0 THEN GOSUB 400               :rem 130
```

```
110 PRINT V$(XV(I))TAB(NH(I))BL$V$(NV(I))TAB(NH(I)
    )FG$(I):XV(I)=NV(I):NEXT                    :rem 94
120 FOR I=5 TO 9:NH(I)=NH(I)+DI(I):IF NH(I)>38 OR
    {SPACE}NH(I)<0 THEN GOSUB 450               :rem 97
130 PRINT V$(NV(I))TAB(XH(I))BL$V$(NV(I))TAB(NH(I)
    )FG$(I):XH(I)=NH(I):NEXT                    :rem 68
140 GOTO 100                                    :rem 95
300 FOR I=0 TO 4:NH(I)=3+I*8:NH(I+5)=NH(I):NV(I)=3
    +I*4:NV(I+5)=NV(I):NEXT                     :rem 116
310 FOR I=0 TO 9:XV(I)=NV(I):XH(I)=NH(I):NEXT
                                                :rem 100
320 V$(0)="{HOME}":FOR I=1 TO 23:V$(I)=V$(I-1)+"
    {DOWN}":NEXT                                :rem 250
340 FG$(0)="{RED}Q":FG$(1)="{PUR}Z":FG$(2)="{BLK}A
    ":FG$(3)="{GRN}{B}":FG$(4)="{BLU}X"         :rem 144
350 FOR I=0 TO 4:FG$(I+5)=FG$(I):NEXT           :rem 242
360 FOR I=0 TO 9 STEP 2:DI(I)=1:DI(I+1)=-DI(I):NEX
    T                                           :rem 22
390 RETURN                                      :rem 124
400 DI(I)=-DI(I):NV(I)=XV(I)+DI(I):RETURN       :rem 181
450 DI(I)=-DI(I):NH(I)=XH(I)+DI(I):RETURN       :rem 158
```

Speed through Complication

If this seems slow to you, remember that speed is largely an illusion in arcade games. We can make this routine seem much, much faster by *slowing down* the ten figures already on the screen, but adding two other characters that travel diagonally at ten times the speed of the other characters.

Program 5-6 does this. Four lines are added *within* each of the two movement loops — lines 105-108 and 125-128. These lines are loops themselves, which are carried out completely each time through the larger movement loops. This means that figures 10 and 11 both move *every* time any *one* of the other figures moves. Notice, too, that figures 10 and 11 always move both horizontally and vertically—the effect is that they move diagonally. They require their own rebound routines at 420 and 470, and they are set up at lines 370 and 380. There are also changes from Program 5-5 in lines 10 and 50.

Simple changes, really—but the effect is quite pronounced. Imagine the same program with redefined characters, so that these are ants moving across a sidewalk, and the two fast ones are spiders. Or these are cars, and the two fast ones are speeders. Or balls, or fish, or whatever you want them to be.

The technique of embedding one figure's movement inside

a loop that controls other figures is one we'll use again. We'll use it to control missiles in Chapter 13, for example, since you'll naturally want missiles to move faster than their targets.

(Much of this program is identical to Program 5-5, Move Ten Figures. If you've already entered and SAVEd Program 5-5, all you have to do is enter the *new* lines. These are: 10, 50, 105-108, 125-128, 370, 380, 420, and 470.)

Program 5-6. Varied Speeds

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM V$(23),FG$(11),XV(11),NV(11),XH(11),NH(11),
   DI(9),DV(11),DH(11)                          :rem 38
20 GOSUB 300                                    :rem 117
30 BL$=" ":BC=1                                 :rem 195
40 POKE 53281,BC:POKE 53280,BC:PRINT "{CLR}":rem 1
50 FOR I=0 TO 11:PRINT V$(NV(I))TAB(NH(I))FG$(I):N
   EXT                                          :rem 243
100 FOR I=0 TO 4:NV(I)=NV(I)+DI(I):IF NV(I)>23 OR
    {SPACE}NV(I)<0 THEN GOSUB 400               :rem 130
105 FOR J=10 TO 11:NV(J)=NV(J)+DV(J):IF NV(J)>23 O
    R NV(J)<0 THEN GOSUB 420                    :rem 251
106 NH(J)=NH(J)+DH(J):IF NH(J)>38 OR NH(J)<0 THEN
    {SPACE}GOSUB 470                            :rem 179
107 PRINT V$(XV(J))TAB(XH(J))BL$V$(NV(J))TAB(NH(J)
    )FG$(J)                                     :rem 253
108 XH(J)=NH(J):XV(J)=NV(J):NEXT                :rem 186
110 PRINT V$(XV(I))TAB(NH(I))BL$V$(NV(I))TAB(NH(I)
    )FG$(I):XV(I)=NV(I):NEXT                    :rem 94
120 FOR I=5 TO 9:NH(I)=NH(I)+DI(I):IF NH(I)>38 OR
    {SPACE}NH(I)<0 THEN GOSUB 450               :rem 97
125 FOR J=10 TO 11:NH(J)=NH(J)+DH(J):IF NH(J)>38 O
    R NH(J)<0 THEN GOSUB 470                    :rem 194
126 NV(J)=NV(J)+DV(J):IF NV(J)>23 OR NV(J)<0 THEN
    {SPACE}GOSUB 420                            :rem 240
127 PRINT V$(XV(J))TAB(XH(J))BL$V$(NV(J))TAB(NH(J)
    )FG$(J)                                     :rem 255
128 XH(J)=NH(J):XV(J)=NV(J):NEXT                :rem 188
130 PRINT V$(NV(I))TAB(XH(I))BL$V$(NV(I))TAB(NH(I)
    )FG$(I):XH(I)=NH(I):NEXT                    :rem 68
140 GOTO 100                                    :rem 95
300 FOR I=0 TO 4:NH(I)=3+I*8:NH(I+5)=NH(I):NV(I)=3
    +I*4:NV(I+5)=NV(I):NEXT                     :rem 116
310 FOR I=0 TO 9:XV(I)=NV(I):XH(I)=NH(I):NEXT
                                                :rem 100
320 V$(0)="{HOME}":FOR I=1 TO 23:V$(I)=V$(I-1)+"
    {DOWN}":NEXT                                :rem 250
```

```
340 FG$(0)="{RED}Q":FG$(1)="{PUR}Z":FG$(2)="{BLK}A
    ":FG$(3)="{GRN}{B}":FG$(4)="{BLU}X"    :rem 144
350 FOR I=0 TO 4:FG$(I+5)=FG$(I):NEXT      :rem 242
360 FOR I=0 TO 9 STEP 2:DI(I)=1:DI(I+1)=-DI(I):NEX
    T                                      :rem 22
370 FOR I=10 TO 11:NH(I)=8:NV(I)=19:XV(I)=19:XH(I)
    =8:FG$(I)="{2}+":NEXT                  :rem 108
380 DV(10)=1:DV(11)=-1:DH(10)=-1:DH(11)=1  :rem 113
390 RETURN                                 :rem 124
400 DI(I)=-DI(I):NV(I)=XV(I)+DI(I):RETURN  :rem 181
420 DV(J)=-DV(J):NV(J)=XV(J)+DV(J):RETURN  :rem 227
450 DI(I)=-DI(I):NH(I)=XH(I)+DI(I):RETURN  :rem 158
470 DH(J)=-DH(J):NH(J)=XH(J)+DH(J):RETURN  :rem 162
```

Smooth Movement

The movement in all these routines has been jerky. This is 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 stages of movement. For instance, if you were going to PRINT the leftward movement of a unicycle, you might have one character that contained the whole unicycle. The next step would be two characters: the left one would contain one-fourth of the unicycle, while the right one contained three-fourths. The next pair would each contain half; the next half would show the movement three-fourths complete; and at last you're back to the starting position.

On some computers, this is the only way to get smooth movement. On the 64, however, you have sprites. They move one dot at a time, so that their animation is very smooth. When you need the effect of smooth animation, sprites will be your first choice.

Fast but Dumb: Moving in a Group

Often, though, you won't need smoothness so much as quantity. There is almost no limit to the number of characters you can have moving around on the screen, but there *is* a limit to the number of sprites you can have.

If your figures move independently, as they have so far, you have to expect them to move fairly slowly. Twelve on the screen at the same time was none too quick in BASIC. Even when they only move in one line, either horizontally or vertically, as long as you have to do edge-checking for each character independently, you are not going to have speed.

There's no law that says figures have to move independently, however. If you put the characters together in a group and move them with a single control loop, you can gain back a lot of speed. Think of the rows of descending aliens in *Space Invaders*. The game was *not* easy, though its day in the arcades has passed; in BASIC you can program the same sort of movement pattern.

Program 5-7 shows something like the *Space Invaders* movement pattern. First, rows of aliens are set up, using the playing card symbols (hearts, spades, clubs, and diamonds) and a few other symbols from the built-in graphics character set. Each row begins with a CURSOR DOWN and contains exactly enough characters, including blanks, to fill a 40-character row. This way, the entire block can be PRINTed in sequence, very quickly. All that needs to change is the starting point of the first string; the rest follow in sequence.

The movement is controlled in two FOR-NEXT loops. First, a positive loop takes the block of characters from left to right. Then a negative loop (STEP −1) takes the group from right to left again. When they reach the left-hand edge, the screen is cleared and the starting position for the two loops is dropped down one line.

Later, in Chapter 13, we'll use this program as the starting point for "Card Invaders," a target shooting game that demonstrates missiles.

Program 5-7. Moving as a Group

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM FG$(6),F$(6),V$(9)                          :rem 192
20 GOSUB 300                                       :rem 117
30 BL$=" ":BC=1                                    :rem 195
40 POKE 53281,BC:POKE 53280,BC:PRINT "{CLR}":rem 1
100 FOR X=0 TO 8:PRINT "{CLR}"                     :rem 182
110 FOR I=0 TO 15:PRINT V$(X)TAB(I);:FOR J=0 TO 6:
    PRINT FG$(J);:NEXT:NEXT                        :rem 191
120 FORI=14 TO 0 STEP-1:PRINT V$(X)TAB(I);:FORJ=0
    {SPACE}TO 6:PRINT FG$(J);:NEXT:NEXT            :rem 89
130 NEXT X                                         :rem 43
140 GOTO 100                                       :rem 95
300 F$(0)="Q":F$(1)="S":F$(2)="A":F$(3)="Z":F$(4)=
    "X":F$(5)="{RVS} {OFF}":F$(6)="£"              :rem 92
310 FORI=0 TO 6:FG$(I)="{DOWN}"                    :rem 33
```

```
320 FORJ=0 TO 7:FG$(I)=FG$(I)+" "+F$(I)+" ":NEXT
                                              :rem 160
330 FG$(I)=FG$(I)+"{16 SPACES}":NEXT          :rem 81
340 V$(0)="{HOME}":FOR I=1 TO 9:V$(I)=V$(I-1)+"
    {DOWN}":NEXT                              :rem 208
390 RETURN                                    :rem 124
```

Animation

In Chapter 4 we created custom characters so we could get exactly the shape we wanted on the screen. In this chapter we've moved characters around on the screen. Now it's time to put those together, and *change* the shape on the screen in order to make figures more realistic, which will help make the game world more real.

The basic technique of animation is to switch from one shape to another to give the illusion of reality within the figure. The two (or more) shapes must be different, or no movement will seem to take place. However, the shapes must also be similar enough that the player will realize that they are meant to represent the same figure.

This balance between similarity and difference requires some planning and thought. An explosion, for instance, radically changes the shape of the figure, but you can get away with it because it happens in the same *place* as the figure that exploded, and because (usually) the player has seen the event that caused the explosion. Even so, the most effective explosion is one in which the actual shape of the exploding spaceship can be seen to break apart, with individual pieces floating away.

The player must always be able to recognize the figure, either by its shape or by its location. You can transform the shape as long as you make it clear that it's really the same object. Or you can move the object as long as you keep the shape clearly similar.

Complex Characters and Direction Changes

Let's start with a fairly simple animation problem. Program 5-8 creates a dragon which will move back and forth across the screen. The dragon is a complex character. It was drawn facing left. This is fine, as long as the dragon is moving left. But we want it to go back and forth across the screen, from one edge to the other. We need the dragon to face right, too.

The easiest method is to create characters representing the left-facing dragon, and other characters representing the right-facing dragon. You can create both dragons separately on graph paper, or you can use the method in this program. The routine from 300 to 370 READs the same DATA for both the left-facing dragon and right-facing dragon. But as it READs the DATA for the right-facing dragon (lines 340-370), it first flips every byte of every character pattern. This is time-consuming in the setup, but it does make sure that the right-facing dragon is an exact mirror image of the left-facing one.

The two complex characters, one for either direction, are set up as a two-element string array, DR$(*n*), in lines 400-430. Each dragon image consists of three rows of four characters each. Between the rows are the CURSOR DOWN and CURSOR LEFT instructions to get from the end of one row to the start of the next. There is a crucial difference between the handling of the two strings. The left-moving dragon has a blank space at the right-hand end of each row of the complex character. The right-moving dragon has a blank space at the left-hand edge of each row. Since the dragon will always move one column at a time, this column of blanks will erase the column of characters left behind when the dragon moved. It saves our having to erase the old image separately, and helps to reduce flicker.

If the dragon were to be moved up and down, too, you would need to add rows of blanks that will be PRINTed *behind* the character, depending on its direction of movement. It would be possible to create upward- and downward-flying images, too.

You might also notice that in the main movement loop, the rightward movement loop goes from 0 to 35, and the leftward STEPs backward (−1) from 35 to 0. Even though the screen is 39 columns wide, 35 is the rightmost position we can use. This is because the dragon is four characters wide, and the string is positioned from its upper left-hand corner. (We are also not using column 39, as is usual with PRINT movement routines.)

If you'd like to see how the characters are actually set up, first RUN the program, then press, RUN/STOP-RESTORE. The screen will return to blue and the regular character set. Then type GOTO 100. Now you'll see the rows of letters moving across the screen.

When the program runs, the dragon's movement is far more realistic, because now it faces the direction it is moving. It still isn't full animation since it doesn't seem to move, but it's an improvement over having only one image.

Program 5-8. Back and Forth Dragon

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM DR$(1)                                    :rem 119
20 CC=14:CH=CC*1024                               :rem 3
30 PRINT "{1}{CLR}":POKE 53272,(PEEK(53272)AND 2
   40)OR CC:POKE 53281,0                          :rem 2
40 GOSUB 300:GOSUB 400                           :rem 197
50 SPEED=99                                      :rem 133
97 REM                                            :rem 84
98 REM MOVE LEFT TO RIGHT                        :rem 216
99 REM                                            :rem 86
100 FOR I=0 TO 35:PRINT DOWN$TAB(I)DR$(1)          :rem 9
110 FOR WA=0 TO SPEED:NEXT:NEXT                  :rem 132
117 REM                                          :rem 125
118 REM MOVE RIGHT TO LEFT                         :rem 1
119 REM                                          :rem 127
120 FOR I=35 TO 0 STEP -1:PRINT DOWN$TAB(I)DR$(0)
                                                 :rem 164
130 FOR WA=0 TO SPEED:NEXT:NEXT                  :rem 134
140 GOTO 100                                      :rem 95
297 REM                                          :rem 134
298 REM MAKE MONSTER SET                         :rem 185
299 REM                                          :rem 136
300 FOR I=0 TO 11:T=I*8:FOR J=CH+T TO CH+T+7
                                                 :rem 108
310 READ N:POKE J,N:NEXT:NEXT                     :rem 29
317 REM                                          :rem 127
318 REM MAKE MIRROR-IMAGE SET                    :rem 245
319 REM                                          :rem 129
320 RESTORE:FOR H=0 TO 8 STEP 4:FOR I=0 TO 3:FOR J
    =0 TO 7                                       :rem 57
330 T=CH+184-(((I+8-H))*8)+J                     :rem 146
337 REM                                          :rem 129
338 REM BYTE-FLIPPING ROUTINE                     :rem 98
339 REM                                          :rem 131
340 READ N:FOR K=0 TO 7:N=N*2                     :rem 35
350 IF N>255 THEN W=W+2↑K:N=N-256                 :rem 76
360 NEXT K:POKE T,W:W=0:NEXT:NEXT:NEXT           :rem 204
370 FOR I=256+CH TO 263+CH:POKE I,0:NEXT I:RETURN
                                                  :rem 56
397 REM                                          :rem 135
```

```
398 REM SET UP STRINGS                              :rem 67
399 REM                                             :rem 137
400 DOWN$="{HOME}{16 DOWN}"                         :rem 148
410 DR$(0)="@ABC {DOWN}{5 LEFT}DEFG {DOWN}{5 LEFT}
    HIJK "                                          :rem 215
420 DR$(1)=" LMNO{DOWN}{5 LEFT} PQRS{DOWN}{5 LEFT}
     TUVW "                                         :rem 105
430 RETURN                                          :rem 119
497 REM                                             :rem 136
498 REM DATA FOR DRAGON                              :rem 69
499 REM                                             :rem 138
500 DATA 0,0,0,1,15,28,8,9                          :rem 229
510 DATA 16,48,240,152,248,124,206,135               :rem 75
520 DATA 8,132,70,39,55,55,55,119                    :rem 95
530 DATA 0,0,0,192,128,128,192,248                  :rem 130
540 DATA 0,0,2,1,5,2,0,0                            :rem 113
550 DATA 3,7,15,30,30,190,95,15                     :rem 236
560 DATA 175,159,255,96,0,0,0,128                    :rem 91
570 DATA 240,240,248,255,48,0,2,194                 :rem 188
580 DATA 0,0,4,11,1,2,0,0                           :rem 164
590 DATA 23,27,61,224,0,0,12,19                     :rem 230
600 DATA 241,255,254,248,225,98,194,161             :rem 144
610 DATA 226,49,25,50,194,9,21,226                  :rem 140
```

Character Set Flipping

To create much more realistic movement, we need something that will change the shapes far more often than just when the figure reaches the edge of the screen and changes direction. You could create two figures the way we did with the dragon, and switch back and forth between them each time it is PRINTed. This would be fairly simple—you could do it by PRINTing the array with a toggle switch variable, like this:

```
K=-(K=0):PRINT FG$(K)
```

Every time through the loop, the variable K will switch from 0 to 1 or from 1 to 0. That way you will PRINT alternate figure strings.

This is not a bad method, but it can require an awful lot of programming to set up all the string arrays. And if you try to do it with more than one or two characters on the screen, you'll find that your program slows down unbearably. Worst of all, any character you animate using this method must be tracked constantly through all its movements. This means your loops will get quite long, and movement will be slow. You have to rePRINT or rePOKE every animated character, every time!

Fortunately, there is a way to animate entire screen displays, with hundreds of characters at once—and it only takes a single POKE each time through the main loop. Not a single POKE for each character—a single POKE for the entire screen. Like all special effects, it takes some setting up—but it's less time-consuming than setting up dozens of string arrays and tracking them all over the screen. The effect can be dazzling, without costing your game any speed during play.

The technique is *character set flipping*. You start out by setting up two (or more) custom character sets. One might start at the 12K boundary and the other at the 14K boundary. Any characters that you don't want to animate must be contained in *both* sets, exactly alike. Any characters that you do want to animate must be different in the two sets. Then you POKE a character set instruction into location 53272 each time through the loop, alternating between the two sets.

When you change that single pointer at 53272, the effect is instantaneous. Every character that is different in the two sets changes at once; and every character that is the same in both sets remains the same. This demonstration program shows the basic technique. Only two custom characters are different between the two character sets. So those two characters are the only ones that seem to move. The other characters on the screen give no sign that their patterns are coming from somewhere else. The illusion of movement is complete. And it's so fast that if you don't have a delay loop in the program, the animation will turn into a blur.

Program 5-9. Two Character Sets

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
5 T=160                                          :rem 93
10 DIM CC(1),CH(1),CB(1)                        :rem 175
15 CC(0)=12:CC(1)=14                            :rem 241
20 FOR I=0 TO 1:CH(I)=1024*CC(I):CB(I)=(PEEK(53272
   )AND 240)OR CC(I):NEXT                       :rem 131
25 GOSUB 1100                                   :rem 169
30 FOR I=0 TO 1:FOR J=27 TO 28:FOR K=0 TO 7
                                                :rem 165
40 READ A:POKE CH(I)+8*J+K,A:NEXT:NEXT:NEXT
                                                :rem 116
100 X$="£ [ £ [ £ [ £ [ £ [ ":R$="-----------
    WATCH THE CHANGES-----------"               :rem 129
```

```
105 PRINT "{HOME}";:FOR I=Ø TO 11:PRINT X$X$R$:NEX
    T                                          :rem 63
110 FOR I=Ø TO 1:POKE 53272,CB(I):FOR J=Ø TO T:NEX
    T:NEXT:GOTO 110                            :rem 128
1100 POKE 56334,PEEK(56334)AND 254:POKE 1,PEEK(1)A
     ND 251                                    :rem 227
1110 FOR I=Ø TO 1:RM=53248-CH(I)               :rem 162
1120 FOR J=CH(I) TO CH(I)+511:POKE J,PEEK(J+RM):NE
     XT:NEXT                                   :rem 60
1130 POKE 1,PEEK(1) OR 4:POKE 56334,PEEK(56334)OR
     {SPACE}1                                  :rem 180
1140 RETURN                                    :rem 166
1300 DATA Ø,Ø,3,3,3,3,Ø,Ø                      :rem 158
1310 DATA Ø,12,18,34,36,24,Ø,Ø                 :rem 165
1320 DATA 192,192,Ø,Ø,Ø,Ø,192,192              :rem 68
1330 DATA Ø,Ø,48,72,68,36,24,Ø                 :rem 183
```

Movement and Animation Together

Now let's look at this with moving figures. Notice that the animation requires no tracking whatsoever, and no arrays. Once the two sets are in memory, it is just a single POKE each time through the movement loop. It makes for a much more interesting display than our earlier screens.

Program 5-10. Two Sets Animation

```
1 T=3                                          :rem 245
5 POKE 53281,7:POKE 53280,7:PRINT "{23}"       :rem 56
10 DIM CC(1),CH(1),CB(1),S$(23),A$(T),X(T),Y(T),XD
   (T),YD(T)                                   :rem 108
15 CC(Ø)=12:CC(1)=14                           :rem 241
18 REM                                         :rem 77
19 REM SET UP TWO CHARACTER SETS               :rem 165
20 FOR I=Ø TO 1:CH(I)=1024*CC(I):CB(I)=(PEEK(53272
   )AND 240)OR CC(I):NEXT                      :rem 131
30 FOR I=Ø TO 1:FOR J=27 TO 29:FOR K=Ø TO 7
                                               :rem 166
40 READ A:POKE CH(I)+8*J+K,A:NEXT:NEXT:NEXT
                                               :rem 116
50 FOR I=Ø TO 1:FOR J=256 TO 263:POKE CH(I)+J,Ø:NE
   XT:NEXT                                     :rem 170
58 REM                                         :rem 81
59 REM DEFINE CURSOR MOVEMENTS                 :rem 153
60 GOSUB 1100                                  :rem 168
68 REM                                         :rem 82
69 REM SET STARTING POSITIONS                  :rem 115
70 FOR I=Ø TO T:GOSUB 940:A$(I)=CHR$(91+R):NEXT:X$
   =" "                                        :rem 177
```

```
75 FOR I=Ø TO T:GOSUB 95Ø:X(I)=R:GOSUB 96Ø:Y(I)=R:
   NEXT                                          :rem 161
8Ø FOR I=Ø TO T:GOSUB 98Ø:XD(I)=R:GOSUB 98Ø:YD(I)=
   R:NEXT                                         :rem 42
9Ø PRINT "{CLR}"                                 :rem 2Ø5
98 REM                                            :rem 85
99 REM MAIN MOVEMENT LOOP                         :rem 32
1ØØ FOR I=Ø TO T:K=-(K=Ø)                         :rem 29
1Ø8 REM                                          :rem 125
1Ø9 REM CHECK FOR EDGES                           :rem 43
11Ø IF X(I)<1 OR X(I)>37 THEN GOSUB 3ØØ          :rem 253
12Ø IF Y(I)<1 OR Y(I)>22 THEN GOSUB 32Ø          :rem 252
123 REM                                          :rem 122
124 REM ERASE OLD POSITIONS                      :rem 146
125 PRINT S$(Y(I))TAB(X(I))X$                    :rem 118
128 REM                                          :rem 127
129 REM SET NEW POSITION VALUES                  :rem 155
13Ø X(I)=X(I)+XD(I):Y(I)=Y(I)+YD(I)              :rem 213
138 REM                                          :rem 128
139 REM PRINT AT NEW POSITIONS                    :rem 85
14Ø PRINT S$(Y(I))TAB(X(I))A$(I)                 :rem 246
148 REM                                          :rem 129
149 REM ANIMATE BY SWITCHING SETS                 :rem 11
15Ø POKE 53272,CB(K):NEXT                        :rem 142
16Ø GOTO 1ØØ                                      :rem 97
298 REM                                          :rem 135
299 REM REBOUND FROM SIDE                        :rem 24Ø
3ØØ XD(I)=-XD(I)                                 :rem 1Ø5
3Ø3 REM                                          :rem 122
3Ø4 REM SET REBOUND DIRECTION                     :rem 23
3Ø5 IF(Y(I)>Ø)AND(Y(I)<38) THEN ON YD(I)+2 GOSUB 9
    9Ø,98Ø,97Ø:YD(I)=R                           :rem 178
31Ø RETURN                                       :rem 116
318 REM                                          :rem 128
319 REM REBOUND FROM TOP OR BOTTOM                :rem 45
32Ø YD(I)=-YD(I)                                 :rem 1Ø9
323 REM                                          :rem 124
324 REM SET REBOUND DIRECTION                     :rem 25
325 IF(X(I)>Ø)AND(X(I)<23) THEN ON XD(I)+2 GOSUB 9
    9Ø,98Ø,97Ø:XD(I)=R                           :rem 17Ø
33Ø RETURN                                       :rem 118
938 REM                                          :rem 136
939 REM RANDOMIZE CHARACTER                      :rem 191
94Ø R=INT(3*(RND(9))):RETURN                     :rem 158
948 REM                                          :rem 137
949 REM SET RANDOM STARTING POSITIONS            :rem 1Ø7
95Ø R=1+INT(RND(9)*36):RETURN                    :rem 224
96Ø R=1+INT(RND(9)*21):RETURN                    :rem 219
968 REM                                          :rem 139
```

```
969 REM RANDOMIZE REBOUND DIRECTION          :rem 229
970 R=INT(2*RND(9)):RETURN                    :rem 79
980 R=INT(2*RND(9)):IF R=0 THEN R=-1         :rem 218
985 RETURN                                   :rem 134
990 R=-INT(2*RND(9)):RETURN                  :rem 126
1098 REM                                     :rem 182
1099 REM SET UP CURSOR POSITIONS             :rem 238
1100 S$(0)="{HOME}":FOR I=1 TO 23:S$(I)=S$(I-1)+"
     {DOWN}":NEXT:RETURN                      :rem 56
1998 REM                                     :rem 191
1999 REM DATA FOR CHARACTER SET 1            :rem 107
2000 DATA 0,66,60,60,60,60,66,0              :rem 224
2010 DATA 24,24,36,36,66,66,129,129          :rem 191
2020 DATA 0,12,18,34,68,72,48,0              :rem 232
2998 REM                                     :rem 192
2999 REM DATA FOR CHARACTER SET 2            :rem 109
3000 DATA 0,60,126,126,126,126,60,0          :rem 161
3010 DATA 24,24,24,24,36,36,36,36             :rem 78
3020 DATA 0,48,72,68,34,18,12,0              :rem 233
```

Here is the same routine, but with an improved illusion of fast movement. Instead of moving every character each time through the movement loop, a random number is generated, which points to one character. That character is moved, and no others. This means that sometimes a single character can move in a burst of speed, while at other times it can rest. The unpredictabililty makes the movement more exciting to watch. And with the animation going on constantly, the characters that aren't moving from place to place still seem to be "alive."

Program 5-11. Two Sets Random

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
1 T=9                                        :rem 251
5 POKE 53280,9 :POKE 53281,9:PRINT "{WHT}":rem 172
10 DIM CC(1),CH(1),CB(1),S$(23),A$(T),X(T),Y(T),XD
   (T),YD(T)                                 :rem 108
15 CC(0)=12:CC(1)=14                         :rem 241
18 REM                                        :rem 77
19 REM SET UP TWO CHARACTER SETS             :rem 165
20 FOR I=0 TO 1:CH(I)=1024*CC(I):CB(I)=(PEEK(53272
   )AND 240)OR CC(I):NEXT                    :rem 131
30 FOR I=0 TO 1:FOR J=27 TO 29:FOR K=0 TO 7
                                             :rem 166
40 READ A:POKE CH(I)+8*J+K,A:NEXT:NEXT:NEXT
                                             :rem 116
```

```
50 FOR I=0 TO 1:FOR J=256 TO 263:POKE CH(I)+J,0:NE
   XT:NEXT                                        :rem 170
58 REM                                             :rem 81
59 REM DEFINE CURSOR MOVEMENTS                     :rem 153
60 GOSUB 1100                                      :rem 168
68 REM                                             :rem 82
69 REM SET STARTING POSITIONS                      :rem 115
70 FOR I=0 TO T:GOSUB 940:A$(I)=CHR$(91+R):NEXT:X$
   =" "                                            :rem 177
75 FOR I=0 TO T:GOSUB 950:X(I)=R:GOSUB 960:Y(I)=R:
   NEXT                                            :rem 161
80 FOR I=0 TO T:GOSUB 980:XD(I)=R:GOSUB 980:YD(I)=
   R:NEXT                                          :rem 42
90 PRINT "{CLR}"                                   :rem 205
98 REM                                             :rem 85
99 REM MAIN MOVEMENT LOOP                          :rem 32
100 K=-(K=0):I=INT((T+1)*RND(9))                   :rem 228
108 REM                                            :rem 125
109 REM CHECK FOR EDGES                            :rem 43
110 IF X(I)<1 OR X(I)>37 THEN GOSUB 300            :rem 253
120 IF Y(I)<1 OR Y(I)>22 THEN GOSUB 320            :rem 252
123 REM                                            :rem 122
124 REM ERASE OLD POSITIONS                        :rem 146
125 PRINT S$(Y(I))TAB(X(I))X$                      :rem 118
128 REM                                            :rem 127
129 REM SET NEW POSITION VALUES                    :rem 155
130 X(I)=X(I)+XD(I):Y(I)=Y(I)+YD(I)                :rem 213
138 REM                                            :rem 128
139 REM PRINT AT NEW POSITIONS                     :rem 85
140 PRINT S$(Y(I))TAB(X(I))A$(I)                   :rem 246
148 REM                                            :rem 129
149 REM ANIMATE BY SWITCHING SETS                  :rem 11
150 POKE 53272,CB(K)                               :rem 21
160 GOTO 100                                       :rem 97
190 GOTO 100                                       :rem 100
298 REM                                            :rem 135
299 REM REBOUND FROM SIDE                          :rem 240
300 XD(I)=-XD(I)                                   :rem 105
303 REM                                            :rem 122
304 REM SET REBOUND DIRECTION                      :rem 23
305 IF(Y(I)>0)AND(Y(I)<38) THEN ON YD(I)+2 GOSUB 9
    90,980,970:YD(I)=R                             :rem 178
310 RETURN                                         :rem 116
318 REM                                            :rem 128
319 REM REBOUND FROM TOP OR BOTTOM                 :rem 45
320 YD(I)=-YD(I)                                   :rem 109
323 REM                                            :rem 124
324 REM SET REBOUND DIRECTION                      :rem 25
325 IF(X(I)>0)AND(X(I)<23) THEN ON XD(I)+2 GOSUB 9
    90,980,970:XD(I)=R                             :rem 170
```

```
330 RETURN                                          :rem 118
938 REM                                             :rem 136
939 REM SET RANDOM CHARACTER VALUES                 :rem 147
940 R=INT(RND(9)*3):RETURN                           :rem 77
948 REM                                             :rem 137
949 REM SET RANDOM STARTING POSITIONS               :rem 107
950 R=1+INT(RND(9)*36):RETURN                       :rem 224
960 R=1+INT(RND(9)*21):RETURN                       :rem 219
968 REM                                             :rem 139
969 REM RANDOMIZE REBOUND DIRECTION                 :rem 229
970 R=INT(2*RND(9)):RETURN                           :rem 79
980 R=INT(2*RND(9)):IF R=0 THEN R=-1                :rem 218
985 RETURN                                          :rem 134
990 R=-INT(2*RND(9)):RETURN                         :rem 126
1098 REM                                            :rem 182
1099 REM SET UP CURSOR POSITIONS                    :rem 238
1100 S$(0)="{HOME}":FOR I=1 TO 23:S$(I)=S$(I-1)+"
     {DOWN}":NEXT:RETURN                             :rem 56
1998 REM                                            :rem 191
1999 REM DATA FOR CHARACTER SET 1                   :rem 107
2000 DATA 0,66,60,60,60,60,66,0                     :rem 224
2010 DATA 24,24,36,36,66,66,129,129                 :rem 191
2020 DATA 0,12,18,34,68,72,48,0                     :rem 232
2998 REM                                            :rem 192
2999 REM DATA FOR CHARACTER SET 2                   :rem 109
3000 DATA 0,60,126,126,126,126,60,0                 :rem 161
3010 DATA 24,24,24,24,36,36,36,36                    :rem 78
3020 DATA 0,48,72,68,34,18,12,0                     :rem 233
```

Moving the Entire Screen

There is still another technique that is used in games like *Scramble, Cobra, Defender,* and *Zaxxon*—the scrolling screen. The player-figure is a craft of some kind—a rocket, a helicopter—and it seems to be moving over a landscape of obstacles and enemies. Actually, the player-figure has only limited movement, if any. Often the player-figure stays in one place, but seems to be moving because the screen display "moves" behind it.

This technique is called scrolling. It's as if you were filming an old-time western, with the hero sitting on a mechanical horse while a long painting is unrolled at one end and wound up at the other, so that he seems to be passing by the rocks and trees and mountains. To the observer, limited by what can be seen on the screen, it's impossible to tell which is moving, the hero or the background.

Screen movement is programmed on the 64 exactly the

way that a character is moved—the old display is replaced by a new display that is only slightly different from the old one. However, the number of bytes involved—1000 every time you scroll—is far too many to handle screen scrolling in BASIC. Remember how long it takes to copy 64 characters from the ROM set into another area of memory? Now imagine going through that long a delay each time the screen scrolls. Scrolling—especially fine scrolling—requires machine language to be effective on the 64.

With one exception. There is a built-in machine language scrolling routine. It works only in one direction, but it is very fast. That is the built-in scrolling feature when you try to do a CURSOR DOWN on the bottom line of the screen. The whole display jumps up one row. Instantly. Once again, using PRINT, you can get machine language speed in your BASIC program.

Program 5-12 combines animation with scrolling and character movement to give us the basis of a scrolling space game. The program is based on the random starfield generating program from Chapter 3, but now it uses custom characters for the large and small stars. By flipping character sets, it makes the small stars twinkle and the large stars pulsate. There is also a starship that has animated exhaust coming out of it, and a space station that spins rapidly.

BASIC scrolling. At the beginning of the main loop, the program checks for the value of TI$, the built-in timer feature of 64 BASIC. If enough time has passed since the last time the screen scrolled, the program jumps to a scrolling routine at line 600.

The actual scrolling is easy. The variable M$ contains exactly enough CURSOR DOWN characters to scroll the screen one line. Just PRINT M$, and the whole screen moves up a line.

The problem is that the top line of stars now disappears, and the bottom row of the screen is blank. We have to have a way of putting random star patterns on that new line at the bottom, so that the starfield doesn't empty out.

New random lines. If we generated a new line of stars by doing 40 RND functions, one for every character position on the line, we would lose all the speed of scrolling—we'd have a long delay waiting for the new line to fill up.

There is a way around it, however. At lines 400-420, during the program setup, we created the string T$. T$

consists of the maximum number of characters possible in a 64 string—255 of them. They are randomly generated, with more blanks likely than stars, and more small stars than large stars, just like the starfield.

Now when the screen scrolls, we'll PRINT a 40-character section of that long randomly generated string. And we'll make each new line different by randomly selecting the *starting point* of the section we'll PRINT. That means we only generate *one* random number each time the screen scrolls, and then PRINT using a MID$ section starting at the position of the random number we generated. The random number has to be at least 1 and no larger than the length of the string minus 40.

Increasing difficulty. For the first time, we can start building difficulty levels into a game program. At first, the interval between scrolls is long, but we can decrease it a tiny bit each time we scroll, until the new lines are appearing quite rapidly. This will make the game harder and harder.

Also, we can make it so that at higher levels, the screen will scroll more than one line at a time. All we have to do is PRINT an 80-character section of the string instead of a 40-character section. Now two lines will jump up. This program includes both features. It will seem to be moving slowly when it starts, but if you watch patiently you'll see it pick up speed until it flies along. As each new difficulty level comes, the starship will change colors. Also, every now and then a space station will be POKEd onto the screen at a random position in the new line.

"Moving" a stationary figure. The starship on the screen actually does not move, but we still have to provide a movement routine for it. Why? Because when the screen scrolls, the starship character will scroll with it. If we were using a sprite, of course, it would not happen—but we aren't using sprites yet. So each time the screen scrolls, the starship character is erased (replaced by whatever was under it), the screen is scrolled, and the starship is POKEd back into its old location. Color memory also has to be erased and POKEd. Even though the starship is still at exactly the same position in memory, it seems to be moving through the stars.

Reconfigured video memory. This program relocates video memory, putting the new video block at 32768. This is the same 16K section of RAM where BASIC itself sits, using the upper 8K. So two character sets and screen memory will both

be located in the lower 8K of the new video block.

You'll notice the difference at once, because a strange group of odd-looking characters will appear on the screen as soon as you RUN the program. They look odd because the VIC-II is now looking for character memory in a different block, and the character set isn't there. Later, just before the new screen display is created, they'll turn to garbage again.

The only problem with this comes at the end of the program. If you stop the program with RUN/STOP-RESTORE, you'll think your computer has gone crazy. You'll type, but instead of the letters you typed, you'll see completely unrelated characters—or no characters at all. This is because even though RUN/STOP-RESTORE sets the video block back to normal, it doesn't tell BASIC where screen memory is. BASIC is still looking up around 32768 for screen memory, though of course it has no trouble finding color memory, which never moved.

To get things back to normal, type SHIFT-CLEAR. Then type POKE 648,4. You won't be able to see the letters you type, but BASIC will see them and act accordingly. This tells BASIC to find screen memory back at 1024.

This program will be expanded into a complete game in Chapter 9, called "Mission: Nova!," with the starship under the player's control. The objective will be to collide with the large stars while avoiding the small ones. Still later, we'll add sound, introductions, and farewells. It will be one of our first playable games.

Program 5-12. Flickering Stars

Remember, do not type the checksum number at the end of each line. For example, do not type *:rem 123*. Please read Appendix J, "Automatic Proofreader," before entering this program.

```
10 DIM CC(1),CH(1),CB(1)                          :rem 175
17 REM                                             :rem 76
18 REM SET VALUES FOR MEMORY VARIABLES             :rem 98
19 REM                                             :rem 78
20 CC(0)=12:CC(1)=14:VB=32768:VM=VB/256:SB=128:SC=
   SB*256:CM=55296                                 :rem 166
30 FOR I=0 TO 1:CH(I)=1024*CC(I)+VB:CB(I)=CC(I)+16
   *(SB-VM):NEXT                                   :rem 229
37 REM                                             :rem 78
38 REM SET VIDEO, SCREEN, & CHAR BLOCKS            :rem 204
39 REM                                             :rem 80
40 POKE 648,SB:POKE 56578,PEEK(56578)OR 3:POKE5657
   6,(PEEK(56576)AND252)OR 1                       :rem 167
```

```
50 PRINT "{CLR}JUST A MOMENT, PLEASE . . ."
                                                :rem 144
57 REM                                           :rem 80
58 REM INITIALIZE SCREEN, CHAR SET               :rem 57
59 REM                                           :rem 82
60 GOSUB 1100:GOSUB 400:GOSUB 500                :rem 69
67 REM                                           :rem 81
68 REM INITIALIZE DISPLAY VARIABLES             :rem 243
69 REM                                           :rem 83
70 FG=31:OC=1:NC=2:Q=3:QQ=Q+1:GOSUB 800          :rem 28
80 C=500:W=30:SS=27:CS=2:TS=0:POKE CM+C,NC:POKE SC
   +C,FG                                         :rem 11
97 REM                                           :rem 84
98 REM MAIN LOOP                                :rem 180
99 REM                                           :rem 86
100 FOR NC=3 TO 15:X=0                          :rem 129
107 REM                                         :rem 124
108 REM TIME TO SCROLL THE SCREEN?              :rem 254
109 REM                                         :rem 126
110 IF VAL(TI$)>Q THEN GOSUB 600                :rem 234
117 REM                                         :rem 125
118 REM SWITCH CHAR SETS: ANIMATION             :rem 135
119 REM                                         :rem 127
120 K=-(K=0):POKE 53272,CB(K)                    :rem 10
127 REM                                         :rem 126
128 REM END OF MAIN LOOP                         :rem 74
129 REM                                         :rem 128
130 FOR I=0 TO 49:NEXT                          :rem 186
140 IF X<10 THEN 110                            :rem 218
150 NEXT NC:END                                 :rem 119
397 REM                                         :rem 135
398 REM SET UP THE STRING OF NEW STARS          :rem 221
399 REM                                         :rem 137
400 T$="":FOR I=0 TO 254:T=32                   :rem 210
410 IF INT(RND(9)*7)<1 THEN T=92:IF INT(RND(9)*5)<
    1 THEN T=93                                  :rem 50
420 T$=T$+CHR$(T):NEXT                           :rem 13
430 M$=CHR$(19):FOR I=0 TO 24:M$=M$+CHR$(17):NEXT:
    RETURN                                      :rem 178
497 REM                                         :rem 136
498 REM GENERATE THE STARFIELD                   :rem 83
499 REM                                         :rem 138
500 POKE 53281,0:POKE 53280,0:PRINT "{WHT}{CLR}"
                                                :rem 141
505 FOR I=CM TO CM+999:POKE I,1:NEXT             :rem 40
510 Q=100*RND(9)+20:Q1=6*RND(9)+2               :rem 254
520 FOR I=0 TO Q:X=1000*RND(9):N=28:GOSUB 540:NEXT
                                                 :rem 75
530 FOR I=0 TO Q1:X=1000*RND(9):N=29:GOSUB 540:NEX
    T:RETURN                                    :rem 152
```

```
540 POKE SC+X,N:RETURN                                  :rem 117
597 REM                                                 :rem 137
598 REM SCROLL THE SCREEN                               :rem 250
599 REM                                                 :rem 139
600 POKE SC+C,W:POKE CM+C,OC:T=1+INT(RND(9)*167)
                                                        :rem 10
605 PRINT M$;MID$(T$,T,40*INT(QQ-Q)-1);:P=P-INT(7*
    Q):GOSUB 800                                        :rem 78
607 REM                                                 :rem 129
608 REM SAVE CHARACTER BEFORE SCROLL                    :rem 192
609 REM                                                 :rem 131
610 W=PEEK(SC+C)                                        :rem 165
617 REM                                                 :rem 130
618 REM PUT SPACESHIP IN SAME LOCATION                  :rem 50
619 REM                                                 :rem 132
620 POKE SC+C,FG:POKE CM+C,NC                           :rem 168
627 REM                                                 :rem 131
628 REM TIME FOR A SPACE STATION?                       :rem 168
629 REM                                                 :rem 133
630 TS=TS+1:IF TS>10*(QQ-Q)THEN GOSUB 850               :rem 118
640 X=X+1                                               :rem 227
690 RETURN                                              :rem 127
797 REM                                                 :rem 139
798 REM RESET THE TIMER                                 :rem 113
799 REM                                                 :rem 141
800 TI$="000000":Q=99*(Q/100):RETURN                    :rem 218
847 REM                                                 :rem 135
848 REM PUT ON A SPACE STATION                          :rem 237
849 REM                                                 :rem 137
850 TS=0:T2=1001-INT(RND(9)*40):POKE SC+T2,SS:POKE
    CM+T2,CS:RETURN                                     :rem 221
1097 REM                                                :rem 181
1098 REM COPY ROM CHARACTER SET                         :rem 88
1099 REM                                                :rem 183
1100 POKE 56334,PEEK(56334)AND 254:POKE 1,PEEK(1)A
    ND 251:POKE 657,0                                   :rem 74
1107 REM                                                :rem 173
1108 REM COPY FIRST 64 CHARACTERS TWICE                 :rem 55
1109 REM                                                :rem 175
1110 FOR I=0 TO 1:RM=53248-CH(I)                        :rem 162
1120 FOR J=CH(I) TO CH(I)+511:POKE J,PEEK(J+RM):NE
    XT:NEXT                                             :rem 60
1127 REM                                                :rem 175
1128 REM PUT CUSTOM CHARACTERS IN PLACE                 :rem 96
1129 REM                                                :rem 177
1130 FOR I=0 TO 1:FOR J=27 TO 31:FOR K=0 TO 7
                                                        :rem 1
1140 READ N:POKE CH(I)+8*J+K,N:NEXT:NEXT:NEXT
                                                        :rem 240
```

```
1150 POKE 1,PEEK(1)OR 4:POKE 56334,PEEK(56334)OR 1
                                                :rem 182
1160 POKE 53272,CB(0):RETURN                     :rem 70
1297 REM                                        :rem 183
1298 REM DATA FOR CUSTOM CHARACTERS             :rem 116
1299 REM                                        :rem 185
1300 DATA 24,12,14,12,48,112,48,24              :rem 117
1310 DATA 0,0,40,16,40,0,0,0                     :rem 50
1320 DATA 0,60,126,126,126,126,60,0             :rem 164
1330 DATA 0,32,116,249,159,46,4,0                :rem 78
1340 DATA 211,137,153,255,153,24,24,36           :rem 72
1350 DATA 0,32,112,241,143,14,4,0                :rem 56
1360 DATA 0,0,16,40,16,0,0,0                     :rem 58
1370 DATA 0,66,60,60,60,60,66,0                 :rem 233
1380 DATA 0,32,112,241,143,14,4,0                :rem 59
1390 DATA 203,145,153,255,153,24,24,36           :rem 77
```

6 Sprites

Sprites

Made for Games

One of the Commodore 64's most powerful game-designing tools is its sprite animation abilities. These sprites, also called MOBs (for Movable Object Blocks), are in effect graphics blocks which you can sculpt into any shape and move about the screen. They move independently of the main screen image, move quite smoothly, and if only a few are on the screen at once, move quickly enough for an arcade-style game, even though only BASIC is used.

Sprites can save a lot of programming because they are movable blocks. To create a custom character of the same size, move it from one location to another, and move it smoothly, would take much longer in terms of program length. Each pixel would need a location in memory, and each of these pixels would have to change its memory location each time the character moved. A sprite can do all this with much less effort on your part.

Because sprites are projected independently of any characters already on the screen, you can create the illusion of three dimensions in your game. Using standard or custom characters, you have to redraw any figure that is passed over by another character. But sprites do not erase the other characters when they move past them, so three-dimensional movement can be simulated by moving a sprite in front of, or behind, another figure.

The computer also handles collisions between sprites, or between a sprite and another character. This collision detection is a powerful tool in its own right, and will be handled in Chapter 9. Sprites can also be displayed in color, or even in multicolored mode.

You'll have eight sprites to work with on the 64, which should be enough for almost any game you want to design. Many times, however, you'll not use all eight, for as you increase the number of sprites on the screen, movement slows down. This is one of the drawbacks of sprite use. It is not something at fault with the sprites, but with BASIC. Oftentimes, game programmers use machine language routines to move sprites. Chapter 7 includes some excellent machine language subroutines to move your sprites more rapidly.

Drawing Sprites

Drawing a sprite is much like drawing a custom character—it must be *drawn* in memory. The difference is that a sprite is a much larger character, consisting of a block 24 pixels wide by 21 pixels high. Each sprite occupies 63 bytes of memory (24 bits * 21 bits = 504 bits/8 bits per byte = 63 bytes) and uses a 64-byte block.

There are utility programs available which can help you draw sprites. They handle the more tedious tasks of creating sprites, for they allow you to draw a sprite and then count the values for the DATA statements you'll need to display them. One such utility, "Commodore 64 Sprite Editor," appeared in the December 1982 issue of COMPUTE!. Sprite editors will make it easier to design these characters, but you can do the same thing using Figure 6-1, a Sprite Design Worksheet.

Notice that the worksheet shows the entire sprite area, 24 columns wide by 21 rows high. The 24 columns have been divided into 3 sections of 8 columns each. Just as in creating custom characters, each memory location is made up of 8 bits. Each group of 8 columns on the worksheet, then, corresponds to one memory location used when creating a sprite. For instance, columns 0 through 7 of the first row will be represented by one memory location in the 64. Again, just as with custom characters, you'll need to figure out the value of each of these groups of 8 bits by adding the values of each *on* bit. Bit values have been included on the worksheet, near the top, to make this computation easier.

A completed sprite will have 63 bit values, since there are 21 rows with 3 memory locations in each row (21*3=63). As with custom characters, the sprite is drawn using a READ statement which looks to several DATA statements in the program. Computing the numbers for the DATA statements and placing them in the correct order are essential for drawing the sprite.

Let's go ahead and draw a sprite. Using the Sprite Design Worksheet, simply sketch the outline of your sprite, filling in the blocks that you want to be *on*. It could look something like Figure 6-2.

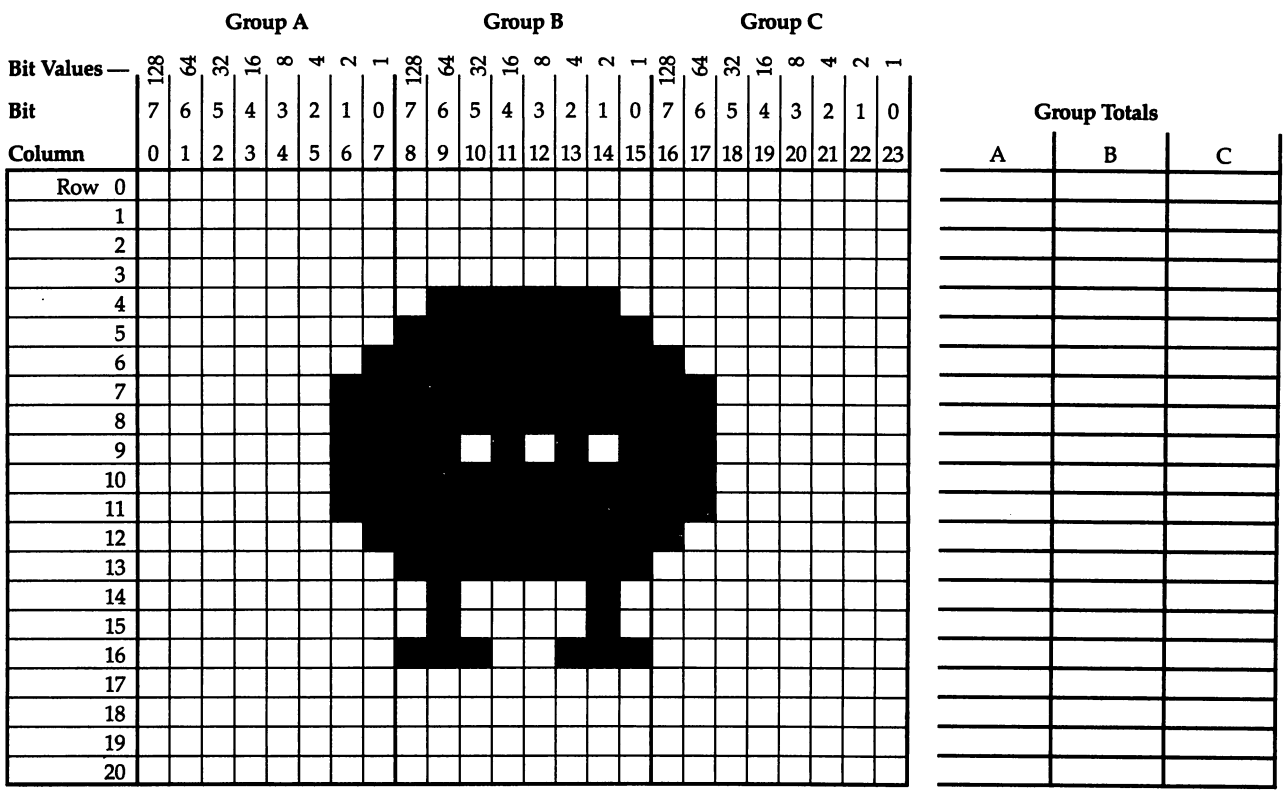
Figure 6-1. Sprite Design Worksheet

	Group A								Group B								Group C							
Bit Values —	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Row 0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								

Group Totals

A	B	C

Figure 6-2. Sprite Sketch



The blocks that are filled in are the pixels that will be *on* when the sprite is displayed. As when creating custom characters, a 1 in a particular bit becomes an *on* pixel in the sprite. A 0 becomes an *off* pixel by displaying it in the background color. To show some pixels *on* and others *off*, you simply add the bit values at the top of the Worksheet, counting only those bits which show pixels *on*. Take a look at Figure 6-3, which is the same sprite with its bit values calculated.

The first four rows in this sprite have none of their bits turned *on*. The bit values for all 12 bytes in the first 4 rows are thus 0. Row 4 is the first row which shows any pixels *on*. In this row, the first group of 8 bits has none *on*, so its value is 0. The second group has 6 bits on, bits 1 through 6. Adding the bit values gives you a total of 126 (64 + 32 + 16 + 8 + 2). The third group of 8 bits in row 4 has no bits *on*, and so has a value of 0.

Each of the bytes in the sprite is calculated like this; remember that each row consists of three bytes, and that each byte must be figured separately. The worksheet makes this easier, since you can calculate each byte's value and then enter it at the right in the appropriate group total column.

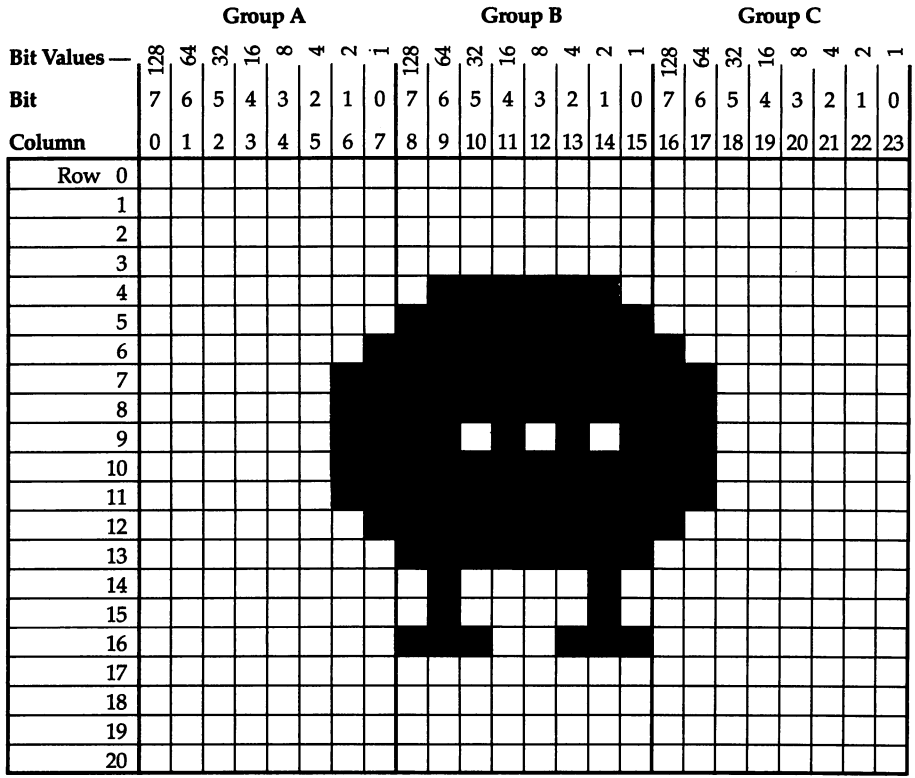
Go through your own sprite design and compute the bit values of each of the bytes. Make sure that the 0 values are entered where all bits are *off*, and that each row's bytes are in the correct order. This is important, for when you create the DATA statements, your sprite will not appear as expected if the numbers are not in the right order.

When you've finished, you should have 63 numbers for each sprite, the numbers starting with the top-left corner of the sprite design and moving across each row, then to the leftmost byte of the row below. The last number represents the byte in the bottom-right corner. These numbers will be placed in DATA statements within your program so that the computer can READ them and then display the sprite on the screen.

The sprite we've drawn, for example, would have these DATA statements:

```
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 126, 0, 0, 255, 0, 1, 255,
128
DATA 3, 255, 192, 3, 255, 192, 3, 213, 192, 3, 255, 192, 3, 255,
192, 1, 255, 128, 0, 255, 0
DATA 0, 66, 0, 0, 66, 0, 0, 231, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0
```

Figure 6-3. Sprite Sketch with Bit Values



	Group A								Group B								Group C							
Bit Values —	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Row 0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								

Group Totals

A	B	C
0	0	0
0	0	0
0	0	0
0	0	0
0	126	0
0	255	0
1	255	128
3	255	192
3	255	192
3	213	192
3	255	192
3	255	192
1	255	128
0	255	0
0	66	0
0	66	0
0	231	0
0	0	0
0	0	0
0	0	0
0	0	0

These DATA statements can be anywhere in the program, just as DATA statements needed to create custom characters. Many programmers place them at the end of the program, where they are out of the way, so that they are not altered unintentionally.

Each sprite that you design for your game is created this way. But once you have the sprite designed, you have to POKE other values into the 64 to make it appear.

Creating Your Sprite

Sprite memory. Once you have the DATA statements organized for your sprite, you have to find a place to store the numbers, as well as point the 64 to the correct location so that it can find those values.

The VIC-II chip can access only 16K of the 64's memory at one time, and three things have to be in the same 16K section for the computer to work as we want it to. The screen memory, the character generator, and the sprite information must all appear in the same 16K block of the 64.

Within a 16K block, we could conceivably place 128 sprites. You cannot use all 128 positions, of course, for the screen memory and character generator take up some of the available memory. Depending on the 16K block chosen, other locations may be forbidden. For example, the block normally chosen by the 64 is the one accessed from 0 to 16383. Much of the first 1024 bytes is occupied by a BASIC work area; screen memory is normally 1024 to 2023; the character generator appears at locations 4096 to 8191; and everything above 2048 that isn't used by the character generator is used to store your BASIC program. There's not a lot of room for sprite information.

If you wanted to draw several sprites, you would need to do one of two things. First, you could move BASIC so that it starts at a higher location. You've seen how to do this in Chapter 4 when you created custom characters. The second way to create additional space is to use a different 16K block, one that is not quite so busy with BASIC operations. Keep these things in mind as you develop your own sprites, especially if you're planning on using lots of them in your game. For the moment, however, we'll stay with the first 16K block. It's easiest to work with, and we want to keep things simple.

Locating sprite DATA. If you're using three sprites or less, a convenient place to locate the sprite data information is in the cassette tape buffer. As long as you're not using the cassette tape for memory storage, the locations from 832 to 1022 can be used for your sprite information. One sprite would be placed at locations 832-894, a second sprite at 896-958, and a third at locations 960-1022.

If you're using a cassette tape, however, or planning on more than three sprites, you'll have to locate the sprite information at a different address. A popular location among programmers, and one suggested by the *Commodore 64 Programmer's Reference Guide*, is address 12288. This location is high enough in the first 16K block to normally be safe from a BASIC program. However, to protect this area from BASIC, you may want to POKE the line:

```
POKE 52,48:POKE 56,48
```

This line, once it's included in your program, moves the BASIC pointer and protects the addresses starting at 12288 from any BASIC program. You're almost ready to locate your sprite information. Only a few more things to consider.

The memory locations which are involved with sprite programming run from 53248 to 53294. Each location sets one of the VIC-II chip's *registers*, making different things happen to the computer. Some set sprite color, while others check for collisions between sprites. A memory map of the 64 will show you each register's exact description. The *Programmer's Reference Guide* has such a map.

The first location, 53248, is important when programming sprites, for it is a value often used when setting color and screen location. You'll use it many times, so it's a good idea to memorize at least this one location concerning sprites.

POKEing in sprites. POKEing in sprites on the 64 is a step-by-step process. Once you have your sprites designed and the accompanying DATA statements prepared, you need to POKE a series of values into the computer's memory. You'll want to do this as quickly and as easily as you can. The following table gives you the necessary information you'll need to do this for your eight sprites. Although you can certainly spend some time to look at this table, you'll probably find it more worthwhile to refer to it as you work with your own sprite creations.

Table 6-1. Sprite POKE Table

	SPRITE 0	SPRITE 1	SPRITE 2	SPRITE 3	SPRITE 4	SPRITE 5	SPRITE 6	SPRITE 7
Put in Memory (Set Pointers)	2040, 192	2041, 193	2042, 194	2043, 195	2044, 196	2045, 197	2046, 198	2047, 199
Locations for Sprite Pixel (12288–12798)	12288 to 12350	12352 to 12414	12416 to 12478	12480 to 12542	12544 to 12606	12608 to 12670	12672 to 12734	12736 to 12798
Turn On Sprite	V+21,1	V+21,2	V+21,4	V+21,8	V+21,16	V+21,32	V+21,64	V+21,128
Sprite Color	V+39,C	V+40,C	V+41,C	V+42,C	V+43,C	V+44,C	V+45,C	V+46,C
Set X Position (0–255)	V+0,X	V+2,X	V+4,X	V+6,X	V+8,X	V+10,X	V+12,X	V+14,X
Set X Position Right of Seam (0–255)	V+16,1 V+0,X	V+16,2 V+2,X	V+16,4 V+4,X	V+16,8 V+6,X	V+16,16 V+8,X	V+16,32 V+10,X	V+16,64 V+12,X	V+16,128 V+14,X
Set Y Position	V+1,Y	V+3,Y	V+5,Y	V+7,Y	V+9,Y	V+11,Y	V+13,Y	V+15,Y
Expand Sprite Horizontally	V+29,1	V+29,2	V+29,4	V+29,8	V+29,16	V+29,32	V+29,64	V+29,128
Expand Sprite Vertically	V+23,1	V+23,2	V+23,4	V+23,8	V+23,16	V+23,32	V+23,64	V+23,128
Turn On Multicolor Mode	V+28,1	V+28,2	V+28,4	V+28,8	V+28,16	V+28,32	V+28,64	V+28,128
Multicolor 1	V+37,C	V+37,C	V+37,C	V+37,C	V+37,C	V+37,C	V+37,C	V+37,C
Multicolor 2	V+38,C	V+38,C	V+38,C	V+38,C	V+38,C	V+38,C	V+38,C	V+38,C
Set Sprite Priority over Background	V+27,1	V+27,2	V+27,4	V+27,8	V+27,16	V+27,32	V+27,64	V+27,128

The first step is to actually place your sprite information in one of the available memory location areas. As we said before, the easiest location in the first 16K block begins at address 12288. You can place the information for up to 10 sprites in this area before you start running into the section reserved for the custom character set you developed in Chapter 4.

To begin locating your sprite information, you could type the lines:

```
10 PRINT "{CLR}":V=53248                          :rem 152
20 POKE 52,48:POKE 56,48                          :rem 252
30 FOR X=0 TO 62:READ A:POKE 12288+X,A:NEXT
                                                  :rem 136
```

These clear the screen, set BASIC's pointers and protect the memory locations above address 12288 from BASIC, and begin READing the DATA for the sprite. Notice that the variable *V* is set as the beginning of the sprite registers. It is simpler to set this variable, and use the form V + *n* to POKE particular values into each register, than to memorize each register's address.

Now that you have the sprite information beginning at location 12288, you have to tell the 64 where that information is. Take a look at the Sprite POKE Table. To set the 64's pointers so that it can locate the information for the first sprite (sprite 0), you have to POKE location 2040 with a value of 192. This sets the sprite pointer so that it now looks to location 12288 for its data. To do this, you add:

```
40 POKE 2040,192
```

The next step is to turn on or *enable* your 0 sprite. As you can see from the Sprite POKE Table, this is done by typing in:

```
50 POKE V+21,1
```

This allows you to see the sprite on the screen. V + 21 is the control register at location 53269 (V = 53248: V + 21 = 53269). You'll notice that all the sprites, from 0 to 7, are turned on with the same register. The only difference is the number that is POKEd into that location. When you turn on the sprite, make sure you're POKEing in the correct value, and not turning on a sprite you're not using.

You can turn on more than one sprite with a single POKE statement. If you were using sprites 0, 1, 2, and 3, for instance, all you would do is add together the values you POKE in. 1 + 2 + 4 + 8 = 15, which is the number you'd POKE into location V + 21. Just as simple is the way you turn off selective sprites. If you wanted to turn off sprite 2, for instance, during a game, all you would have to do is use:

POKE V + 21,11

The value 11 is simply 15 (the previous total value) − 4 (the value POKEd in for sprite 2). This is a common technique used when calculating sprite collisions, and we'll look at it more closely in the next chapter.

If you were to RUN the program at this point, even if you added the DATA statements for your sprite, you would not be able to see the sprite on the screen. That's because the sprite would appear in the background color, and because it has no

screen location. To set the color, refer to the Sprite POKE Table.

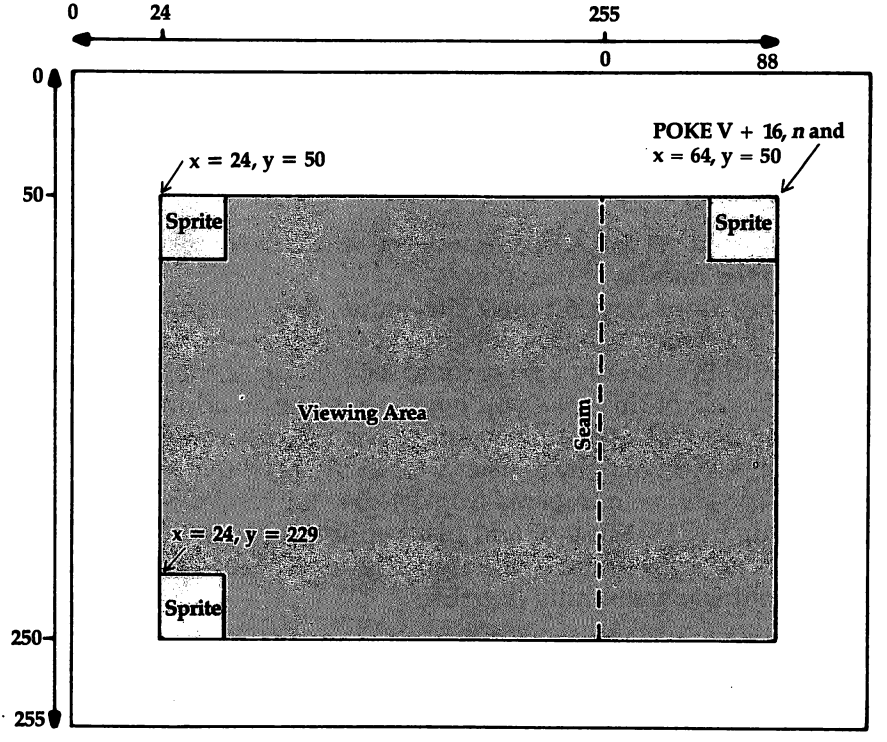
```
6Ø POKE V+39,1
```

This POKE statement sets the color of the 0 sprite to white. The same values used to change screen color are used to color the sprites. Each sprite has its own register address to set color. Sprite 6, for example, uses V + 45 as its address when selecting color.

Finally, the sprite's screen location must be set so that it will display.

Positioning the sprite on the screen. The screen display is divided into a grid of X and Y coordinates, just like a graph. The X coordinate is the *horizontal* position, while the Y coordinate is the *vertical* position. Take a look at Figure 6-4 for a moment.

Figure 6-4. Screen Display



To place a sprite on the screen, you'll have to POKE in two settings, both the X and Y position. This tells the 64 where to position the upper-left corner of the sprite. This is important, for since the sprite is 24 pixels across by 21 down, you have to make sure it's positioned correctly. If you place it in the upper-left corner of the screen display, for instance, it will run 24 pixels to the right from there, 21 pixels down for that starting location.

Notice that the screen viewing area is smaller than the area of possible sprite location. If you place your sprite at X position 0 and Y position 0, it will not appear on the screen. The first location that the sprite *will* show on the screen is X position 24, Y position 50.

To display a particular sprite, you must POKE the X and Y settings. Each sprite has its own unique POKEs that must be used. Refer to the Sprite POKE Table for these values when you set your own sprite locations.

Setting the X position. The possible values for X are 0 to 255, counting from left to right. Values from 0 to 23 will place all or part of the sprite outside the viewing area, as you can see from Figure 6-4. To POKE in the 0 sprite, for example, you could use something like:

```
70 POKE V+0,180
```

which would set the X position of the sprite to somewhere close to the middle of the screen.

X positions beyond the 255th value. Because the viewing area of the screen extends beyond the 255th position (and since you can POKE in only values up to 255), to place the sprite on the right side of this *seam* you'll have to place an additional POKE in your program. Each sprite has its own value that must be POKEd into the register V + 16. This gives you 65 additional X positions, numbered from 0 to 64. You can POKE in values up to 255, but it would place your sprite off the viewing area. To set the 0 sprite in the right section of the screen, then, you could use something like this:

```
70 POKE V+16,1:POKE V+0,32
```

which would locate the sprite's X position at the midpoint of the section on the right side of the seam.

Setting the Y position. Placing the Y position for a sprite is much like placing the X position. A different POKE is used,

each unique to a particular sprite, and values from 0 to 255 (counting from top to bottom) are allowed. Only the values from 50 to 229, however, will place the entire sprite in the viewing area. Using the 0 sprite again, you could type:

```
80 POKE V+1,120
```

which would place the sprite about midway down the screen viewing area. There is no *seam* on the Y position.

The First Sprite

You've designed and created your first sprite. To actually show it on the screen, you only need to add the DATA statements you made when you designed your sprite. Using the example sprite designed earlier, you could enter:

```
200 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,126,0,0,255,0,1
    ,255,128
210 DATA 3,255,192,3,255,192,3,213,192,3,255,192,3
    ,255,192,1,255,128,0,255,0
220 DATA 0,66,0,0,66,0,0,231,0,0,0,0,0,0,0,0,0,0,0
    ,0,0
```

Now you have a short program which should create and display a sprite. The complete program below is simply a compilation of all the various POKEs necessary to create a sprite.

Program 6-1. Sprite #1

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 PRINT"{CLR}":V=53248                          :rem 152
20 POKE 52,48:POKE 56,48                         :rem 252
30 FOR X=0 TO 62:READ A:POKE 12288+X,A:NEXT
                                                 :rem 136
40 POKE 2040,192                                  :rem 33
50 POKE V+21,1                                   :rem 213
60 POKE V+39,1                                   :rem 223
70 POKE V+0,180                                   :rem 12
80 POKE V+1,120                                    :rem 8
200 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,126,0,0,255,0,1
    ,255,128                                     :rem 185
210 DATA 3,255,192,3,255,192,3,213,192,3,255,192,3
    ,255,192,1,255,128,0,255,0                   :rem 146
220 DATA 0,66,0,0,66,0,0,231,0,0,0,0,0,0,0,0,0,0,0
    ,0,0                                         :rem 236
```

Expanding sprites. The VIC-II has the ability to expand a sprite in the horizontal direction, in the vertical direction, or in both at the same time. This is a feature of sprite creation that allows you to create large figures on the screen without having to design custom characters, or even without combining two sprites. When a sprite is expanded, each dot becomes twice as tall or twice as wide. The sprite's resolution doesn't increase; it simply gets larger.

As with other sprite POKE commands, each of the eight sprites uses its own unique register value. To expand your first sprite, sprite 0, for example, you would type the following line after you've loaded and run the short sprite creation program.

POKE V+29, 1

Because this is in direct mode, the change is only temporary. Your sprite is now double its previous width. To change it back to its original size, type:

POKE V+29, 0

Changing the vertical size of the sprite is just as easy. Simply type:

POKE V+23, 1

and the 0 sprite is twice as tall as it was before. To change it back, type:

POKE V+23, 0

Both the horizontal and vertical size can be expanded by combining the POKEs as:

POKE V+29, 1:POKE V+23, 1

You can refer to the Sprite POKE Table for the different values to POKE into the register for each sprite. Just as in turning sprites *on* or *off*, each bit of these two registers controls a different sprite. Make sure that you use the appropriate value for each sprite you're expanding.

Adding these POKEs to your program is easy. If you want to show the sprite doubled both horizontally and vertically, add:

```
90 POKE V+29,1:POKE V+23,1
```

Expanded sprites have many uses in game design. One possible application would be to simulate three dimensions by making a sprite expand as it moves, making it seem to close in on the player. Another use could be sprite animation, for

example, showing an explosion expanding, then contracting. You'll come up with other ideas for this use of sprites when you design your own games.

Multicolor Sprites

Multicolor sprites work much the same as multicolor characters. Bits within the sprite drawing are paired, and the combination of the *on* and *off* bits within each pair determines its color. Although designing the sprite may be slightly more difficult with multicolor, the effort is well worth it.

Let's look at this bit-pairing again. With only two bits, there are four possible combinations

Figure 6-5. Bit-Pairing

Bit Pairs

Just as with custom characters, this is how the 64 sets colors for multicolored sprites. Figure 6-6 shows what each bit-pair represents, along with the locations for alterating a sprite's colors.

Figure 6-6. Sprite Bit-Pairing

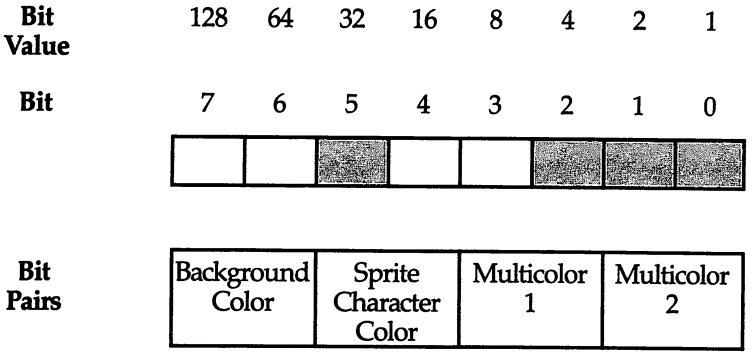
Bit Pairs

Background	POKE 53281,C
Multicolor 1	POKE V + 37,C
Sprite character color	POKE V + (39 to 46), C
Multicolor 2	POKE V + 38,C

(A 1 indicates an *on* bit, while 0 indicates an *off* bit.)

Keep in mind as you're designing multicolored sprites that these bits set the pixels as the sprite is drawn. Each colored *dot* will consist of two pixels, side by side. You have four colors to work with: background color (the present color of the screen), sprite character color, Multicolor 1, and Multicolor 2. Depending on which bit(s) are *on* or *off* in each pair, a different color is displayed in the two-pixel wide block. For example, look at Figure 6-7, a representation of one eight-pixel block of a sprite drawing.

Figure 6-7. Sprite Combinations



Bits 7 and 6 are both set to 0, or blank. This two-pixel block will show in the background color. The second block, bits 5 and 4, are *on* and *off* respectively. This pattern will display the sprite character color. The third pair, bits 3 and 2, are *off* and *on*, which gives that two-pixel block the color selected by Multicolor 1. Finally, bits 1 and 0 are both *on*. Multicolor 2 will be displayed in this block.

As you design your multicolored sprite, you need to indicate which bit-pairs will be displayed in which of the four colors. If the first eight-pixel block of your sprite is designed to look like Figure 6-7, for instance, you simply add the values of all the bits that are to be turned *on*. 32+4+2+1=39; that's the value you would place in the DATA statement for that first eight-pixel block of your sprite.

Creating an entire multicolored sprite is more complicated than figuring the value for only part of one row, of course. Designing a multicolored sprite has its own peculiarities, for

with dots two-pixels wide, resolution is cut by half. Although you can simply turn on the multicolored mode and use an existing sprite, sometimes it will be better to redesign the entire sprite so that it looks as you want it to. Figure 6-8, the Multicolor Sprite Design Worksheet, helps you do this in the same way the Sprite Design Worksheet did earlier in this chapter.

The main difference between the two worksheets is that the Multicolor Sprite Design Worksheet groups the pixels into paired blocks. The bit-pair patterns are also helpful when you're designing your own multicolored sprite. Look at Figure 6-9 for a moment. This shows the sprite designed earlier, now set up for multicolor mode. It is slightly different than the original, since bit patterns have to be set to show the various colors. The values you would place in a DATA statement are different as well, as you can see by comparing these numbers with the ones in Figure 6-3.

Figure 6-10 shows the same sprite in multicolor, in the way it would appear on your screen. Notice how the bit-pairs create different colors.

Simply changing the DATA statement numbers will not make your sprite display in multicolors, however.

Turning on the color. To show a sprite in multicolor, you first have to turn the mode on. Setting the multicolor mode is done by POKEing in a value to register V + 28. Each sprite has its own value to POKE to that register. The 0 sprite, for instance, would use a value of 1. Adding this line to the program you've already typed in would set the sprite to multicolored mode.

```
1ØØ POKE V+28,1
```

You now have to assign colors to Multicolor 1 and Multicolor 2 for your sprite. Before you do that, however, there are several things you should know. First of all, all the sprites displayed on the screen will share background color, Multicolor 1 and Multicolor 2. To make each sprite as different from the next as possible, the majority of a sprite's color should be set to the sprite character color. The more each sprite has of its own character color, the more different each will seem.

Another thing to consider is that Multicolor 1 is transparent to collision detection. If a sprite is composed of only Multicolor 1, no collisions will register between that sprite and

Figure 6-8. Multicolor Sprite Design Worksheet

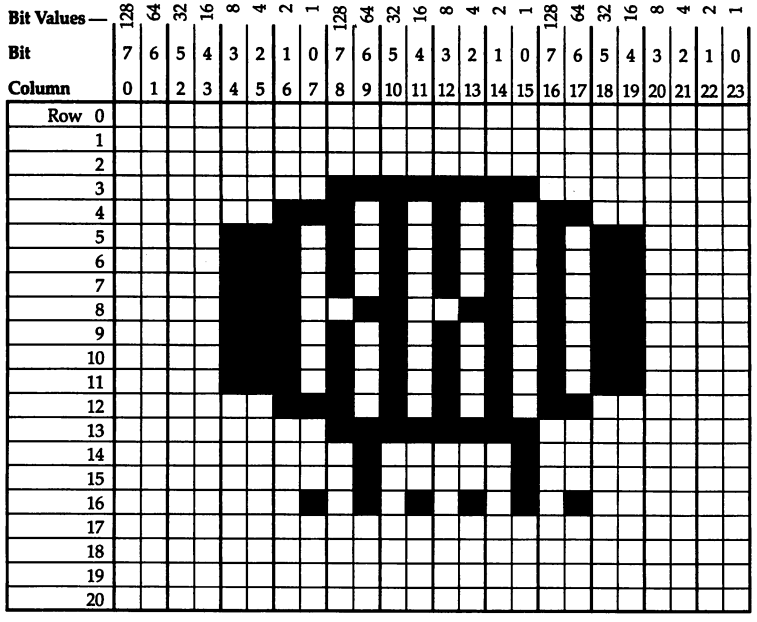
	Group A								Group B								Group C							
Bit Values—	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1	128	64	32	16	8	4	2	1
Bit	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0	7	6	5	4	3	2	1	0
Column	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23
Row 0																								
1																								
2																								
3																								
4																								
5																								
6																								
7																								
8																								
9																								
10																								
11																								
12																								
13																								
14																								
15																								
16																								
17																								
18																								
19																								
20																								

Group Totals

A	B	C

Bit Pattern	Color, Register	Controlling Location		
	– Background or screen color	53281	– Multicolor 1	V + 37
	– Sprite color	V + (39 to 46)	– Multicolor 2	V + 38

Figure 6-9. Multicolor Sprite Sketch

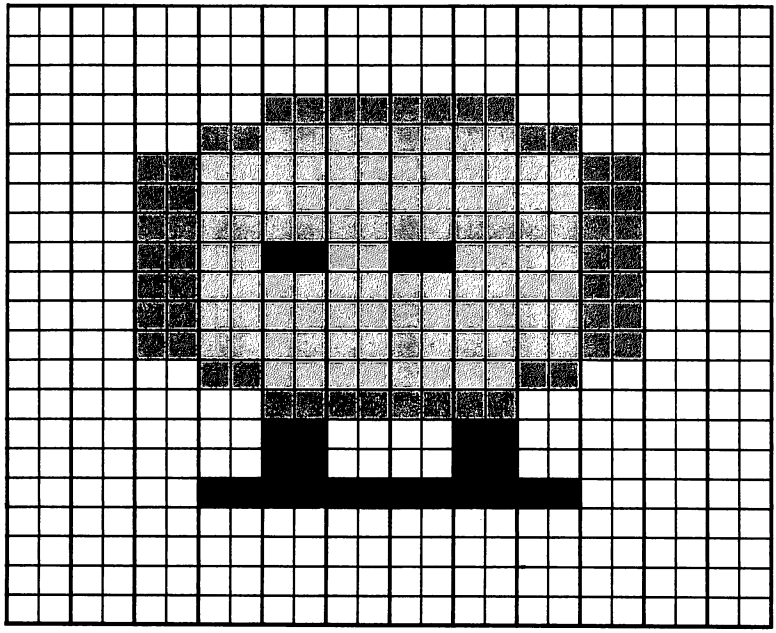


Group Totals

A	B	C
0	0	0
0	0	0
0	0	0
0	255	0
3	170	192
14	170	176
14	170	176
14	170	176
14	102	176
14	170	176
14	170	176
14	170	176
3	170	192
0	255	0
0	65	0
0	65	0
1	85	64
0	0	0
0	0	0
0	0	0
0	0	0

Bit Pattern	Color, Register	Controlling Location
□□	– Background or screen color	53281
■□	– Sprite color	V + (39 to 46)
□■	– Multicolor 1	V + 37
■■	– Multicolor 2	V + 38

Figure 6-10. Multicolor Sprite as It Appears on the Screen



Bit Pattern	Color, Register	Controlling Location
00	– Background or screen color	53281
01	– Sprite color	V + (39 to 46)
10	– Multicolor 1	V + 37
11	– Multicolor 2	V + 38

any other sprite. This can be a handy tool, for by creating background colors in Multicolor 1, you can make sure they are not detected in a collision.

You can also create interesting effects using the multicolor mode. One example would be a flashing window, or figure's eye. You can do this by changing the color of a block at regular intervals. All you have to do is POKE a different number into the register that controls one of the sprite colors.

Assigning colors to Multicolor 1 and 2 is as easy as assigning colors to the sprite's character color or to the screen background color. You use the same numbers as values to POKE in colors to the multicolor mode as you do for all other colors. To set Multicolor 1 to white and Multicolor 2 to yellow for our sample sprite, all you do is type:

```
11Ø POKE V+37,1:POKE V+38,7
```

You can change the values POKEd in to create other colors.

The following program creates a multicolored sprite, using the same steps we've already described. Notice that the DATA statements in lines 200-220 use the numbers from the newly created multicolored sprite. There are also two short delays, in lines 120 and 140, followed by color changes, to show you how to turn colors on and off.

Program 6-2. Multicolored Sprite

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
1Ø PRINT"{CLR}":V=53248                          :rem 152
2Ø POKE 52,48:POKE 56,48                         :rem 252
3Ø FOR X=Ø TO 62:READ A:POKE 12288+X,A:NEXT
                                                 :rem 136
4Ø POKE 2Ø4Ø,192                                  :rem 33
5Ø POKE V+21,1                                   :rem 213
6Ø POKE V+39,1                                   :rem 223
7Ø POKE V+Ø,18Ø                                   :rem 12
8Ø POKE V+1,12Ø                                    :rem 8
9Ø POKE V+23,1:POKE V+29,1                       :rem 141
1ØØ POKE V+28,1                                    :rem 8
11Ø POKE V+37,1:POKEV+38,7                       :rem 193
12Ø FOR T=Ø TO 5ØØ:NEXT                          :rem 236
13Ø POKE V+39,2                                   :rem 14
14Ø FOR T=Ø TO 5ØØ:NEXT                          :rem 238
15Ø POKE V+39,4                                   :rem 18
```

```
160 GOTO 120                                          :rem 99
200 DATA 0,0,0,0,0,0,0,0,0,0,255,0,3,170,192,14,17
    0,176,14,170,176                                  :rem 101
210 DATA 14,170,176,14,102,176,14,170,176,14,170,1
    76,14,170,176,3,170,192,0,255                     :rem 38
220 DATA 0,0,65,0,0,65,0,1,85,64,0,0,0,0,0,0,0,0
    ,0,0,0                                            :rem 88
```

By changing the numbers in the DATA statements, you can create your own multicolored sprites. Here are two more multicolored sprites you can try. All you need to do is replace the numbers in the program's DATA statements with these values.

Robot
DATA 0, 0, 0, 0, 0, 0, 0, 0, 0
DATA 0, 0, 0, 0, 168, 0, 0, 100, 4
DATA 0, 152, 8, 0, 168, 32, 0, 32, 128
DATA 10, 170, 128, 34, 186, 0, 34, 170, 0
DATA 34, 186, 0, 130, 170, 0, 66, 170, 0
DATA 2, 170, 0, 0, 136, 0, 0, 136, 0
DATA 0, 136, 0, 0, 136, 0, 2, 138, 0

Airplane
DATA 0, 0, 0, 0, 128, 0, 0, 144, 0
DATA 0, 160, 0, 0, 168, 0, 0, 168, 0
DATA 0, 168, 0, 0, 168, 1, 0, 168, 2
DATA 42, 170, 170, 234, 170, 170, 42, 170, 170
DATA 0, 168, 2, 0, 168, 1, 0, 168, 0
DATA 0, 168, 0, 0, 168, 0, 0, 160, 0
DATA 0, 144, 0, 0, 128, 0, 0, 0, 0

One of the interesting things you can do with sprites is change their colors. There are other ways besides the FOR-NEXT loop method to make changes like this, however. You could insert lines similar to these in the main loop of your program, so that every 15 times through the loop, colors of your sprites change:

C=C+1:IF C=15 THEN POKE V+39,1
IF C=30 THEN POKE V+39,7:C=0

Counter variables such as this can be set for many purposes, to change from one thing to another, or to turn registers on and off.

Sprite Parade

Examples of sprites are sometimes hard to find. For the beginning programmer, the first step of creating a sprite is oftentimes the hardest. To help in creating and designing your own, the program which follows displays nine sprites for you to look at. You can use them in your own game, or modify them if you wish.

The program shows each sprite moving from the bottom of the screen to the top. As each moves upward, it passes in front of the graphic display near the bottom and behind the screen title. Although these movements are covered more completely in the next chapter, you can get an idea of their effects in this program.

Each sprite's color is set randomly. Sprite 206 is the only one in multicolor mode. As you can see when you run the program, some of the color combinations look better than others. You can experiment even more by changing the screen background color to something other than black.

Take a look at the sprite patterns before you type the program in. The DATA statements are set to display these sprites, but you could change the numbers to show a modified picture.

Figure 6-11. Space Fighter

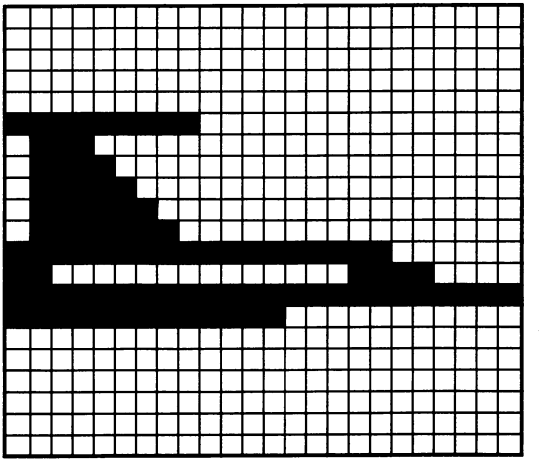


Figure 6-12. Alien Lander

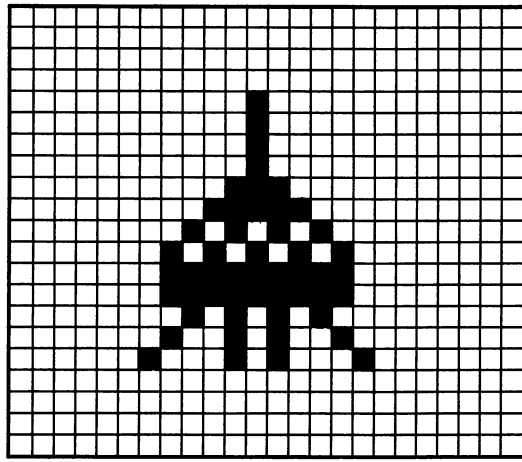


Figure 6-13. Rocket

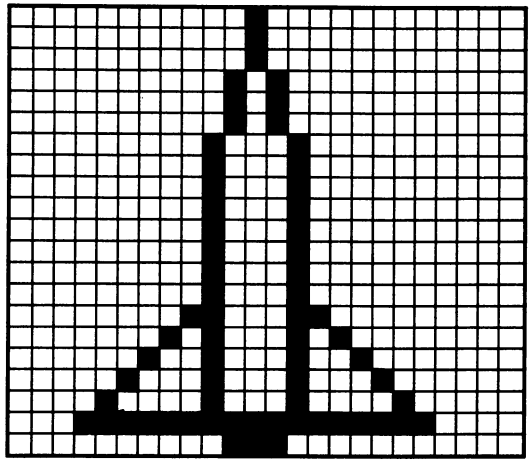


Figure 6-14. Wooden Block

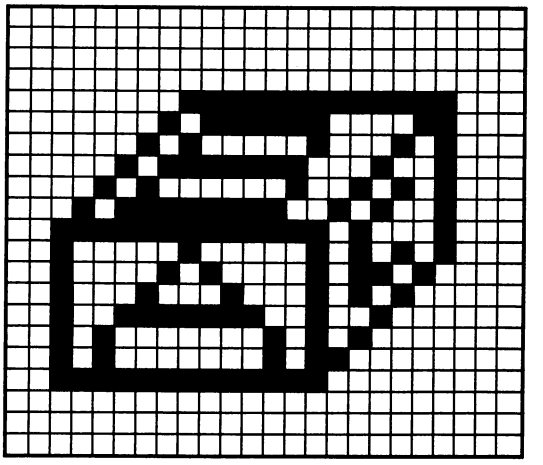


Figure 6-15. Space Destroyer

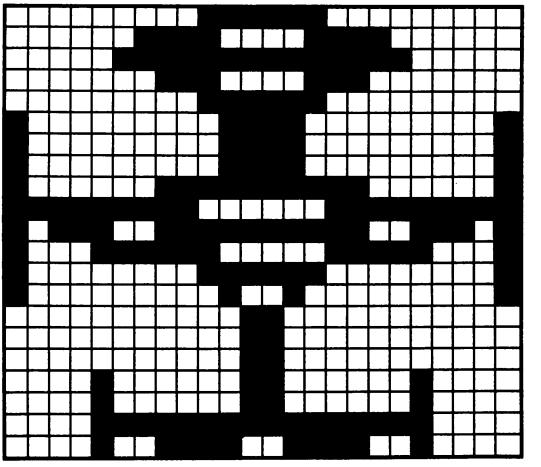


Figure 6-16. Enterprise

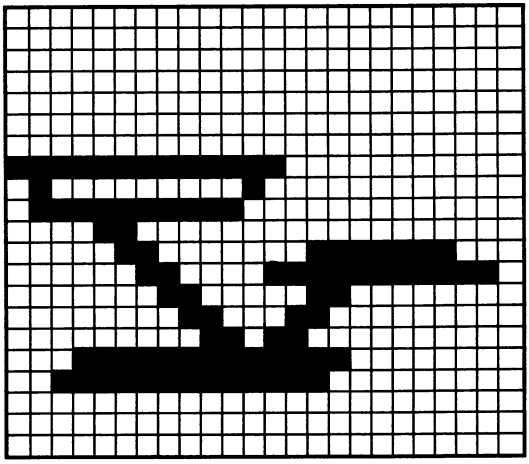


Figure 6-17. Alien Saucer in Multicolor

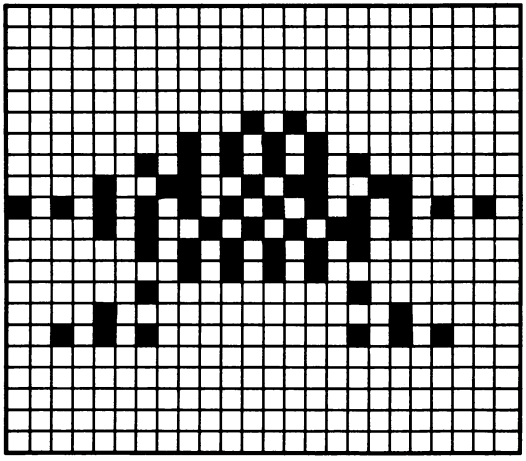


Figure 6-18. Alien Face

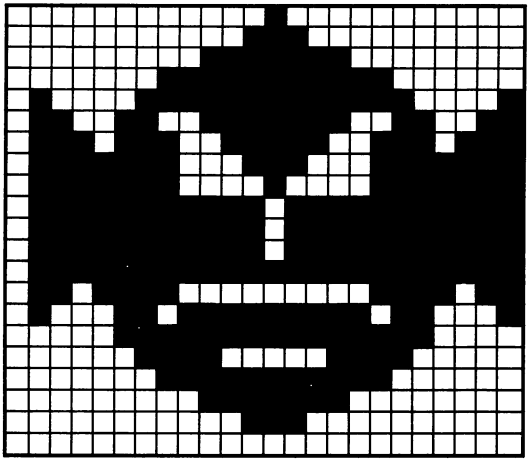
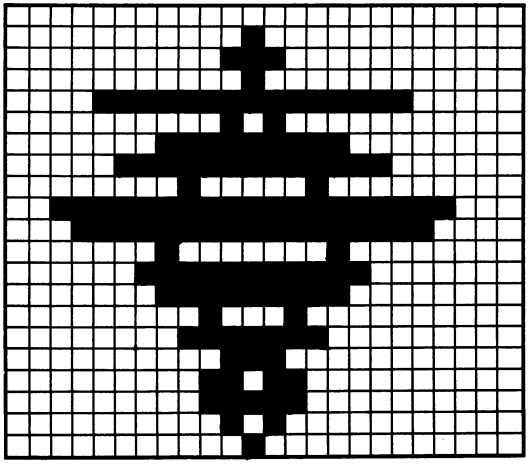


Figure 6-19. Space Fortress



Type in and run the program to see these nine sprites on the screen. An explanation of the program follows.

Program 6-3. Sprite Parade

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
5 GOSUB 300                                          :rem 72
10 V=53248:S=200:PRINT"{CLR}{WHT}":POKE 53281,0:Y=
   255                                               :rem 45
20 PRINT"{4 DOWN}{WHT}{7 SPACES}*** SPRITES ON PAR
   ADE ***"                                          :rem 236
30 PRINT"{10 DOWN}{63}{40 £}";                       :rem 242
40 PRINT"++++++++++++++++++++++++++++++++++++++++
   ;                                                 :rem 168
50 PRINT"{40 +}"                                     :rem 38
60 POKE V,160:POKE V+1,Y:POKE V+21,1                 :rem 248
70 POKE V+29,1:POKE V+23,1                           :rem 139
80 POKE V+39,10:POKE V+37,3:POKE V+38,7              :rem 124
90 POKE 2040,S                                       :rem 221
100 R=INT(RND(1)*15)+1                               :rem 174
110 Y=Y-2:IF Y<O THEN S=S+1:Y=255:POKE V+39,R:POKE
    V+27,0                                           :rem 234
120 IF S=209 THEN S=200                              :rem 158
130 IF S=206 THEN POKE V+28,1                        :rem 241
140 IF Y<120 THEN POKE V+27,1                        :rem 241
150 IF S=207 THEN POKE V+28,0                        :rem 243
160 POKE V+1,Y:GOTO 90                               :rem 217
300 FOR X=0 TO 8                                     :rem 26
310 FOR S= 0 TO 63:READ A:POKE 12800+(64*X)+S,A:NE
    XT:NEXT X                                        :rem 223
320 RETURN                                           :rem 117
999 REM SPRITE # 200 SPACE FIGHTER                   :rem 144
1000 DATA0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,128,0,1
     12,0,0,120,0,0                                  :rem 237
1010 DATA 124,0,0,126,0,0,127,0,0,255,255,192,192,
     0,240,255,255,255,255                           :rem 132
1020 DATA 248,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
     0,0                                             :rem 171
1049 REM SPRITE #201 ALIEN LANDER                    :rem 94
1050 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,16,0,0,16,0,0,
     16,0,0,16,0,0                                   :rem 140
1060 DATA 56,0,0,124,0,0,170,0,1,85,0,1,255,0,1,25
     5,0,0,170,0,1,41,0                              :rem 185
1070 DATA 2,40,128,0,0,0,0,0,0,0,0,0,0,0,0,0
                                                     :rem 23
1099 REM SPRITE # 202 ROCKET SHIP                    :rem 65
1100 DATA 0,16,0,0,16,0,0,16,0,0,40,0                :rem 217
1110 DATA 0,40,0,0,40,0,0,68,0,0,68,0                :rem 229
```

```
1120 DATA 0,68,0,0,68,0,0,68,0,0,68,0      :rem 250
1130 DATA 0,68,0,0,68,0,0,198,0,1,69,0,2,68,128,4,
     68,64,8,68,32,31                      :rem 159
1140 DATA 255,240,0,56,0,0                 :rem 233
1149 REM SPRITE # 203 WOODEN BLOCK         :rem 121
1150 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,255,248,1,126,
     24,2,130,40,5,252                     :rem 123
1160 DATA 72,10,4,168,23,249,72,63,254,136,32,130,
     168,33,66,208,34,34                   :rem 72
1170 DATA 160,39,242,64,40,10,128,40,11,0,63,254,0
     ,0,0,0,0,0,0,0,0,0                    :rem 26
1199 REM SPRITE # 204 SPACE DESTROYER      :rem 117
1200 DATA 0,126,0,3,195,192,7,255,224,1    :rem 108
1210 DATA 195,128,0,126,0,128,60,1,128,60,1,128,60
     ,1,129,255,129,255,129                :rem 203
1220 DATA 255,185,255,157,143,195,241,128,126,1,12
     8,36,1,0,24,0,0,24                     :rem 2
1230 DATA 0,8,24,16,8,24,16,8,24,16,15,255,240,9,2
     31,144,0                              :rem 15
1249 REM SPRITE # 205 USS ENTERPRISE       :rem 65
1250 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
     0,255,248,0,64                        :rem 198
1260 DATA 16,0,127,224,0,12,0,0,6,3,248,3,15,254,1
     ,131,0,0,198,0,0,108                  :rem 55
1270 DATA 0,15,255,0,31,254,0,0,0,0,0,0,0,0,0,0,0
                                           :rem 21
1299 REM SPRITE # 206 SAUCER ALIEN         :rem 119
1300 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,20,0,0,1
     70,0,2,170,128                        :rem 193
1310 DATA 9,150,96,170,170,170,10,85,160,2,170,128
     ,0,170,0,2,0,128,8,0                  :rem 80
1320 DATA 32,42,0,168,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
     0,0                                   :rem 190
1349 REM SPRITE # 207 ALIEN VISAGE         :rem 112
1350 DATA 0,8,0,0,28,0,0,127,0,1,255,192,67,255,22
     5,102,127,51,119,62                    :rem 36
1360 DATA 119,127,28,127,127,8,127,127,247,255,127
     ,247,255,127,247,255,127               :rem 92
1370 DATA 255,255,111,0,123,70,255,177,7,255,240,3
     ,193,224,1,255,192,0                  :rem 110
1380 DATA 127,0,0,28,0,0,0,0,0             :rem 154
1399 REM SPRITE # 208 SPACE FORTRESS        :rem 50
1400 DATA 0,0,0,0,16,0,0,56,0,0,16,0,15,255,224,0,
     40,0,1,255,0,7,255                    :rem 190
1410 DATA 192,0,130,0,63,255,248,31,255,240,1,1,0,
     3,255,128,1,255,0                     :rem 186
1420 DATA 0,68,0,0,254,0,0,56,0,0,108,0,0,124,0,0,
     40,0,0,16,0,0                         :rem 188
```

How the Program Works:

Line	Function
5	Send the program to the routine at line 300 which READs and POKEs the DATA numbers for the nine sprites.
10	Set the variables V, S, and Y and the screen background color to black.
20	Print the screen title in white.
30-50	Print three lines of graphics characters for the sprites to pass in front of as they move on the screen.
60	POKE in the X and Y coordinate position of the sprite and turn it on.
70	Expand the sprite both horizontally and vertically. Omitting this line leaves the sprites at normal size.
80	Set the color for the sprites.
90	Begin the main loop of the program by POKEing the sprite pointer to the first sprite, 200. This simply tells the 64 where to get its DATA for each sprite picture.
100	Set the random number for sprite color. Adding 1 to the INT result eliminates the possibility of generating a 0, which would create a black sprite. Since the screen background is also black, this would make the sprite invisible.
110	Move the sprite by subtracting 2 from its Y coordinate position each time this line is executed. When Y = 0, the sprite is off the viewing area, and 1 is added to S, changing the sprite picture. Y is reset to 255, starting the next sprite at the bottom of the screen. The V + 27 register is also turned on, then off, to force the sprite to first pass *in front of*, then *behind* the screen characters. (The next chapter explains this more fully.)
120	Reset the variable S once all sprites have been displayed. This repeats the display.
130	Turn the multicolor mode on for sprite 206.

150	Turn off the multicolor mode once sprite 206 has disappeared.
160	POKE in the new sprite and repeat the main loop.
1000-1420	DATA statements for the nine sprites.

Sprites, Sprites, Sprites

The Commodore 64's sprite-handling ability is an excellent tool for the game designer and programmer. You can design and create your own sprites, either in single colors or in a multicolor mode. The sprites can be displayed in normal size, or expanded. They can even be switched off and on, appearing and disappearing, or their colors can change during a game.

Creating sprites is not the only thing the 64 does, however. It also moves them. The next chapter shows you how this is done.

7
Moving Sprites

Moving Sprites

Character movement on the 64, as detailed in Chapter 5, is actually two separate problems: *movement* and *animation*. Movement is getting the character from one place to another on the screen. Animation is making that player seem natural in its movements.

Moving sprites presents much the same problem to a game designer. Creating a character that seems to move naturally is often a challenge, especially to the beginning programmer. Fortunately, moving a sprite is much simpler than moving a character. This is because the 64 is designed to do much of the programmer's work. To move a sprite, all you have to do is set the X and Y positions, and then instruct the computer to provide a series of new locations. The sprite, unlike a character, does not have to be constantly erased and rePOKEd into a new location.

Sprite Priorities

Sprites have a peculiarity not shared by characters. As they move across the screen, sprites have the ability to cross each other's paths. Characters that attempt to do this are erased, and must be redrawn. Sprites can even move in front of, or behind, other objects on the screen, giving a simulated three-dimensional effect. This is especially useful in games.

Sprites cross other figures on the screen according to a procedure that doesn't change. This is called sprite priority, and should be mentioned before you actually begin working with sprite movements.

Lower-numbered sprites will always pass in *front* of higher-numbered sprites. That is, sprite 0 passes in front of all other sprites; sprite 6 passes only in front of sprite 7; and so on. You can create a window effect by making a sprite with a gap in it, then causing another sprite to pass behind it. This makes for an impressive three-dimensional effect.

Sprites can also be set to pass in front of, or behind, other characters or background displays. To do this the register V+27 is POKEd. Each of the eight sprites must have its own bit set, just as when sprites are enabled, or turned on. If a particular sprite's bit is *on*, that sprite will pass or appear *behind* background or screen characters. If the bit is *off* (as it normally is for all sprites), then it will pass *in front of* any screen characters.

To give the screen and background priority over a particular sprite, POKE in the appropriate value. Refer to the Sprite POKE Table in Chapter 6 for these values. For instance, setting the screen priority over sprite 0, you would type in:

```
POKE V+27, 1
```

Just as when you enable any of the one-byte registers for sprites, to turn on more than one, you must add the sprites' values together. For example:

```
POKE V+27, 129
```

will set the background and screen priority over sprites 0 and 7.

Computer-Controlled Sprites

Many times you'll want computer-controlled sprites moving on the screen. That's what this chapter will explore; player-controlled sprites, either through a joystick or the keyboard, will be dealt with in the next chapter.

Sprites can be used for the same purposes as standard or custom characters. They can serve as targets, as opposing figures, as objects to avoid, or simply as projectiles. Because their movements are so easy to program, you'll probably find yourself using them in place of custom characters in many instances.

Moving a sprite becomes a series of commands which change the location in as small increments as possible. The larger the distance between screen positions, the jerkier the movement will seem, so you'll want to change the position one pixel at a time if possible.

We'll consider three kinds of sprite movements that are directed by the computer rather than the player. They are: simple one-direction movements, random movements, and intelligent movements. Each type of sprite movement has its own application in a game, so we'll consider them separately.

Falling Objects

One of the most popular videogames is *Space Invaders*. The opponents are not intelligent, nor do they move in a random pattern. Over and over again, they descend toward the player. They are actually little more than falling objects that also move across the screen. *Astrosmash*, an arcade-style game for the Mattel Intellivision system, shows this type of movement even

better. In that game, asteroids continually fall on your player-controlled character. The angle they approach your character varies from time to time, but they are basically falling rocks.

Creating something similar on the 64 is quite simple using sprites. We'll begin with this sort of sprite movement because it is the easiest to program, and will set a base for the later, more complicated movements of sprites.

You've created your sprite (it doesn't have to look like a falling rock for you to see how to move it), and you're ready to see it smoothly move about the screen. You can use the DATA information for your own sprite in place of the statements used in the program below. All you have to do is insert your DATA numbers in lines 200-260.

Program 7-1. Falling Blocks

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 PRINT"{CLR}":POKE 53281,0:V=53248:X=20:Y=20
                                              :rem 195
20 FOR X=0 TO 62:READ A:POKE 12288+X,A:NEXT
                                              :rem 135
30 POKE 2040,192                               :rem 32
35 POKE V+23,1:POKE V+29,1                    :rem 140
40 POKE V+21,1                                :rem 212
50 POKE V+39,10                                :rem 14
60 POKE V+0,X                                 :rem 202
70 POKE V+1,Y                                 :rem 205
80 Y=Y+1:IF Y>255 THEN Y=20                   :rem 214
100 GOTO 60                                    :rem 48
200 DATA 255,255,255,255,255,255,255,255,255
                                              :rem 136
210 DATA 255,255,255,255,255,255,255,255,255
                                              :rem 137
220 DATA 255,255,255,255,255,255,255,255,255
                                              :rem 138
230 DATA 255,255,255,255,255,255,255,255,255
                                              :rem 139
240 DATA 255,255,255,255,255,255,255,255,255
                                              :rem 140
250 DATA 255,255,255,255,255,255,255,255,255
                                              :rem 141
260 DATA 255,255,255,255,255,255,255,255,255
                                              :rem 142
```

Much of this program should be familiar to you if you've read Chapter 6, for it uses the 0 sprite. The screen color is changed

to black, variables are set, and the sprite DATA is read and POKEd into the proper locations. The pointer is set in line 30, line 40 turns the sprite on, and line 50 sets its color to orange.

Lines 60 and 70 POKE in the sprite's X and Y starting positions on the screen. Line 80 is really the only line you haven't yet used. All it does is change the sprite's Y position by one pixel each time through the loop. When Y = 255, the lowest position on the screen, it is reset to 20, the Y coordinate starting place, and the process begins all over again.

After typing in and running this program, notice how smoothly the sprite moves down the screen. This is one of the advantages of using sprites, for they move much more smoothly than characters.

If you'd wanted the sprite to make only one pass down the screen, you could have changed line 80 to:

```
80 Y=Y+1:IF Y>255 THEN Y=255
```

This would cause the sprite to reach the bottom and remain there. Remember that you'll see an ILLEGAL QUANTITY error on the screen if you try to POKE a number greater than 255 into any memory location.

Moving the sprite in other directions is just as easy. Using the loop technique in lines 60-100 in the previous program, you can make a sprite go up, down, to either side, or even diagonally across the screen. Look at the program below for a moment, then type it in to see a sprite move in several directions. Again, you can insert your own sprite's DATA statements in lines 200-260 as a replacement.

Program 7-2. Diagonal Movement

```
10 PRINT"{CLR}":POKE 53281,0:V=53248:X=50:Y=30
                                              :rem 199
20 FOR X=0 TO 62:READ A:POKE 12288+X,A:NEXT
                                              :rem 135
30 POKE 2040,192                               :rem 32
40 POKE V+21,1                                :rem 212
50 POKE V+39,10                                :rem 14
60 POKE V+0,X                                 :rem 202
70 POKE V+1,Y                                 :rem 205
80 Y=Y+1:IF Y=220 THEN 90                      :rem 62
85 GOTO 60                                     :rem 12
90 POKE V+0,X:POKE V+1,Y                      :rem 109
100 X=X+1:IF X=255 THEN 110                   :rem 149
105 GOTO 90                                    :rem 56
110 POKE V+0,X:POKE V+1,Y                     :rem 150
120 Y=Y-1:IF Y=50 THEN 130                    :rem 103
```

```
125 GOTO 110                                        :rem 99
130 POKE V+0,X:POKE V+1,Y                          :rem 152
140 X=X-1:IFX=50 THEN 150                          :rem 104
145 GOTO 130                                       :rem 103
150 POKE V+0,X:POKE V+1,Y                          :rem 154
160 X=X+1:Y=Y+1:IF X=255 THEN PRINT "{CLR}{3 DOWN}
    {WHT}ALL DONE!":END                             :rem 91
165 GOTO 150                                       :rem 107
200 DATA 0,0,0,0,0,0,0,0,0                         :rem 188
210 DATA 0,0,0,0,126,0,0,255,0                     :rem 146
220 DATA 1,255,128,3,255,192,3,255,192              :rem 76
230 DATA 3,213,192,3,255,192,3,255,192              :rem 74
240 DATA 1,255,128,0,255,0,0,66,0                   :rem 64
250 DATA 0,66,0,0,231,0,0,0,0                       :rem 99
260 DATA 0,0,0,0,0,0,0,0,0                         :rem 194
```

The sprite moves down, to the right, up the screen, left, and finally diagonally to the lower-right corner of the viewing area. Each of the five loops moves the sprite in one straight-line direction. The down movement you've already seen.

To move a sprite to the right, all you have to do is increase its X coordinate position one pixel at a time. The line below does that:

```
100 X=X+1:IF X=255 THEN 110
```

When the X position reaches its maximum, 255, the sprite begins the next loop in line 110.

To move up, you must decrease the sprite's Y coordinate position:

```
120 Y=Y-1:IF Y=50 THEN 130
```

Moving to the left is done with the line:

```
140 X=X-1:IF X=50 THEN 150
```

This decreases the X coordinate by one each time through the loop. The sprite will end up in its starting location.

To move diagonally, you only need to combine the increases of both the X and Y coordinates. Line 160 does this:

```
160 X=X+1:Y=Y+1
```

When X = 255, the sprite will have moved off the viewing area and a message will display. Something similar can be used to move the sprite in the other diagonal directions. For instance, to move up and to the right, you could type:

X=X+1: Y=Y−1

Moving down and to the left-hand corner can be accomplished with:

X=X−1: Y=Y+1

Up and to the left would look like:

X=X−1: Y=Y−1

You have eight possible movements with the sprite so far. Since it is moving one pixel at a time, the movement is smooth. Sprites erase themselves, so there is no need to do that in the program, unlike when you're using custom characters. However, sprites can move in more than eight directions, not just in the eight lines that a character can move. To show this, change line 160 in the original program to:

```
160 X=X+1:Y=Y+.5:IF X=255 THEN PRINT "{CLR}
    {3 DOWN}{WHT}ALL DONE!":END              :rem 141
```

If you run this version of the program, you'll see the sprite moving to the left and down at a different angle than before. You can change the angle by simply altering the .5 to values such as .3 or .25.

You can even make the sprite head toward the right of the original path by changing line 160 to:

```
160 X=X+1:Y=Y+.25:IF X>254 OR Y>254 THEN PRINT "
    {CLR}{3 DOWN}{WHT}ALL DONE!":END          :rem 146
```

The additional changes in the line are necessary, for if they are omitted, you'll see an ILLEGAL QUANTITY error message. What happens is that the Y coordinate position becomes greater than 255.

The Invisible Seam on the 64

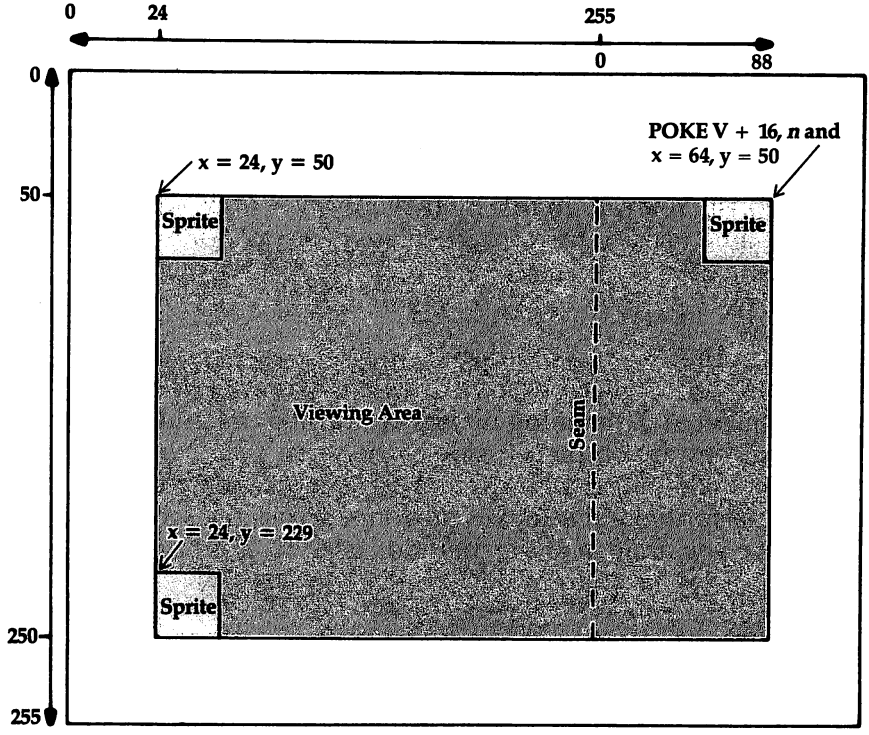
As you ran the sample programs moving the sprite from side to side, or around the edge of the screen, you probably noticed that it did not move all the way to the right-hand side of the viewing area. This is because of the *seam* that separates two different sections of the 64's screen.

You didn't have to worry about this seam when you created and moved custom characters. Characters will move only within the viewing area. Sprites are not bound by this rule, however. They are not confined to the 1000 screen locations, as are characters, but have a playing area greater than what the monitor screen displays at one time. Although this is in some ways an advantage when using sprites in a game, it is

also a bit more difficult to program.

Take a look at Figure 7-1. This shows the viewing area of your television or monitor screen, along with the X and Y coordinates where your sprites can be located.

Figure 7-1. Screen Display



Not every coordinate you can place your sprite on will appear in the viewing area. If you located a sprite at X=0: Y=0, it would not appear on the screen. The upper-left corner of the viewing area is X=24: Y=50. This is the first location which will display all of your unexpanded sprite. The bottom-left corner has the same problem. To show a full sprite in its entirety, you could only use X=24: Y=229. If the Y position is greater than 229, part or all of the sprite may be cut off.

The horizontal position of a sprite is even more complex. Twice as wide as it is high, the sprite movement area extends 512 positions horizontally. You'll actually use only a portion of

these locations, the ones shown in the viewing area, but that number is still higher than 255. You'll remember that 255 is the highest number you can POKE into a location, and since you set and move sprites by doing just that, getting to the far right-hand side of the screen is complicated.

POKEing to Cross the Seam

The 64 creates, in effect, two screens. One is accessed normally, simply by POKEing in the X and Y positions. The other, near the right-hand side, is accessed only when an *additional* register is turned on. This register's address is V+16, (or location 53264), and must have one bit turned on for each of the eight sprites that wish to cross the seam which divides the 64's screen. You can do this by POKEing V+16, *x*, where *x* is the value used to turn a particular sprite on. For example, POKEing

V+16,1:V+0,0

allows sprite 0 to cross the seam. In effect, the register V+16 tells the sprite which of the two screens it is on. (Normally, V+16 is set at 0, meaning no bits are turned on, and all sprites assume they are on the left side of the seam.)

The following program shows this movement. There is only one sprite, and it moves just once across the screen, but it will give you an idea of the methods used to cross the seam.

Program 7-3. Moving Across the Seam

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 PRINT"{CLR}":POKE 53281,0:V=53248:X=20:Y=120
                                              :rem 244
20 FOR X=0 TO 62:READ A:POKE 12288+X,A:NEXT
                                              :rem 135
30 POKE 2040,192                               :rem 32
40 POKE V+21,1                                :rem 212
50 POKE V+39,10                                :rem 14
60 POKE V+0,X                                 :rem 202
70 POKE V+1,Y                                 :rem 205
80 X=X+1:IF X>255 THEN POKE V+16,1:X=1         :rem 79
90 IF (PEEK(V+16) AND 1)=1 AND X=91 THEN PRINT"MAD
   E IT ACROSS!":END                          :rem 156
100 GOTO 60                                    :rem 48
200 DATA 0,0,0,0,0,0,0,0,0                    :rem 188
210 DATA 0,0,0,0,126,0,0,255,0                :rem 146
220 DATA 1,255,128,3,255,192,3,255,192         :rem 76
```

```
230 DATA 3,213,192,3,255,192,3,255,192      :rem 74
240 DATA 1,255,128,0,255,0,0,66,0           :rem 64
250 DATA 0,66,0,0,231,0,0,0,0               :rem 99
260 DATA 0,0,0,0,0,0,0,0,0                 :rem 194
```

The only lines that may look unfamiliar are lines 80 and 90. Line 80 moves the sprite across the screen. When the X coordinate equals 255, in other words, when the seam is encountered, the V + 16 register is turned on for the 0 sprite. X is reset to 0. Line 90 prints a message only when the sprite reaches the far right-hand side (X = 91) and only when the V + 16 register is turned on (PEEK(V + 16) AND 1) = 1. If this were omitted, then the message would be printed the *first* time X = 91, which would have been on the left side of the seam.

Remember that once the seam is crossed, the X positions are renumbered, beginning with 0 again, and run to 88, the last X location which shows in the viewing area. If the sprite is moving to the left, and crosses the seam, then the V + 16 register must be turned *off* by using:

POKE V + 16,PEEK(V + 16) AND 254:POKE V + 0,255

The sprite moves over the seam and is placed at the far right-hand side of that part of the screen. Although you could use the statement POKE V + 16,0:POKE V + 0,255 instead, it's good programming practice to use the PEEK method, for there will be times you'll have more than one sprite on the screen.

To move the sprite back to the left, you need to add another loop, much like you did when you moved the sprite in only the left-hand side of the screen. To make the sprite move back and forth across the screen, you would add these lines:

Program 7-4. Back and Forth

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
80 X=X+1:IF X>255 THEN POKE V+16,PEEK(V+16)OR 1:X=
   1                                          :rem 78
90 IF (PEEK(V+16) AND 1)=1 AND X=91 THEN GOTO 110
                                             :rem 229
110 POKE V+0,X:POKE V+1,Y                    :rem 150
120 X=X-1:IF X=0 THEN POKE V+16,PEEK(V+16)AND 254:
    X=255                                    :rem 21
125 IF (PEEK(V+16)AND 1)=0 AND X=1 THEN GOTO 60
                                             :rem 174
130 GOTO 110                                 :rem 95
```

Lines 110 and 120 POKE in the sprite's location and move it to the left (X=X−1). When the seam is reached (X=0), the V+16 register is turned off and X is reset to 255. Line 130 tests to see if the sprite has reached the X=1 location on the left side of the seam. If it has, the process repeats as the program moves back to line 60.

More Than One Sprite

Turning one sprite on or off is relatively simple. Once you have several sprites on the screen, all of them moving from one side of the seam to the other, it can get more difficult.

To turn on more than one sprite, for instance, you have to add together the sprite values when you POKE V+16,*x*. If both sprites 1 and 3 were crossing the seam, for example, you would POKE V+16,10. But if you wanted to allow only one of the sprites to cross the seam, you would use a different type of POKE statement.

If you had a number of sprites on the screen, and wanted only sprite 0 to move from the right-hand section to the left-hand section, you couldn't just POKE V+16,0. This would force all the sprites to move to the left-hand side of the seam. Instead, you could use:

```
POKE V+16,PEEK(V+16)AND(255-1)
```

Change the 1 in (255−1) to 2, 4, 8, 16, 32, 64, or 128 for the other seven sprites.

This clears only sprite 0's bit, so it will be the only one to move to the left of the seam. You can use the logical AND to turn off just one sprite, leaving the others on, for any of the other sprite functions where all the sprites share a single register.

You would do something similar to move one sprite to the right of the seam. In order to keep other sprites which may already be in that section of the screen in place, you could use the logical OR.

```
POKE V+16,PEEK(V+16)OR 1
```

OR 1 could be replaced by OR 2, 4, 8, 16, 32, 64, or 128, depending on the sprite.

This allows the 0 sprite to move to the right of the seam, but does not disturb any of the others. You'll use this method of moving sprites from one side of the seam to the other constantly. If you're unfamiliar with the concept of the logical

AND and OR, refer to Chapter 4, Custom Characters, where it is explained in detail.

More about the Seam

To move a sprite completely across the screen and have it reappear on the left-hand edge can also be complicated. As the sprite nears the right edge, you would normally POKE V+16,0 to allow it to return to the left-hand section, then POKE a new X coordinate to show it in its new location.

Unfortunately, if you POKE V + 16,0 while X=91 (just off the right side of the visible area), the sprite will appear for a moment in the X=91 location on the *left* side of the seam. If you reverse the order of the statement and first set X=0, then POKE V+16,0, the sprite will flash in the X=0 location on the *left* side of the seam.

To prevent this, you can turn the sprite *off* just before this process and then turn it back *on* after both the POKE V+16,0 and X=0 have been executed. The program below does this.

Program 7-5. Smooth Sprite Movement

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 PRINT"{CLR}":POKE 53281,0:V=53248:X=50:Y=120
                                              :rem 247
20 FOR X=0TO63:READ A:POKE 12288+X,A:NEXT :rem 136
30 POKE 2040,192                               :rem 32
40 POKE V+21,1                                :rem 212
50 POKE V+39,4                                :rem 225
60 POKE V+0,X                                 :rem 202

70 POKE V+1,Y                                :rem 205
80 X=X+2:IF X>255 THEN POKE V+16,PEEK(V+16) OR 1:X
   =1                                          :rem 79
90 IF (PEEK(V+16) AND1)=1 AND X=91 THEN 120
                                              :rem 173
100 GOTO 60                                    :rem 48
120 POKE V+21,0:X=0:POKE V+0,X:POKE V+16,PEEK(V+16
    ) AND 254:POKE V+21,1:GOTO60              :rem 107

200 DATA 0,0,0,0,0,0,7,255,224,15,255,240,25,36,15
    2,31,255,248,15,255                        :rem 27
210 DATA 240,7,255,224,0,24,0,32,189,4,49,255,140,
    43,255,212,36,60,36,40,24                  :rem 87
220 DATA 20,48,36,12,32,90,4,0,189,0,1,126,128,3,0
    ,192,0,0,0,0,0,0,36,40,24                  :rem 35
```

The sprite is set up in the way you've already used. Lines 10-70 draw the sprite and place it on the screen at location 50,120. Line 80 moves the sprite to the right, constantly checking for the seam. Notice that it moves faster than your previous sprites because X=X+2. You can change this to a larger number, but the movement will seem jerkier. Once the X position reaches 255, the V+16 register is turned on and X is reset.

Line 90 checks to see if the far right-hand side of the screen is reached. When it is (X=91), the sprite is turned *off* (V+21,0), X is reset, and the program moves to line 120.

Line 120 POKEs the X location in again, sets the V+16 register back to zero to indicate the sprite is on the left side of the seam, and then turns the sprite back *on*.

Note the use of the logical AND and OR in this program. It's a good idea to always use this form, in case there is more than one sprite on the screen at a time.

You can move your sprite almost any direction on the screen, even across the troublesome seam on the 64. The patterns you've used so far, however, are rather predictable. In a game, a player would quickly discover that pattern. It might not be much of a game. Fortunately, programming sprites to make random movements isn't that difficult.

The Random Factor

Computer-controlled sprites that move randomly make for more difficult opponents. They can move in different patterns each game, or even during each pass across the screen.

The random generator in the 64 allows you to place and move sprites in this less predictable way. Each time a sprite moves on the screen, for example, you could have it change its altitude (Y position) or its speed (X+*n*). You can randomly set the number of sprites on the screen, the angle at which they move, or when they attack the player-controlled character.

To create this random effect, you'll use the RND function in your program. To produce a random number *R* between 0 and 3, for instance, the form would be:

```
R=INT(RND(9)*4)
```

This is the same form you used to create random screen displays in Chapter 3. If you've forgotten how to create random numbers, refer to the section in that chapter.

You can create minimums and maximums using the RND

function, as well as fractions if you wish. But once you have a random number, what can you do with it?

If you want to vary the speed of a sprite each time the main loop is executed, you could add these lines to Program 7-3.

Program 7-6. Random Sprites

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
75 R=INT(RND(9)*4)+1                                    :rem 95
80 X=X+R:IF X>255 THEN POKE V+16,PEEK(V+16)OR 1:X=
   1                                                    :rem 111
90 IF (PEEK(V+16) AND 1)=1 AND X>91 THEN GOTO 110
                                                        :rem 230
110 POKE V+0,X:POKE V+1,Y                               :rem 150
115 R=INT(RND(9)*4)+1                                   :rem 138
120 X=X-R:IF X<0 THEN POKE V+16,PEEK(V+16)AND 254:
    X=255                                               :rem 53
125 IF (PEEK(V+16)AND 1)=0 AND X<4 THEN GOTO 60
                                                        :rem 176
130 GOTO 110                                            :rem 95
```

Lines 75 and 115 create the random numbers ranging from 1 to 4. X=X+R in line 80 and X=X−R in line 120 move the sprite across the screen at varying speeds. Notice that it was necessary to change the X=91 to X>91 in line 90 and the X=0 to X<0 in line 120. This prevents any ILLEGAL QUANTITY error messages. Line 125 includes the value X<4 for the same reason. You can't POKE in negative numbers, remember.

To show random movement in a different way, take a look at the program below. This is a variation of Program 7-3, but now it POKEs in the X position of the sprite randomly. The sprite appears in different X locations, ranging from 24 to 255, but stays on the same Y location "line." The delay loop is necessary to see the sprite's change of position. In a game, this would make the sprite a difficult target.

Program 7-7. Random Appearances

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 PRINT"{CLR}":POKE 53281,0:V=53248:X=20:Y=120
                                                        :rem 244
```

```
20 FOR X=0 TO 62:READ A:POKE 12288+X,A:NEXT
                                              :rem 135
30 POKE 2040,192                               :rem 32
40 POKE V+21,1                                :rem 212
50 POKE V+39,10                                :rem 14
55 X=INT(RND(9)*232)+24                       :rem 251
60 POKE V+0,X                                 :rem 202
70 POKE V+1,Y                                 :rem 205
80 FOR T=0 TO 500:NEXT T                       :rem 21
100 GOTO 55                                    :rem 52
200 DATA 0,0,0,0,0,0,0,0,0                    :rem 188
210 DATA 0,0,0,0,126,0,0,255,0                :rem 146
220 DATA 1,255,128,3,255,192,3,255,192         :rem 76
230 DATA 3,213,192,3,255,192,3,255,192         :rem 74
240 DATA 1,255,128,0,255,0,0,66,0              :rem 64
250 DATA 0,66,0,0,231,0,0,0,0                  :rem 99
260 DATA 0,0,0,0,0,0,0,0,0                    :rem 194
```

Line 55 creates a random number from 24 to 256, the X coordinate positions in the screen to the left of the seam. The delay in line 80 simply slows down the movement so you can see it working.

You can also move the sprite in random directions by using the RND function. The sprite will not move quickly in the following program, but it *will* move unpredictably. It's a matter of setting up several IF/THEN statements, each executed when a particular number is created by the RND function. The sprite creation and design are similar to the other sample programs you've already seen in this chapter. Notice lines 80-110, which contain the IF/THEN statements.

Program 7-8. Random Directions

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 PRINT"{CLR}":POKE 53281,0:V=53248:X=20:Y=120
                                              :rem 244
20 FOR X=0 TO 62:READ A:POKE 12288+X,A:NEXT
                                              :rem 135
30 POKE 2040,192                               :rem 32
40 POKE V+21,1                                :rem 212
50 POKE V+39,10                                :rem 14
60 POKE V+0,X                                 :rem 202
70 POKE V+1,Y                                 :rem 205
72 FOR T=0 TO 200:NEXT T                       :rem 19
75 R=INT(RND(9)*4)                              :rem 3
80 IF R=0 THEN X=X+5:GOTO 60                   :rem 11
```

```
90 IF R=1 THEN X=X-5:GOTO 60                    :rem 15
100 IF R=2 THEN Y=Y+5:GOTO 60                   :rem 56
110 IF R=3 THEN Y=Y-5:GOTO 60                   :rem 60
200 DATA 0,0,0,0,0,0,0,0,0                     :rem 188
210 DATA 0,0,0,0,126,0,0,255,0                 :rem 146
220 DATA 1,255,128,3,255,192,3,255,192          :rem 76
230 DATA 3,213,192,3,255,192,3,255,192          :rem 74
240 DATA 1,255,128,0,255,0,0,66,0               :rem 64
250 DATA 0,66,0,0,231,0,0,0,0                   :rem 99
260 DATA 0,0,0,0,0,0,0,0,0                     :rem 194
```

A number from 0 to 3 is created in line 75. Depending on the number generated, the sprite will move to the right, the left, down, or up five pixels. The delay loop in line 72 was added to slow down the sprite's motion. Remove line 72 to see an entirely different effect.

Of course, in a game, you would have to add lines that would test to see if the sprite had reached any of the four borders of the display area. (If you let this program run long enough, eventually you'll see an ILLEGAL QUANTITY error message.)

Randomness in a game can create difficult opponents for the player. Sometimes too difficult. Keep in mind that every player will be slower than the computer-controlled sprites. You can make the game *too* hard by making the characters *too* unpredictable. Experimenting with your own game design, you'll quickly find out that often the best use of any game design tool is in moderation. It's difficult to enjoy a game when it feels as if you have no control at all over its course. Most players want to feel it was skill, not luck, that allowed them to win.

Intelligent Sprites

Using simple and random sprite movements, you can create opponents that may be difficult, and, at times, impossible for a player to beat. Many times this would depend on the sheer number of computer-controlled figures. If there is a large number, the player is simply overwhelmed. Many arcade games use this technique. *Defender*, for example, eventually throws so many opponents at the player's ship that it's impossible to continue.

At times, however, you'll want something more than overwhelming numbers to challenge your game's players. If a

player discovers that outguessing the computer-controlled characters makes more of a difference than eliminating a seemingly inexhaustible supply, he or she will play the game longer, and with more interest.

Intelligent sprites are one solution to this problem. Programming intelligent sprites is the most difficult way to make sprites move, but it is potentially the most rewarding. If you can program a sprite to maneuver, even defeat the player, you've accomplished a great deal.

Creating intelligent sprites is a matter of using a number of IF/THEN statements in the game program, outlining how the sprite reacts to the player-controlled character. When you're using BASIC, however, you'll quickly notice that a program slows down as more and more IF/THEN statements are added. Again, you must decide if the added intelligence of sprites is worth the slowing of a game's actions. Often you must compromise your ideal game, keeping such things as intelligent sprites to a minimum just to keep the game moving at an acceptable speed.

Homing In

One of the most popular and useful ways of programming intelligent sprites is to create computer-controlled figures that home in on the player's character. Most arcade-style games that use intelligent opponents use this technique. After a while in *Joust*, for instance, the enemy knights begin to move toward your knight quicker and in a more direct line. You can duplicate this by programming several sprites.

To do this, you program the sprite so that it moves on the screen, checks to see where the player's figure is, and then moves in that direction. No matter where the player's character is, the sprite moves toward it. The quickness or slowness with which the sprite moves determines how hard or easy it will be for the player to evade it.

Look at the program below for a moment, then type it in and run it. It's a short game that uses two sprites. You control one using the keyboard. The pattern of keys is: f5 = up; f7 = down; Commodore key = left; and SHIFT = right. The sprite with flashing windows is controlled by the computer, and will move to intercept your sprite.

Program Explanation:

Line	Explanation
10-80	Create the four sprite drawings. There are three pictures of the space station (to make it seem to flash) and one of the space saucer. The sprites are turned on; the mutlticolor mode is enabled; color is placed, and located onto the screen.
90	POKE in the new location of the player-controlled sprite.
100	Animate the space station sprite by establishing a pointer *P*, which can be incremented each time the main loop executes. This causes the small changes in the station's windows as it moves across the screen.
110-140	Read the keyboard to move the space saucer. The keycodes are read by PEEKing location 197 for the two function keys, and location 653 for the Commodore and SHIFT keys. Depending on which key is pressed, the saucer moves in one of four directions.
150-180	Check to see if the space saucer is near the seam on the screen. If it is, and moving from left to right, the appropriate POKEs are executed.
190	Force the sprite to remain on the screen. It cannot move off to the left. Similar lines could be added to keep it on the screen near the other three borders.
200	Check to see if the homing sprite is within 30 pixels of the player-controlled sprite. If it is, it stops moving. The ASB function returns the absolute value of a number, without any signs. All values returned are thus positive.
205	Use the SGN function to create a 1, 0, or −1. If the argument (X1 − A or Y1 − B) is positive, the result is 1; if zero, the result is also 0; and if negative, the result is −1. Two is added so that the value of Q or QQ is always positive. This value is then used in line 210 to execute the appropriate line. The value of F is also set in this line.
210	Depending on the result of the SGN function in line 205, the program executes one of the four lines from 230 to 260. If SGN(X1 − A) equals a positive number—in other words, if the player's sprite is to the right of the computer-controlled sprite—line 240

is executed, moving the computer's sprite to the right.

220 Check to make sure that the computer's sprite stays in the sprite viewing-area.

230-260 Move the station sprite toward the saucer.

270-275 Keep the station on the screen.

500-690 DATA statements for all four sprite drawings. The first three pictures are of the space station in its various stages of animation, while the fourth is the space saucer.

Program 7-9. Avoid the Ship

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 FOR X = 0 TO 62: READ A: POKE 12288 + X, A: NEX
   T                                          :rem 134
20 FOR X = 0 TO 62: READ A: POKE 12352 + X, A: NEX
   T                                          :rem 127
30 FOR X = 0 TO 62: READ A: POKE 12416 + X,A: NEXT
                                              :rem 129
40 FOR X = 0 TO 62: READ A: POKE 12736 + X, A: NEX
   T                                          :rem 135
50 PRINT"{CLR}": V = 53248: POKE 53281,0: P = 192:
    X1 = 50: Y1=50: A=168: B=148               :rem 63
60 POKE 2040, P: POKE 2041, 199               :rem 214
70 POKE V+21,255: POKE V+28,255: POKE V+39,1: FOR
   {SPACE}X=0 TO 6:POKE V+40 +X,3: NEXT       :rem 121
80 POKE V + 37, 2: POKE V+ 38, 15: POKE V +16, 0
                                              :rem 116
90 POKE 2040, P: POKE V, A: POKE V + 1, B: POKE V
   {SPACE}+ 2, X1: POKE V + 3, Y1              :rem 51
100 P = P + 1: IF P = 195 THEN P= 192:REM ANIMATIO
    N SEQUENCE                                :rem 46
110 IF PEEK (197) = 6 THEN Y1 = Y1 - 3: REM UP
                                              :rem 78
120 IF PEEK (653) = 1 THEN X1 = X1 + 3: REM RIGHT
                                              :rem 28
130 IF PEEK (197) = 3 THEN Y1 = Y1 + 3: REM DOWN
                                              :rem 222
140 IF PEEK (653) = 2 THEN X1 = X1 - 3: REM LEFT
                                              :rem 206
150 REM CHECK FOR SEAM FOR SAUCER             :rem 143
160 IF X1>252 AND PEEK(653)=1 THEN POKE V+16, PEEK
    (V+16) OR 2: POKE V+2,0: X1=0             :rem 38
170 IF X1 > 91 AND PEEK (V+16) AND 2 = 2 THEN X1 =
     91                                       :rem 92
```

```
180 IFX1<3ANDPEEK(V+16)AND2=2ANDPEEK(653)=2THENX1=
    255:POKEV+16,PEEK(V+16)AND253          :rem 244
190 IFX1<ØTHENX1=Ø                          :rem 67
200 IF(ABS(X1-A)+ABS(Y1-B))<3Ø THEN 9Ø      :rem 18
205 Q=SGN(X1-A)+2:QQ=SGN(Y1-B)+2:F=2        :rem 73
210 ON Q GOSUB 23Ø,23Ø,24Ø:ON QQ GOSUB 25Ø,25Ø,26Ø
                                            :rem 48
220 Q=A:GOSUB 27Ø:A=Q:Q=B:GOSUB 27Ø:B=Q:GOTO 9Ø
                                             :rem 2
230 A=A-F:RETURN                           :rem 225
240 A=A+F:RETURN                           :rem 224
250 B=B-F:RETURN                           :rem 229
260 B=B+F:RETURN                           :rem 228
270 IF Q<5Ø THEN Q=5Ø                       :rem 60
275 IF Q>229 THEN Q=229                    :rem 179
280 RETURN                                 :rem 122
500 REM CENTER #1 DATA                     :rem 168
510 DATA 252,Ø,63,24Ø,Ø,15,192,Ø,3,192,6Ø,3,192,25
    5,3,195,255,195,2Ø7,255,243            :rem 197
520 DATA 255,255,255,54,89,1Ø8,54,89,1Ø8,63,255,25
    2,57,1Ø1,156,57,1Ø1,156                 :rem 41
530 DATA 255,255,255                       :rem 222
540 DATA 2Ø7,255,243,195,255,195,192,255,3,192,6Ø,
    3,192,Ø,3,24Ø,Ø,15,252,Ø,63            :rem 2ØØ
550 REM CENTER #2 DATA                     :rem 174
560 DATA 252,Ø,63,24Ø,Ø,15,192,Ø,3,192,6Ø,3,192,25
    5,3,195,255,195,2Ø7,255,243            :rem 2Ø2
570 DATA 255,255,255,57,1Ø1,156,57,1Ø1,156,63,255,
    252,54,89,1Ø8,54,89,1Ø8                 :rem 46
580 DATA 255,255,255                       :rem 227
590 DATA 2Ø7,255,243,195,255,195,192,255,3,192,6Ø,
    3,192,Ø,3,24Ø,Ø,15,252,Ø,63            :rem 2Ø5
600 REM CENTER #3 DATA                     :rem 171
610 DATA 252,Ø,63,24Ø,Ø,15,192,Ø,3,192,6Ø,3,192,25
    5,3,195,255,195,2Ø7,255,243            :rem 198
620 DATA 255,255,255,53,15Ø,92,53,15Ø,92,63,255,25
    2,53,15Ø,92,53,15Ø,92                  :rem 18Ø
630 DATA 255,255,255                       :rem 223
640 DATA 2Ø7,255,243,195,255,195,192,255,3,192,6Ø,
    3,192,Ø,3,24Ø,Ø,15,252,Ø,63            :rem 2Ø1
650 REM SAUCER DATA                         :rem 92
660 DATA Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø,Ø
    ,Ø,6Ø,Ø,Ø,255,Ø,2,17Ø,128,6            :rem 43
670 DATA1Ø2,96,42,17Ø,168,255,255,255,42,17Ø,168
                                           :rem 127
680 DATA 1Ø,17Ø,16Ø,2,17Ø,128,Ø,195,Ø,Ø,Ø,Ø,Ø,Ø,Ø,
    Ø,Ø,Ø,Ø,Ø,Ø,Ø                          :rem 92
690 IF X1 < A THEN A = A - 2: IF A < Ø THEN A = Ø
                                           :rem 212
```

Lines 200-280 allow the station to track down your character. The X and Y coordinates are compared, and the station sprite is moved in the same direction as your figure. Notice that the station does not move to the right of the seam, although your saucer can. This may be something you'll want to use in your own game. It could be useful as a safe area for a player-controlled sprite.

By reversing this process, you can create a sprite that will *avoid* your character. Changing lines 200 – 210 will do this.

Program 7-10. Catch the Ship

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
200 IF(ABS(X1-A)+ABS(Y1-B))>50 THEN 90       :rem 22
205 Q=SGN(X1-A)+2:QQ=SGN(Y1-B)+2:F=4         :rem 75
210 ON Q GOSUB 240,240,230:ON QQ GOSUB 260,260,250
                                             :rem 50
```

Now the station will race away from your saucer, but only after it's gotten close to the station. Line 200 sets the distance at which the station begins moving. Line 210 reverses the order of the ON/GOSUB statements, so that the station moves *away*, not toward the saucer as it did in the earlier version of the program. The saucer will never quite catch the station, since the latter moves faster. Changing the value of F in line 205 will allow the saucer to eventually overtake the station.

Programming intelligent sprites is not impossible. Trade-offs must be made, however, when you decide to use this method of sprite movement. It does make the program more complicated and tends to slow down movement. In combination with other movements, it can add an arcade-like element to your game. Random movements, along with some intelligent sprites, can make for an exciting, challenging game.

Making It Easier

One of the most difficult things about sprite movement on the Commodore 64 is the seam. As you've noticed, moving back and forth across the seam requires programming to test for the sprite's location and which direction it's moving, and then more programming to reset its X coordinate. You can avoid this complication by using only the area on the left side of the seam.

Although this shrinks the screen display area, you can use the right-hand side for such things as time displays, scores, and other player information. You'll see many games on the 64 like this.

All sprite movement then takes place only on the left-hand side of the screen. Characters can still move across the entire display area, but it may look strange when some figures cross the seam, while others don't. For this reason, you may want to create a border near the seam by POKEing into screen memory. For an example of this, see Chapter 3, which includes a routine for setting up a border. With only a few changes, you can move the left border toward the seam, saving everything to the right of that line for player information.

Again, this is a compromise. To make things easier, you're sacrificing some of the 64's abilities. As game designer, you must decide what to keep and what to throw away. That's part of the fun of creating your own arcade-style game.

8 Controlling Movement

Controlling Movement

You've created character and sprite figures for your game and a playing field for them to live in. You've even made the computer control sprites as opponents. But so far your player has nothing to do.

Most games allow the player to assume a character, establishing some control over the game world. *Joust*, for example, lets the player become a knight on a strange steed. The player controls how this knight behaves. If the player wants the knight to fly to the left, it does. The game is won or lost depending on how the player uses this character.

In order to control a character, the player has to have a way to tell the computer what it is to do. In most arcade games, a joystick does this. If the player presses the joystick to the left, the player's character moves to the left. Other games use buttons, similar to the keys on the Commodore keyboard, to operate the character. Still others use a paddle, as one of the first videogames, *Pong*, did. As the game designer and programmer, you have the option of deciding how the player will control the game's action.

The Keyboard

The keyboard is very versatile. Since the Commodore 64 has 66 keys, you could conceivably give each a different function in a game. Of course, most games use only a few buttons or keys. But that doesn't mean the Commodore's keyboard should go to waste. Using the keyboard to move characters on the screen, the most common of all game controls, gives a great deal of precision to those movements. You can assign functions to keys in your game much like the arcade game *Defender*. That game uses buttons to control the four directions the player's character moves, as well as a button to fire, one for smart bombs, and another for jumping to hyperspace. If your game requires several different buttons or keys such as this, the keyboard may be your first choice.

There are trade-offs, however. Unlike a joystick controller, which allows the player to move a character in eight different directions, the keyboard is usually restricted to only four. This

is not a problem with the keyboard, but with the player, for it's very difficult to operate more than four keys at a time with any accuracy. For this reason, in most games that use a keyboard, you'll usually see only the up, down, left, and right directions used. That doesn't mean you can't move a character diagonally, however. You'll see how that's done shortly.

A keyboard, like any other game controller, is really just a series of switches. By changing which switches are on, and which are off, certain values are placed into the computer's memory locations. The 64 then looks to those memory addresses, and executes its tasks.

Reading the Keyboard

Normal BASIC INPUT and GET statements, although useful for other player operations, won't work too well in moving a figure. They're too slow. You'll want to get the player's instructions directly from the hardware—the keyboard in this case.

Every time a key is pressed on the 64, the operating system stores a code number at memory location 197. This is the key code, and it has no relation to either the ASCII or screen code. Location 653 is similar, except it holds the key code for the SHIFT, Commodore, and CTRL keys. Appendix H shows the key codes for every key on the 64 that can be read from those locations.

Control matrix. When your program uses the keyboard for movement control, you'll want separate keys for left, right, up, and down, and perhaps keys to perform other functions, the way the joystick button is used in games to make a figure jump or shoot or disappear.

But you can't use just any keys. You want players to put their fingers on the right keys and then forget about having to find them again. 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 too far apart, and only touch-typists can remember which is which without constantly looking. 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 a game.

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 at the bottom, left is at the left, and right is to the right:

```
          T                    @
F                 H      :           =
          V                    /
```

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 executing this line, 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=53):UD=(A=46)-(A=55)
```

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 (true (−1) minus false (0)). If A=53, however, LR will equal 1 (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, remember it this way:

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

Inside the parentheses you put the expression:

Keypress=Directionvalue

So the whole thing looks like this:

Keypress=PEEK(197)
Horizontal=(Keypress=Leftvalue)−(Keypress=Rightvalue)
Vertical=(Keypress=Upvalue)−(Keypress=Downvalue)

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

Allowing diagonals. 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 diagonal movement up and to the right. You can get only right *or* up.

In most games that's good enough. Diagonal movements don't matter much. But if your game *requires* this kind of character movement, there is a solution. Use SHIFT, Commodore, and/or CTRL for horizontal movement and any two other keys

for vertical movement. Since SHIFT, Commodore, and CTRL are read from location 653 and the rest of the keys are read from address 197, your program can read vertical and horizontal movement separately, and get diagonals when two keys are pressed at once.

You may run into another problem using this method. SHIFT, Commodore, and CTRL can be pressed in combination, and then a different value is stored in location 653. 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 the Commodore key is pressed, and 0 if both *or* neither is pressed.

```
LR=PEEK(653):LR=(LR=1)-(LR=2)
```

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

```
LR=PEEK(653) AND 3:LR=(LR>1)-(LR=1)
```

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

It's just as easy to read the values at location 197. Use *f5* and *f7* as up-and-down controls.

```
UD=PEEK(197):UD=(UD=6)-(UD=3)
```

Simple. After this program line is executed, UD will equal −1 if the f5 key is pressed and 1 if the f7 key is pressed.

Since UD and LR are read from separate locations, a player-controlled character can be moved diagonally, not just horizontally or vertically alone.

Horizontal and Vertical Movement

Except for erasing the figure in its old location, you've already done everything that's involved in horizontal and vertical movements. To move horizontally, you change the screen locations by adding or subtracting 1. To move vertically, you add or subtract 40. Subtract to move up or left; add to move down or right.

You can use the 1 or −1 from the key-reading routines directly for horizontal movement. But for vertical movement, you'll have to change the lines to reflect the fact that 40 must be added or subtracted. Here's what it would look like in the diagonal movement routine.

```
UD=PEEK(197):UD=40*((UD=6)-(UD=3))
```

If f5 (the up key) is pressed, the value will be −40 (40* −1); if f7 is pressed, it will be 40 (40*1); and if an invalid key is pressed, the value will be 0 (40*0).

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

Both methods work the same way. First, the program checks to see if the player wants to move the figure. Then it calculates the new address where the figure will be POKEd. If the address is beyond the screen, then either the characters don't move at all, or the program changes the address so that the figures appear on the opposite side. Only then do you erase the figures at the old location and POKE them in at the new.

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

Here's how the program works:

Line	Function
10	Set the screen background color, clear the screen, and disable the character set shift.
15	Establish variable values: SC = starting address of screen memory; OL = old location (starting at 1); FG = figure value—the screen code for the character to be moved; CM = starting location of color memory; and CL = character color (0, for black). Then POKE FG into SC + OL and CL into CM + OL to make the figure appear on the screen in the starting position.
20	Start the main 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 (LR and UD) are 0, go back and look for a new key code value.
30	Set NL to the new location for the figure (OL + LR + UD). Then GOSUB to one of the two edge-test routines, at 100 for the left/right test and 200 for the top/bottom edge test.
35	If NL has come back from the routines with the same

value as OL, go back to the start of the main loop without moving the figure.

40 Erase the figure at the old location (OL) and POKE it into the new location (NL). Also POKE in the color location CM+NL,CL. Then set OL to equal NL and go back to the start of the main loop.

100 Left/right edge test subroutine. EH is set to a number from 0 to 39, 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 isn't allowed. Likewise, if the figure will be at the rightmost position (EH=39) and got there by moving left (LH= −1), that means the figure crossed the left-hand edge of the screen. The movement is cancelled in either case by setting NL equal to OL.

120 RETURN to line 35.

200-210 Top/bottom edge-test subroutine. This test is easier. If the new location will be off the top of the screen (NL<0) or the bottom (NL>1000), the movement is not allowed.

Program 8-1. Character Movement with POKE

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 POKE 53281,1:PRINT"{CLR}":POKE 657,128  :rem 96
15 SC=PEEK(648):CM=55296:SC=SC*256:OL=1:CL=0:FG=90
   :POKE SC+OL,FG:POKE CM+OL,CL            :rem 171
20 A=PEEK(197):LR=(A=45)-(A=53):UD=40*((A=46)-(A=5
   5))                                      :rem 54
25 IF LR=0 AND UD=0 THEN 20                :rem 107
30 NL=OL+LR+UD:ON-(LR<>0)-2*(UD<>0)GOSUB 100,200
                                           :rem 235
35 IF NL=OL THEN 20                        :rem 250
40 POKE SC+OL,32:POKE SC+NL,FG:POKE CM+NL,CL:OL=NL
   :GOTO 20                                :rem 247
100 EH=INT(NL/40):EH=NL-40*EH              :rem 170
110 IF (EH=0 AND LR=1) OR (EH=39 AND LR=-1) THEN N
    L=OL                                    :rem 32
120 RETURN                                 :rem 115
```

```
200 IF NL>1000 OR NL<0 THEN NL=OL                    :rem 2
210 RETURN                                          :rem 115
```

Moving sprites. Controlling sprite movement with the keyboard is even easier. Using the matrix for diagonal movement (f5 = up, f7 = down, Commodore = left, SHIFT = right), you can move a sprite about the screen. The following program uses only the part of the screen to the *left* of the seam and the sprite remains on the screen at all times. This simplifies the programming.

This is how the program functions:

Line	Function
10	GOSUB to subroutine at 300, which sets up the sprite.
20	POKE in the X and Y coordinate positions of the 0 sprite.
30-60	PEEK to check which key was pressed, then move the sprite accordingly. Notice that the sprite moves faster since it uses increments of three pixels each time it changes position. If the sprite reaches a border (either an actual border or the seam), it stops and waits for another key to be pressed.
70	Send the program back to line 20, where the new X and Y coordinate positions are POKEd in.
300	Set variables for V, X and Y. Clear the screen and set background color to white.
310	READ the sprite DATA and place it in memory starting at 12288.
320	Enable sprite, set sprite pointer, and set sprite color to black.
500-520	Sprite DATA numbers.

Program 8-2. Keyboard Sprite Movement

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 GOSUB 300                                        :rem 116
20 POKE V+0,X:POKE V+1,Y                            :rem 102
30 IF PEEK(653)=1 THEN X=X+3:IF X>255 THEN X=255
                                                     :rem 73
40 IF PEEK(653)=2 THEN X=X-3:IF X<24 THEN X=24
                                                    :rem 223
```

```
50 IF PEEK(197)=3 THEN Y=Y+3:IF Y>229 THEN Y=229
                                                  :rem 86
60 IF PEEK(197)=6 THEN Y=Y-3:IF Y<50 THEN Y=50
                                                  :rem 234
70 GOTO 20                                         :rem 2
300 V=53248:PRINT"{CLR}":POKE 53281,1:X=160:Y=100
                                                  :rem 90
310 FOR I=0 TO 63:READ A:POKE 12288+I,A:NEXT
                                                  :rem 156
320 POKE V+21,1:POKE 2040,192:POKE V+39,0:RETURN
                                                  :rem 200
500 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,16,0,0,16,0,0,1
    6,0,0,16,0,0                                   :rem 91
510 DATA 56,0,0,124,0,0,170,0,1,85,0,1,255,0,1,255
    ,0,0,170,0,1,41,0                             :rem 136
520 DATA 2,40,128,0,0,0,0,0,0,0,0,0,0,0,0,0
                                                  :rem 230
```

Wrapping around. With this next program, we'll change several things as we move a character. First, instead of POKEing the figure on the screen, we'll PRINT it there. Second, instead of reading the control matrix, we'll read the f5, f7, Commodore, and SHIFT matrix, as with the sprite movement program.

Here's how the program operates:

Line	Function
5	Set up the string array V$(*n*), in which each value of V$ PRINTs HOME and the same number of CURSOR-DOWN characters as the subscript. In other words, V$(10) PRINTs HOME and ten CURSOR-DOWN characters; V$(2) PRINTs two HOME and two CURSOR-DOWN characters.
10	The same as line 10 in Program 8-1, Character POKE Movement.
15	Set the string FG$ to the character that will become the figure PRINTed on the screen. Set NV to the row and NH to the column where the figure first appears. Skip the next few lines to execute the PRINT routine at line 40.
20	The beginning of the main loop. Set *A* to the value of the key pressed; set *B* to the value from SHIFT and Commodore.
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, set *H* to 0; if neither f5 nor f7 is pressed, 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, 23, and *V* calls for a movement down (+1), subtract 23; if OV is at 0 and *V* calls for a movement up (−1), add 23.

35 Set NH (New Horizontal position) to OH (Old Horizontal position) plus *H* (Horizontal movement). If OH is set at the last valid column, 39, and *H* calls for a movement right (+1), subtract 38; if OH is set at 0 and *H* calls for a movement left (−1), add 38.

40 Skip down the number of rows from HOME to the row where the figure is located (PRINT V$(OV)), move right the number of columns to the figure's location (TAB(OH)), and erase the figure by printing a blank character (" "). Repeat the process, using only the new location values (NV and NH) so you can PRINT the figure (FG$) at the new position.

45 Set OH and OV to the current figure position (NH and NV), and go back to 20 to get the new instructions from the player.

Program 8-3. PRINT Character Movement

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
5 DIM V$(24):V$(Ø)="{HOME}":FOR I=1 TO 24:V$(I)=V$
  (I-1)+"{DOWN}":NEXT                          :rem 224
1Ø POKE 53281,1:PRINT"{CLR}{BLK}":POKE 657,128
                                               :rem 24Ø
15 FG$="Z":NV=12:NH=2Ø:GOTO 4Ø                  :rem 54
2Ø A=PEEK(197):B=PEEK(653)AND3                 :rem 2Ø2
25 H=(B>1)-(B=1):V=(A=6)-(A=3):IF V=Ø AND H=Ø THEN
    2Ø                                         :rem 194
3Ø NV=OV+V+24*((OV=23 AND V=1)-(OV=Ø AND V=-1))
                                               :rem 249
35 NH=OH+H+39*((OH=38 AND H=1)-(OH=Ø AND H=-1))
                                               :rem 168
4Ø PRINT V$(OV);TAB(OH);" ";V$(NV);TAB(NH);FG$
                                               :rem 1Ø5
45 OH=NH:OV=NV:GOTO 2Ø                         :rem 1Ø4
```

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 worth looking more closely at what is going on. Let's take a closer look at line 35:

```
35 NH=OH+H+39*((OH=38 AND H=1)-(OH=Ø AND H=-1))
                                        :rem 168
```

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 0, of course, in which case this entire line will do nothing but add 0 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 AND H=−1)). If the figure was already in column 0 (OH=0) and the movement called for is *left* (H=1), 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 at the rest of that expression.

If the second half is true, the first half (OH=38 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 39, with a result of 39, which is added to OH+H. Since we know that OH=0 and *H* is −12, that means that NH is set to 0−1+39, or 38. Now the PRINT instruction in line 40 will TAB to column 38, the last legal column.

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

Now you'll see that if the first condition is true, the second will be false, and we'll be setting NH to the value of 38+1−39, or 0, and the figure will jump from the right edge of the screen to the leftmost position.

And if neither condition is true, 39*(0−0) gives 0, and NH will simply equal OH+H—the normal movement will take place.

This keyboard routine, which includes the wraparound feature, is used in "Mission: Nova!," the game we're developing throughout this book. Take a look at Chapter 9 to see

how this movement routine was added to an actual game program.

Wrapping sprites. Making player-controlled sprites wrap around the screen is similar to the programming you did in Chapter 7, when your computer-controlled sprites disappeared at one edge and reappeared at the opposite border. Because of the problem with the seam, additional POKEs are required to check to see if the sprite has reached the seam, to cross the seam, and then to reset the X coordinate position. These conditions must be constantly checked when you use BASIC to move sprites, so the movement slows down somewhat. To see a sprite wrap around the screen in horizontal movement, you need to change only two lines of Program 8-2, Keyboard Sprite Movement, and add two new lines.

The changes work like this:

Line	Function
30	Move the sprite to the right, checking for the seam (X>255). When the seam is reached, POKE V+16,1 to enable the 0 sprite to cross. Reset X to 0.
35	Check to see if the sprite is at the far right-hand edge of the screen. If it is (X 65 AND PEEK (V + 16) = 1), the V+16 register is turned off (V+16,0) and its X coordinate is reset to 1.
40	Move the sprite to the left, checking for the seam (IF X<1 AND PEEK(V+16)=1). When the seam is reached, the V+16 register is turned off and X is reset to 255.
45	Check to see if the far left-hand edge of the screen is reached (IF X<1 AND PEEK(V+16)=0). When the edge is reached, turn on the V+16 register and reset X coordinate position to 65, the location at the far right-hand edge of the screen.

Program 8-4. Wrapping Around Sprites

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
30 IF PEEK(653)=1 THEN X=X+3:IF X>255 THEN POKE V+
   16,1:X=Ø                                  :rem 139
35 IF X>65 AND (PEEK(V+16)=1) THEN POKE V+16,Ø:X=1
                                             :rem 138
4Ø IF PEEK(653)=2 THEN X=X-3:IF X<1 AND PEEK(V+16)
   =1 THEN X=255:POKE V+16,Ø                 :rem 44
```

```
45 IF X<1 AND PEEK(V+16)=Ø THEN X=65:POKE V+16,1
                                          :rem 56
```

The sprite doesn't wrap around in the Y direction in this sample, although it would be relatively easy to do. Notice that as you add more lines to the movement loop, as we did in this program, the more the sprite slows down. In Program 8-4, for instance, the sprite moves much faster in the vertical direction than in the horizontal since the program does not have to check for the seam or the edges of the screen. You may want to restrict your sprites' movements to the screen area, perhaps even to just the portion to the left of the seam, to speed them up.

Reading the Joystick

Joysticks are the most commonly used game controllers today, although that doesn't mean you have to include them in your game. Oftentimes keyboard controllers will work fine. Sometimes, however, you'll want your players to use joysticks. They are more convenient in some ways, for they allow the player to move a character easily with only one hand, leaving the other free to press keys, for instance. Most joysticks also include a fire button, which you can program to serve almost any function you wish. Of course, many games use the fire button to shoot missiles, drop objects, or make the character disappear, but you could use it for starting the game, going to another level, or clearing the screen.

Joysticks on the 64 are read by PEEKing two locations. At location 56321, you read a joystick plugged into port #1. At location 56320, you read a joystick plugged into port #2. When you create your game, make sure the players are using the joystick port your program is reading by PEEKs.

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. In other words, a joystick is simply a series of switches, which are either *on* or *off*. Using PEEKs, you can read which are set to 0 and which to 1.

Here are the tests for each direction for a joystick plugged into port #1:

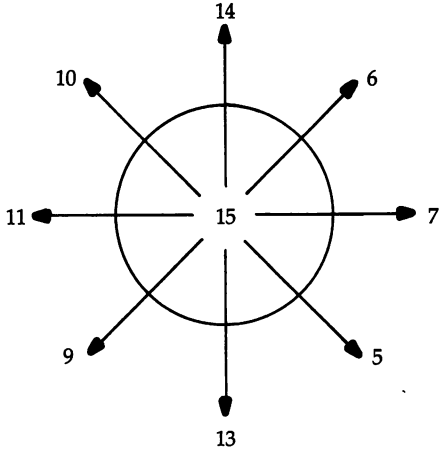
1. If the joystick is pushed left, PEEK(56321)AND15=11
2. If the joystick is pushed right, PEEK (56321)AND15=7
3. If the joystick is pushed up, PEEK(56321)AND15=14
4. If the joystick is pushed down, PEEK(56321)AND 15=13

5. If the joystick button is pushed, −((PEEK(56321)AND 16)=0)

If any of these expressions are true, the joystick is being pushed in that direction. Remember, too, that a joystick can be pushed in a diagonal direction—it is possible for both 2 and 3 to be true, or 1 and 4, and so on. And the joystick button can be pressed at the same time.

The values returned for each of the directions in which you can push the joystick are shown in Figure 8-1, Joystick Values. The value 15 is returned if the joystick is not pushed at all.

Figure 8-1. Joystick Values



Character moves. Using the logical AND is called *masking*. Just as you use masking tape to cover something you don't want painted, for instance, you can use the AND function to cover up bits you don't want exposed. It's then simple to read the joystick, just as you earlier read the keyboard, to move a character around the screen. Look at the following program for a minute:

Program 8-5. Joystick Character Movement

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
5 DIM V$(24):V$(Ø)="{HOME}":FOR I=1 TO 24:V$(I)=V$
  (I-1)+"{DOWN}":NEXT                          :rem 224
```

```
10 POKE 53281,1:PRINT"{CLR}{BLK}":POKE 657,128:FG$
   ="Q":NV=12:NH=20:GOTO 45                    :rem 246
15 JY=PEEK(56321)AND15                         :rem 246
20 H=(JY=11)-(JY=7)                             :rem 15
25 V=(JY=14)-(JY=13)                            :rem 82
30 IF H=0 AND V=0 THEN 15                      :rem 210
35 NV=OV+V+24*((OV=23 AND V=1)-(OV=0 AND V=-1))
                                               :rem 254
40 NH=OH+H+39*((OH=38 AND H=1)-(OH=0 AND H=-1))
                                               :rem 164
45 PRINT V$(OV);TAB(OH);" ";V$(NV);TAB(NH);FG$
                                               :rem 110
50 OH=NH:OV=NV:GOTO 15                         :rem 104
```

This is a variation of Program 8-2, which moved a character around the screen using keyboard controllers. Instead of reading the keyboard, however, this program reads the joystick plugged into port #1. Lines 15-25 do this. Notice that since PEEK(56321) is ANDed with 15, only horizontal and vertical movement is possible. Lines 20 and 25 set *H* and *V* to 1, −1, or 0, depending on which way the joystick is being pushed.

This program also uses a wraparound feature, which moves the character from the top to the bottom of the screen, and from side to side. Lines 35 and 40 do this.

To read the joystick fire button, more lines are needed in the program. It's a good idea to use a subroutine called only when the fire button is pressed. This shortens the main loop and makes the character move faster than if the fire button were read and checked each time through the loop. To see the fire button read, and a message displayed when it is pressed, add these lines to Program 8-5:

Program 8-6. Joystick Fire Routine

```
48 FB=-((PEEK(56321) AND 16)=0):IF FB=1 THEN GOSUB
   100                                          :rem 29
100 PRINT "FIRE!":FOR T=0 TO 100:NEXT T        :rem 140
110 PRINT "{CLR}":RETURN                        :rem 16
```

The fire button is read in line 48, and if it is pressed (FB = 1), the subroutine at line 100 is executed and a message is shown on the screen. If the fire button is not pressed, the PEEK in line 48 returns a 0, and the subroutine is not called.

To make the joystick move a character diagonally, more changes are needed in this sample program. Changing lines 20

and 25 in Program 8-5 allows the joystick to read the diagonal directions and then move the character. Here are the new lines.

```
20 H=((JY AND 4)=0)-((JY AND 8)=0)
25 V=((JY AND 1)=0)-((JY AND 2)=0)
```

Now you have a method to move characters on the screen with the same ease as when the Commodore, SHIFT, f5, and f7 keys were used before.

Moving sprites with the joystick. The joystick can also be used to move player-controlled sprites on the 64. The joystick is read instead of the keyboard with a PEEK statement, and the appropriate movement is then executed, depending on the direction the stick is pushed. Changing Program 8-2, Keyboard Sprite Movements, moves the sprite with a joystick instead of the keyboard.

Program 8-7. Joystick Sprite Moves

```
10 GOSUB 300                                          :rem 116
20 POKE V+0,X:POKE V+1,Y                              :rem 102
25 JY=PEEK(56321) AND 15                              :rem 247
30 IF JY=7 THEN X=X+3:IF X>255 THEN X=255 :rem 222
40 IF JY=11 THEN X=X-3:IF X<24 THEN X=24  :rem 158
50 IF JY=13 THEN Y=Y+3:IF Y>229 THEN Y=229 :rem 19
60 IF JY=14 THEN Y=Y-3:IF Y<50 THEN Y=50  :rem 165
70 GOTO 20                                             :rem 2
300 V=53248:PRINT"{CLR}":POKE 53281,1:X=160:Y=100
                                                       :rem 90
310 FOR I=0 TO 63:READ A:POKE 12288+I,A:NEXT
                                                       :rem 156
320 POKE V+21,1:POKE 2040,192:POKE V+39,0:RETURN
                                                       :rem 200
500 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,16,0,0,16,0,0,1
    6,0,0,16,0,0                                       :rem 91
510 DATA 56,0,0,124,0,0,170,0,1,85,0,1,255,0,1,255
    ,0,0,170,0,1,41,0                                  :rem 136
520 DATA 2,40,128,0,0,0,0,0,0,0,0,0,0,0,0,0,0
                                                       :rem 230
```

The variable JY is set in line 25 to equal PEEK(56321) AND 15. In the next four lines, the joystick is read, and the sprite is moved in the appropriate direction. If the joystick is pushed to the right, for example, the expression JY=7 is true and X=X+3. The sprite then moves to the right. Pushing the joystick up makes the expression JY=14 true in line 60, so Y=Y−3 and the sprite moves upward. Because only the

values received for horizontal and vertical movement are read, however, the sprite will not move diagonally. To do that, you'll need an entirely different kind of routine.

Diagonal joystick movement for sprites. Sprites move quite differently than characters, and creating diagonal movement is more difficult. You can't just increase the character's location by 40 and see it move down a line. Both the X and Y coordinates have to be changed to move a sprite diagonally, and this takes more programming. Program 8-8 shows how you can move a sprite diagonally with the joystick.

Here's how this program works:

Line	Function
10	Shift the program to line 300, where the sprite is established.
25	Set variable JY equal to PEEK(56321)AND 15, which reads the joystick plugged into port #1.
30	The ON GOSUB statement is useful when reading a joystick. Using ON GOSUB, the value of JY sends the program to different locations. If the value of JY is 1, for instance, the program shifts to line 35. Since 1 isn't a value returned for a directional movement, it simply returns to line 30 and waits for another value. If JY = 14, however, that means the joystick was pushed up, and the program shifts to line 110, where Y = Y − 3. The sprite then moves up. (Refer to Figure 8-1, Joystick Values, for the numbers returned for each direction.) The program then goes back to line 25 and reads the joystick again.
35	RETURN the program to line 30 if an unusable value is read from the joystick.
40	First execute line 60, movement to the right, then line 100, movement down, to create the diagonal movement down and to the right.
50	Same as line 40, but movement to the right and movement up (line 110) are combined for diagonal movement up and to the right.
60-65	Move the sprite to the right only. Keep the sprite on the screen, to the left of the seam.
70-75	Move the sprite to the left. The sprite stays in the viewing area because of the expression IF X<24 THEN X = 24.

80 Execute line 100, movement down, then execute line 70, movement to the left, to create diagonal movement down and to the left.

90 Same as line 100, except that movement up is combined with movement to the left. This makes the sprite move diagonally up and to the left.

100-105 Move the sprite downward unless Y is greater than 229.

110-115 Move the sprite upward unless Y is less than 50.

300-320 Sprite setup. Variable *V* is set, screen is cleared, sprite coordinates are set, the sprite is READ from DATA and turned on. Sprite color set.

500-520 DATA statements for the sprite.

Program 8-8. Diagonal Sprite Joystick Movement

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 GOSUB 300                                    :rem 116
25 JY= (PEEK(56321)AND15)                        :rem 72
30 ON JY GOSUB 35,35,35,35,40,50,60,35,80,90,70,30
   ,100,110,35:GOTO25                           :rem 194
35 RETURN                                        :rem 72
40 GOSUB 60:GOSUB 100:RETURN                    :rem 175
50 GOSUB 60:GOSUB 110:RETURN                    :rem 177
60 X=X+3:IF X>255 THEN X=255                     :rem 12
65 POKE V+0,X:RETURN                            :rem 233
70 X=X-3:IF X<24 THEN X=24                      :rem 161
75 POKE V+0,X:RETURN                            :rem 234
80 GOSUB 100:GOSUB 70:RETURN                    :rem 180
90 GOSUB 110:GOSUB 70:RETURN                    :rem 182
100 Y=Y+3:IF Y>229 THEN Y=229                    :rem 61
105 POKE V+1,Y:RETURN                            :rem 22
110 Y=Y-3:IF Y<50 THEN Y=50                     :rem 206
115 POKE V+1,Y:RETURN                            :rem 23
300 V=53248:PRINT"{CLR}":POKE 53281,1:X=160:Y=100
                                                 :rem 90
310 FOR I=0 TO 63:READ A:POKE 12288+I,A:NEXT
                                                :rem 156
320 POKE V+21,1:POKE 2040,192:POKE V+39,0:POKE V+0
    ,X:POKE V+1,Y:RETURN                          :rem 6
500 DATA 0,0,0,0,0,0,0,0,0,0,0,0,0,16,0,0,16,0,0,1
    6,0,0,16,0,0                                 :rem 91
510 DATA 56,0,0,124,0,0,170,0,1,85,0,1,255,0,1,255
    ,0,0,170,0,1,41,0                           :rem 136
520 DATA 2,40,128,0,0,0,0,0,0,0,0,0,0,0,0,0
                                                :rem 230
```

The sprite stays on the screen, does not wrap around, and also uses only the viewing area to the left of the seam. This simplifies programming. Of course, you can make it move across the seam by adding lines which test for the seam, POKE in the V+16 register, and reset the X coordinate. Look at Program 8-4 for an example of how this was done when you used the keyboard to move sprites. You can use the same programming techniques to add this feature to joystick sprite movement.

The joystick button. As long as we're reading the joystick, let's find another use for the button. By adding lines 15, 20, and 120, and changing line 30 in Program 8-8, we can make the screen change color each time the fire button is pressed.

```
15 FB=-((PEEK(56321) AND 16)=0)                    :rem 24
20 ON FB GOSUB 120                                :rem 154
30 ON JY GOSUB 35,35,35,35,40,50,60,35,80,90,70,30
   ,100,110,35:GOTO15                             :rem 193
120 R=INT(RND(9)*16):POKE 53281,R:RETURN           :rem 97
```

Line 15 reads the fire button and sets the value as variable FB. Line 20 uses another ON/GOSUB statement to shift the program to line 120 if FB is 1, which is the value returned when the button is pressed. If the fire button is not pressed, line 15 returns a value of 0, and the program continues as it did before. Line 120 generates a random number from 0 to 15, then POKEs that number into location 53281 to alter screen background color. You had to change the end of line 30 to GOTO 15 so that the fire button is read each time through the main loop.

Machine Language Movement Routines

The process of reading the keyboard or the joystick is something that may not seem complicated once you see a few examples, but it does consume valuable space within your program's main loop. In BASIC games this is important, for it determines the speed with which your characters or sprites move about the screen. The more elaborate your main program loop is, the slower everything moves.

An all-BASIC game can certainly move fast enough to keep the player's interest, but compromises must often be made. Perhaps diagonal movement has to be thrown out in order to make the sprites move at a reasonable speed. Maybe

more than a handful of sprites on the screen slow the game down to a crawl. Machine language routines can speed up this movement.

Machine language is the native language of your 64. It can be hundreds, even thousands, of times faster than BASIC. When a program executes in BASIC, the 64's BASIC interpreter looks at each statement, decides what it means, and translates it into machine language. Machine language routines by-pass this interpretation and translation, and speed up the execution of a program. One of the easiest ways to see machine language's abilities is using it to move characters or sprites.

Programs written in machine language look quite different from those written in BASIC. The final product of a machine language routine is a series of numbers that are placed in particular memory locations. Once you have these numbers, you simply place them in a number of DATA statements and POKE them into an area of computer memory that's available. The 64 then looks to the numbers in this area of memory and executes its instructions.

If you don't know any machine language, even if you don't want to try, don't worry. You don't have to know machine language to use these movement routines. All you have to do is correctly enter the routine, and the computer will do all the rest.

Where they go. Because these machine language routines are little more than values stored in certain memory locations, you first have to find a place to store them. One of the easiest places to locate these machine language instructions is in the cassette buffer.

Usually used to temporarily store information transferred to and from the cassette drive, the cassette buffer uses memory locations 828 to 1019. Since this area is also protected from the computer's BASIC program, it is an ideal area to place machine language DATA.

This can be a problem if you're using a cassette drive to LOAD and SAVE programs, however. When you use the cassette buffer for machine language storage, you can't use the cassette drive, too. Once you have finished LOADing a program, you can use this area of memory to store the routines. You won't be able to save anything on tape, though, since the buffer will be occupied by strange DATA.

SYSing

Once you have the machine language routine in place, you get it to run by typing (or having the program enter it for you) the command SYS(*n*), where *n* is the memory location of the beginning of the routine. If you stored a machine language routine at the start of the cassette buffer, for instance, you would need a SYS(828) to access the routine. The routine automatically returns to your BASIC program when it is finished, which usually takes only a millisecond or so. The following routines all give the SYS address needed to access the routine, as well as the DATA numbers which are POKEd into the cassette buffer to form the routine itself.

Keyboard-Controlled Character-Movement Routines

This first machine language routine reads the keyboard and places values into locations 828 and 829. Once the values are there, you can use them to create an equation which produces the new screen location of the character. Since this is a simple machine language routine, you'll still have to do much of this work yourself.

The routine checks location 197 (just as you did when you created a program which read the keyboard through PEEKs). It then compares the key code value with the values you established as representing the four possible directions and the fire button. You decide which four keys are used for movement, and which is used for the fire button. The key codes must be POKEd into memory locations before the main loop begins. The direction for each key, and the memory location to POKE its key code into, are:

UP—key code value POKEd into location 251
RIGHT—key code value POKEd into location 252
LEFT—key code value POKEd into location 253
DOWN—key code value POKEd into location 254
FIRE BUTTON—key code value POKEd into location 80

This routine is accessed by a SYS(831), which should be in a line within your program's main loop. Once it has been executed, you'll find that the routine has placed values into locations 828 and 829 which will allow you to calculate a new location for your character. The formulas are:

NL(New Location) = OL(Old Location)
+ (PEEK(828) – PEEK(829))

You can then POKE in the character's new location and position in color memory using the value NL. You can also erase the old image with the line:

POKE NL – (PEEK(828) – PEEK(829)),32

The routine does not create a character, nor does it keep the character on the screen. You'll need to provide those in the program. You can insert this into any program you write, which is one of the advantages of machine language routines. Program 8-9 is the BASIC loader and DATA statements for this routine.

Program 8-9. ML Character Keyboard Controller

```
10 REM BASIC LOADER FOR KEYBOARD CONTROLLED CHARAC
   TER MACHINE LANGUAGE ROUTINE          :rem 120
20 FOR X = 831 TO 926: READ A: POKE X, A: NEXT
                                         :rem 252
1000 REM DATA FOR MACHINE LANGUAGE ROUTINE  :rem 5
1020 DATA 165,197,251,208,13,169,0,141,60,3,16
     9,40,141,61,3,96,234                :rem 120
1030 DATA 234,197,252,208,13,169,1,141,60 :rem 220
1040 DATA 3,169,0,141,61,3,96,234,234,197,253,208,
     13,169,0,141,60,3,169               :rem 163
1050 DATA 1,141,61,3,96,234,234,197,254   :rem 126
1060 DATA 208,13,169,40,141,60,3,169,0,141,61,3,96
     ,234,234,197,80,208,8               :rem 166
1070 DATA 169,1,141,62,3,96,234,234,169   :rem 133
1080 DATA 0,141,60,3,141,61,3,141,62,3,96,234,234
                                         :rem 84
```

To see this machine language routine in action, complete with a moving character and programmed keys for movement, look at Program 8-10.

Here's how this demonstration works:

Line	Function
10	Set screen color and clear screen.
15	BASIC loader section of the machine language routine. DATA beginning in line 1020 is POKEd into the cassette buffer.
20	Establish variables L (beginning location), C (character's screen code), CM (number added to location to place the character in the right position in color memory). Also POKE in the character and its color (yellow).

25	POKE the key codes into the proper locations to set the keys which determine movement. @ is up, = is right, : is left, and / is down.
30	Access the machine language routine with SYS(831) and set the variable *R* as PEEK(828) – PEEK(829).
40	If no key is pressed, shift program back to line 30.
50-55	Establish L (New Location of character), then erase it. Last expression keeps the character on the screen. (Unfortunately, it will jump from top or bottom of the screen to either the upper-left corner or the lower-right corner. Stopping at the top or bottom would mean more complicated programming—something a demonstration program doesn't need.)
60	POKE in L and also new color memory location (L + CM,7).
70	Complete the main loop by beginning it again.
1020-1080	DATA statements for machine language routine.

Program 8-10. Character Keyboard Demo

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 POKE 53281,0:PRINT"{CLR}{WHT}"          :rem 146
15 FOR X=831 TO 926:READ A:POKE X,A:NEXT    :rem 0
20 L=1520:C=42:CM=54272:POKE L,C:POKE L+CM,7
                                            :rem 213
25 POKE 251,46:POKE 252,53:POKE 253,45:POKE 254,55
                                            :rem 140
30 SYS (831):R=PEEK(828)-PEEK(829)          :rem 118
40 IF R=0 THEN 30                            :rem 68
50 L=L+R:POKE L-R,32:IF L>2023 THEN L=2023  :rem 21
55 IF L<1024 THEN L=1024                    :rem 199
60 POKE L,C:POKE L+CM,7                      :rem 35
70 GOTO 30                                    :rem 3
1000 REM DATA FOR MACHINE LANGUAGE ROUTINE   :rem 5
1020 DATA 165,197,197,251,208,13,169,0,141,60,3,16
     9,40,141,61,3,96,234                   :rem 120
1030 DATA 234,197,252,208,13,169,1,141,60   :rem 220
1040 DATA 3,169,0,141,61,3,96,234,234,197,253,208,
     13,169,0,141,60,3,169                  :rem 163
1050 DATA 1,141,61,3,96,234,234,197,254     :rem 126
1060 DATA 208,13,169,40,141,60,3,169,0,141,61,3,96
     ,234,234,197,80,208,8                  :rem 166
1070 DATA 169,1,141,62,3,96,234,234,169     :rem 133
```

```
1080 DATA 0,141,60,3,141,61,3,141,62,3,96,234,234
                                          :rem 84
```

A Faster Machine Language Routine

This second machine language routine has some features that will do some of the steps you had to program yourself in the last routine. This routine automatically POKEs the character into its new position on the screen, and keeps it on the screen for you. The character will not return to the corners, as in the last routine, but instead will simply stop at the top or bottom edges when they are reached.

However, you'll lose the ability to test for collisions with other characters with this routine. This will not affect collision detection between sprites, which have a separate register for this. The chapter on collision detection will cover this more closely, so if you don't understand it yet, don't worry.

You will still have to provide parameters for this routine. First of all, the routine needs to know the starting location of your character, both in screen and color memory. There are four memory locations set aside that will contain pointers to the screen and color memory addresses. These are locations 251-254.

Setting these pointers involves a few steps that may seem complicated at first, but should be clear once you've gone through them. Consult the Screen Location Table in Appendix C and decide where the character will first appear. Take that screen location and divide by 256. The number, rounded down to the nearest integer, is then POKEd into location 252. For example:

Starting screen location = 1904
1904/256 = 7.4375 Rounded down to 7
POKE 252,7

Second, take the remainder from that division and POKE the number into both location 251 and location 253. For instance:

1904/256 = 7 with a remainder of 112
POKE 251,112:POKE 253,112

Next, take the number you just POKEd into 252 (7) and add 212 to it. This number is POKEd into location 254.

7 + 212 = 219
POKE 254,219

What you've just done is set the pointers that contain the least significant byte (LSB) and most significant byte (MSB) of the screen and color memory locations of the character. If you haven't worked with machine language before, this may sound confusing, but all you really have to know is how to calculate the appropriate values for these POKEs.

Once the location pointers are set, you also need to decide what character to use, and which color it will be. If you were using this routine with a custom character, for instance, you could easily substitute one as the player-controlled character. Whether you use a standard character, or a custom character, its screen code value must be POKEd into location 832, and its color value into location 833.

The last thing is to decide which keys you'll use for each direction of movement. Enter the key code values for your chosen keys as POKEs in these locations:

UP at location 828
DOWN at location 829
LEFT at location 830
RIGHT at location 831

A total of 10 POKEs are needed for this routine before it can execute properly. Here's the BASIC loader and DATA statements for this machine language routine, which is SYSed with 834:

Program 8-11. BASIC Loader and DATA Statements

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 REM BASIC LOADER FOR FAST(CKX) KEYBOARD CONTROL
   LED CHARACTER ROUTINE                    :rem 164
20 FOR X = 834 TO 1014: READ A: POKE X , A: NEXT
                                             :rem 36
1020 DATA 32,176,3,165,197,205,60,3,208,14,32,183,
     3,32,228,3,176,63,32                    :rem 107
1030 DATA 160,3,76,147,3,205,61,3,208,14,32,160,3,
     32,237,3,144,44,32                      :rem 245
1040 DATA 183,3,76,147,3,205,62,3,208,14,32,199,3,
     32,228,3,176,25,32,217                  :rem 210
1050 DATA 3,76,147,3,205,63,3,208,14,32,217,3,32,2
     37,3,144,6,32,199,3                      :rem 53
1060 DATA 76,147,3,160,0,173,64,3,145,251,173,65,3
     ,145,253,96,24,165                      :rem 24
1070 DATA 251,105,40,133,251,133,253,144,4   :rem 1
```

```
1080 DATA 230,252,230,254,96,160,0,169,32,145,251,
     96,56,165,251,233,40                    :rem 120
1090 DATA 133,251,133,253,176,4,198,252,198,254,96
     ,165,251,208,9,198,251                  :rem 254
1100 DATA 198,253,198,252,198,254,96,198,251,198,2
     53,96,230,251,230,253,208               :rem 154
1110 DATA 4,230,252,230,254,96,165,251,201,1,165,2
     52,201,4,96,165,251                      :rem 53
1120 DATA 201,232,165,252,233,7,96,234        :rem 72
```

Entering the POKEs can be done in the program, just before the machine language routine executes. Program 8-12 is a demonstration of a moving character using this routine. The required POKEs are already included in lines 10 through 40, including the BASIC loader statement in line 35 and the SYS(834) command in line 40.

This is how the program works:

Line	Function
10	Set background color to black and clear screen.
20	Set pointers for screen and color memory locations for the character. Location 1279 was chosen as the screen memory starting location and the calculations for each POKE statement were made on that basis.
30	Set keys for movement. @ is up, = is right, : is left, and / is down.
35	BASIC loader for machine language DATA.
40	Access the machine language routine.
1020-1120	Machine language DATA statements.

Program 8-12. Fast Character Keyboard Demo

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 POKE 53281,0: PRINT"{CLR}"                 :rem 141
20 POKE 251,255:POKE 252,4:POKE 253,PEEK(251):POKE
   254,PEEK(252)+212                          :rem 143
30 POKE 828,46:POKE 829,55:POKE 830,45:POKE 831,53
   :POKE 832,42:POKE 833,7                    :rem 160
35 FOR X=834 TO 1014:READ A:POKE X,A:NEXT     :rem 42
40 SYS(834)                                   :rem 83
50 GOTO 40                                     :rem 2
```

```
1020 DATA 32,176,3,165,197,205,60,3,208,14,32,183,
     3,32,228,3,176,63,32                    :rem 107
1030 DATA 160,3,76,147,3,205,61,3,208,14,32,160,3,
     32,237,3,144,44,32                      :rem 245
1040 DATA 183,3,76,147,3,205,62,3,208,14,32,199,3,
     32,228,3,176,25,32,217                  :rem 210
1050 DATA 3,76,147,3,205,63,3,208,14,32,217,3,32,2
     37,3,144,6,32,199,3                     :rem 53
1060 DATA 76,147,3,160,0,173,64,3,145,251,173,65,3
     ,145,253,96,24,165                      :rem 24
1070 DATA 251,105,40,133,251,133,253,144,4   :rem 1
1080 DATA 230,252,230,254,96,160,0,169,32,145,251,
     96,56,165,251,233,40                    :rem 120
1090 DATA 133,251,133,253,176,4,198,252,198,254,96
     ,165,251,208,9,198,251                  :rem 254
1100 DATA 198,253,198,252,198,254,96,198,251,198,2
     53,96,230,251,230,253,208               :rem 154
1110 DATA 4,230,252,230,254,96,165,251,201,1,165,2
     52,201,4,96,165,251                     :rem 53
1120 DATA 201,232,165,252,233,7,96,234       :rem 72
```

Joystick-Controlled Character Routine

This routine is much like the first keyboard-controlled machine language routine, Program 8-9. It places values into locations 253 and 254 that allow you to calculate the new screen location of the character using the line:

L = L + PEEK(254) − PEEK(253)

This routine also will place a value of 1 in location 80 if the fire button is pressed. You could use this by having the line:

IF PEEK(80) = 1 THEN PRINT "FIRE!"

where the PRINT statement is replaced by whatever function you assign to the fire button.

You'll still have to keep the character on the screen and erase the previous character image. The routine reads the joystick and returns the values needed to calculate the character's new screen location.

If you want to add the routine to your own program, simply enter the following program in the proper place. A SYS(828) accesses this routine.

Program 8-13. Joystick Character Loader and DATA

```
10 REM BASIC LOADER FOR JOYSTICK CONTROLLED CHARAC
   TER MACHINE LANGUAGE ROUTINE              :rem 151
20 FOR X = 828 TO 973: READ A: POKE X, A: NEXT
                                             :rem 4
```

```
1020 DATA 173,1,220,74,176,40,74,74,176,11,169,0
                                                :rem 52
1030 DATA 133,254,169,41,133,253,76,182,3,74,176,1
     1,169,0,133,254,169,39                     :rem 230
1040 DATA 133,253,76,182,3,169,0,133,254,169,40,13
     3,253,76,182,3,74,176                      :rem 178
1050 DATA 39,74,176,11,169,39,133,254,169,0,133,25
     3,76,182,3,74,176,11                       :rem 136
1060 DATA 169,41,133,254,169,0,133,253,76,182,3,16
     9,40,133,254,169,0                         :rem 26
1070 DATA 133,253,76,182,3,74,176,11,169,0,133,254
     ,169,1,133,253,76,182,3                    :rem 16
1080 DATA 74,176,11,169,1,133,254,169,0,133,253,76
     ,182,3,169,0,133,254                       :rem 123
1090 DATA 133,253,173,1,220,74,74,74,74,74,176,7,1
     69,1,133,80,76,203,3                       :rem 126
1100 DATA169,0,133,80,96,234,234                :rem 84
```

The following demonstration program shows this machine language routine in action. It uses the same character as the first keyboard routine, as well as the same methods of keeping the character on the screen and moving the character. The only real difference is that the joystick plugged into port #1 is used instead of the keyboard.

Program 8-14. Joystick Character Demo

```
10 POKE 53281,0:PRINT"{CLR}{WHT}"              :rem 146
15 FOR X=828 TO 973:READ A:POKE X,A:NEXT        :rem 8
20 L=1520:C=42:CM=54272:POKE L,C:POKE L+CM,7
                                                :rem 213
30 SYS(828):R=PEEK(254)-PEEK(253)               :rem 108
40 IF R=0 THEN 30                               :rem 68
50 L=L+R:POKE L-R,32:IF L>2023 THEN L=2023      :rem 21
55 IF L<1024 THEN L=1024                        :rem 199
60 POKE L,C:POKE L+CM,7                         :rem 35
70 GOTO 30                                      :rem 3
1000 REM MACHINE LANGUAGE ROUTINE DATA          :rem 30
1020 DATA 173,1,220,74,176,40,74,74,176,11,169,0
                                                :rem 52
1030 DATA 133,254,169,41,133,253,76,182,3,74,176,1
     1,169,0,133,254,169,39                     :rem 230
1040 DATA 133,253,76,182,3,169,0,133,254,169,40,13
     3,253,76,182,3,74,176                      :rem 178
1050 DATA 39,74,176,11,169,39,133,254,169,0,133,25
     3,76,182,3,74,176,11                       :rem 136
1060 DATA 169,41,133,254,169,0,133,253,76,182,3,16
     9,40,133,254,169,0                         :rem 26
```

```
1070 DATA 133,253,76,182,3,74,176,11,169,0,133,254
     ,169,1,133,253,76,182,3                    :rem 16
1080 DATA 74,176,11,169,1,133,254,169,0,133,253,76
     ,182,3,169,0,133,254                       :rem 123
1090 DATA 133,253,173,1,220,74,74,74,74,74,176,7,1
     69,1,133,80,76,203,3                       :rem 126
1100 DATA169,0,133,80,96,234,234                :rem 84
```

Adding this expression as line 35 will show the results of pressing the fire button:

IF PEEK(80) = 1 THEN PRINT "FIRE!"

Run the program again, watching as you press the fire button. A message should display each time.

Keyboard-Controlled Sprite-Movement Routine

This machine language routine moves sprite 0 around the screen under the control of the keys you select. This routine uses wraparound at all four edges of the screen and also allows the sprite to cross the seam. You can also designate a fire button key read from location 828.

As in the previous examples, you have to assign the keys for each direction of movement. POKE the key code values of your keys into these locations:

UP at 251
DOWN at 252
LEFT at 253
RIGHT at 254
FIRE BUTTON at 80

The BASIC loader and DATA statements for this routine look like this:

Program 8-15. BASIC Loader and DATA for Keyboard Sprite

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 REM LOADER FOR KEYBOARD CONTROLLED SPRITE MACHI
   NE LANGUAGE ROUTINE                          :rem 96
20 FOR X= 830 TO 953: READ A: POKE X , A: NEXT
                                                :rem 251
999 REM MACHINE LANGUAGE ROUTINE DATA            :rem 8
1000 DATA165,197,197,251,208,4,206,1,208,96,197,25
     2,208                                      :rem 213
1010 DATA4,238,1,208,96,197,253,208,38,173,0,208,2
     08,29,173,16,208,41                        :rem 126
```

```
1020 DATA1,208,14,173,16,208,9,1,141,16,208,169,80
     ,141,0,208,96,173,16                    :rem 155
1030 DATA208,41,254,141,16,208,206,0,208,96,197,25
     4,208,42,238,0,208,240                   :rem 5
1040 DATA28,169,80,205,0,208,208,20,173,16,208,41,
     1,240,13,173,16,208                     :rem 98
1050 DATA41,254,141,16,208,169,0,141,0,208,96,173,
     16,208,9,1,141,16,208                  :rem 203
1060 DATA96,197,80,208,6,169,1,141,60,3,96,169,0,1
     41,60,3,96,234,234                       :rem 80
```

Use SYS(830) to access this routine.

A demonstration program using this machine language routine could look like this:

Program 8-16. Keyboard-Controlled Sprite Demo

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 POKE 53281,0:V=53248:X=120:Y=120:PRINT"{CLR}"
                                             :rem 37
15 FOR X=830 TO 953:READ A:POKE X,A:NEXT   :rem 255
20 FOR X=0 TO 63:READ A:POKE 12288+X,A:NEXT
                                            :rem 136
30 POKE 2040,192                             :rem 32
40 POKE V+21,1                              :rem 212
50 POKE V+39,10                              :rem 14
60 POKE V+0,X                               :rem 202
70 POKE V+1,Y                               :rem 205
80 POKE 251,46:POKE 252,55:POKE 253,45:POKE 254,53
                                            :rem 141
100 SYS(830):GOTO100                        :rem 128
999 REM MACHINE LANGUAGE ROUTINE DATA         :rem 8
1000 DATA165,197,197,251,208,4,206,1,208,96,197,25
     2,208                                  :rem 213
1010 DATA4,238,1,208,96,197,253,208,38,173,0,208,2
     08,29,173,16,208,41                    :rem 126
1020 DATA1,208,14,173,16,208,9,1,141,16,208,169,80
     ,141,0,208,96,173,16                   :rem 155
1030 DATA208,41,254,141,16,208,206,0,208,96,197,25
     4,208,42,238,0,208,240                   :rem 5
1040 DATA28,169,80,205,0,208,208,20,173,16,208,41,
     1,240,13,173,16,208                     :rem 98
1050 DATA41,254,141,16,208,169,0,141,0,208,96,173,
     16,208,9,1,141,16,208                  :rem 203
1060 DATA96,197,80,208,6,169,1,141,60,3,96,169,0,1
     41,60,3,96,234,234                       :rem 80
59999 REM SPRITE DATA                       :rem 238
```

```
60000 DATA 0,0,0,0,0,0,7,255,224,15,255,240,25,36,
      152,31,255,248,15,255                  :rem 127
60010 DATA 240,7,255,224,0,24,0,32,189,4,49,255,14
      0,43,255,212,36,60,36,40,24            :rem 187
60020 DATA 20,48,36,12,32,90,4,0,189,0,1,126,128,3
      ,0,192,0,0,0,0,0,0,36,40,24            :rem 135
```

The main movement loop is only one line, line 100. The rest of this program sets up the sprite parameters such as color, starting X and Y coordinate positions, pointers to memory, and sprite DATA information to draw the sprite. Line 80 POKEs in the desired keys for movement. The pattern @ = : / is used again.

As with all these machine language routines and demonstrations, you can use your own program lines instead of what appears here. For example, in the last program, you could replace the sprite DATA information in lines 60000-60020 with numbers which would draw a sprite of your own creation.

Faster sprites. To speed up the sprite movement even more, you can make a few changes to the machine language routine's BASIC loader and DATA statements. After you've entered these changes, you even have the option of deciding how fast the sprite will move across the screen. You can have it move one, two, three, or four pixels at a time. The more pixels it moves at once, the jerkier the motion seems to be. As with many other game design decisions, you may have to compromise, losing some grace in movement for speed.

To move the sprite faster, you'll need to add one line and change two others.

Add this line to the machine language DATA statements:

```
1070 DATA 32,62,3,32,62,3,32,62,3,32,62,3,96
```

Next, change the BASIC loader line to read:

```
15 FOR X=830 TO 966:READ A:POKE X,A:NEXT
```

This will include the 13 new DATA numbers in line 1070 when the machine language routine is READ and then executed.

Finally, you need to change the SYS(830) command in line 100 to reflect how fast you want the sprite to move.

For a 4-pixel move, SYS(954)
For a 3-pixel move, SYS(957)
For a 2-pixel move, SYS(960)
For a 1-pixel move, leave it at SYS(830)

If you find that the fastest movement is too jumpy, simply change the SYS command to the desired speed.

Joystick-Controlled Sprite Routine

This is essentially the same as the previous machine language routine. The sprite is moved on the screen with a joystick controller, but it wraps around on all edges, as well as smoothly crosses the seam. Since you're using a joystick in this routine, the sprite will also move diagonally instead of just horizontally and vertically. However, the fire button is *not* read by this routine. To do that, you'll have to add the line:

```
FB=PEEK(56321) AND 16:IF FB=16 THEN PRINT "FIRE!"
```

where the PRINT statement can be changed to whatever function you want to assign to the fire button.

The BASIC loader and DATA statements for the machine language program are:

Program 8-17. Joystick Sprite BASIC Loader and DATA

```
10 REM BASIC LOADER FOR JOYSTICK CONTROLLED SPRITE
    MACHINE LANGUAGE ROUTINE                :rem 225
20 FOR X=828 TO 936:READ A:POKE X,A:NEXT     :rem 3
990 REM DATA FOR JS ROUTINE                 :rem 74
995 DATA 173,1,220,74,176,3,206,1,208,74,176,3,238
    ,1,208,74,176,42,173                    :rem 134
996 DATA 0,208,208,31,173,16,208,41,1,208,16,173,1
    6,208,9,1,141,16,208                    :rem 114
997 DATA 169,80,141,0,208,96,234,234,173,16,208,41
    ,254,141,16,208,206                     :rem 87
998 DATA 0,208,96,234,234,74,176,32,238,0,208,240,
    30,169,80,205,0,208                     :rem 82
999 DATA 208,20,173,16,208,41,1,240,13,173,16,208,
    41,254,141,16,208,169                   :rem 172
1000 DATA 0,141,0,208,96,234,234,173,16,208,9,1,14
     1,16,208,96,234,234                    :rem 54
```

The routine is accessed with a SYS(828).

The demonstration program uses the same sprite as Program 8-16, so you may want to simply LOAD that program into your computer and change only the BASIC loader line (line 15) and the lines of machine language DATA statements (lines 990-1000).

Program 8-18 shows the machine language routine moving a sprite around the screen with a joystick controller.

Program 8-18. Joystick Sprite Demo

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 POKE 53281,0:PRINT"{CLR}":V=53248:X=120:Y=120
                                              :rem 37
15 FOR X=828 TO 936:READ A:POKE X,A:NEXT      :rem 7
20 FOR X=0 TO 63:READ A:POKE 12288+X,A:NEXT
                                             :rem 136
30 POKE 2040,192                              :rem 32
40 POKE V+21,1                               :rem 212
50 POKE V+39,10                               :rem 14
60 POKE V+0,X                                :rem 202
70 POKE V+1,Y                                :rem 205
80 FOR T=0 TO 1000:SYS(828):NEXT:GOTO 80     :rem 244
990 REM DATA FOR JS ROUTINE                   :rem 74
995 DATA 173,1,220,74,176,3,206,1,208,74,176,3,238
    ,1,208,74,176,42,173                     :rem 134
996 DATA 0,208,208,31,173,16,208,41,1,208,16,173,1
    6,208,9,1,141,16,208                     :rem 114
997 DATA 169,80,141,0,208,96,234,234,173,16,208,41
    ,254,141,16,208,206                       :rem 87
998 DATA 0,208,96,234,234,74,176,32,238,0,208,240,
    30,169,80,205,0,208                       :rem 82
999 DATA 208,20,173,16,208,41,1,240,13,173,16,208,
    41,254,141,16,208,169                    :rem 172
1000 DATA 0,141,0,208,96,234,234,173,16,208,9,1,14
     1,16,208,96,234,234                      :rem 54
59999 REM SPRITE DATA                        :rem 238
60000 DATA0,0,0,0,0,0,7,255,224,15,255,240,25,36,1
      52,31,255,248,15,255                   :rem 127
60010 DATA 240,7,255,224,0,24,0,32,189,4,49,255,14
      0,43,255,212,36,60,36,40,24            :rem 187
60020 DATA 20,48,36,12,32,90,4,0,189,0,1,126,128,3
      ,0,192,0,0,0,0,0,0,36,40,24            :rem 135
```

Both routines for moving sprites allow the sprite to smoothly cross the seam on the screen. This is one of the advantages of using machine language movement routines when sprites are needed in your game.

Even faster sprites. As with the keyboard-controlled sprites, you can make the sprite move even faster by adding a line to the machine language DATA statements and changing two lines in the demonstration program. You'll still have the option of four different speeds for the sprite.

Add this line to the machine language DATA statements:

```
1010 DATA 32,60,3,32,60,3,32,60,3,32,60,3,96
```

To insure that these additional numbers are POKEd into the 64's memory, change line 15 to read:

```
15 FOR X=828 TO 949:READ A:POKE X,A:NEXT
```

Setting the sprite's speed is a matter of using the correct SYS command. Change line 80 as follows:

For a 4-pixel move, SYS(937)
For a 3-pixel move, SYS(940)
For a 2-pixel move, SYS(943)
For a 1-pixel move, leave it at SYS(828)

All of the machine language subroutines listed up to now have moved only one sprite. What if you want to use two player-controlled sprites, using both of the Commodore 64's joystick ports, and want the speed of machine language?

Gregg Peele, assistant programming supervisor at COMPUTE! Publications, has written an excellent machine language routine which moves two sprites, one with each joystick. All the sprite movement calculations are done by the routine, including checking for the seam and keeping the sprites on the screen. The demonstration of this routine even creates two block-shaped sprites for you to move across the screen. Of course, you can enter DATA information for your own sprites, as well as select the sprites' colors. By placing this routine in your own program, you'll have the ability to use both joysticks to control sprites. Although writing a two-player game is somewhat more difficult, this routine will make it much simpler.

Program 8-19. Two-Sprite Joystick Routine

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
40000 FORT=12288TO12288+128:POKET,255:NEXT:REM ADD
      SPRITE DATA HERE                      :rem 217
40010 I=49152                               :rem 128
40020 READ A:IF A=256 THEN40040              :rem 94
40030 POKE I,A:I=I+1:CK=CK+A:GOTO40020      :rem 169
40040 IFCK<>71433THENPRINT"{CLR}ERROR IN DATA STAT
      EMENTS":END                           :rem 120
40050 INPUT "COLOR FOR SPRITE0";C0:POKE49228,C0
                                            :rem 249
```

```
40060 INPUT "COLOR FOR SPRITE1";C1:POKE49233,C1
                                                :rem 249
40070 SYS49152                                  :rem 255
49152 DATA 169,125,141,224,207,169,0            :rem 246
49160 DATA 141,225,207,169,125,141,226           :rem 86
49168 DATA 207,169,0,141,227,207,169              :rem 1
49176 DATA 125,141,0,208,141,1,208              :rem 134
49184 DATA 141,2,208,141,3,208,169              :rem 145
49192 DATA 0,141,16,208,120,169,52              :rem 140
49200 DATA 141,20,3,169,192,141,21              :rem 129
49208 DATA 3,88,96,162,0,32,65                  :rem 212
49216 DATA 192,162,1,32,65,192,76               :rem 103
49224 DATA 49,234,169,3,141,21,208              :rem 147
49232 DATA 169,192,141,248,7,169,1              :rem 158
49240 DATA 141,39,208,169,2,141,40              :rem 141
49248 DATA 208,169,193,141,249,7,189             :rem 18
49256 DATA 0,220,41,15,157,228,207              :rem 141
49264 DATA 56,169,15,253,228,207,157               :rem 8
49272 DATA 232,207,160,0,200,152,221            :rem 224
49280 DATA 232,207,208,249,224,1,208            :rem 245
49288 DATA 2,162,2,152,10,168,185               :rem 100
49296 DATA 135,192,72,185,134,192,72               :rem 9
49304 DATA 96,1,194,213,193,217,193             :rem 206
49312 DATA 1,194,225,193,229,193,236            :rem 255
49320 DATA 193,1,194,221,193,250,193            :rem 246
49328 DATA 243,193,1,194,169,50,221             :rem 203
49336 DATA 1,208,176,12,189,1,208               :rem 100
49344 DATA 56,189,1,208,233,1,157               :rem 104
49352 DATA 1,208,96,169,229,221,1               :rem 102
49360 DATA 208,144,12,189,1,208,24              :rem 145
49368 DATA 189,1,208,105,1,157,1                 :rem 50
49376 DATA 208,96,56,189,224,207,233             :rem 14
49384 DATA 65,157,228,207,189,225,207            :rem 63
49392 DATA 233,1,29,228,207,144,13              :rem 147
49400 DATA 169,65,157,224,207,169,1             :rem 205
49408 DATA 157,225,207,76,247,192,24               :rem 6
49416 DATA 189,224,207,105,1,157,224            :rem 249
49424 DATA 207,189,225,207,105,0,157            :rem 249
49432 DATA 225,207,56,189,224,207,233            :rem 48
49440 DATA 0,157,228,207,189,225,207            :rem 253
49448 DATA 233,1,29,228,207,144,19              :rem 155
49456 DATA 224,2,240,34,173,16,208              :rem 145
49464 DATA 9,1,141,16,208,189,224               :rem 104
49472 DATA 207,157,0,208,96,224,2               :rem 101
49480 DATA 240,30,173,16,208,41,254             :rem 192
49488 DATA 141,16,208,189,224,207,157            :rem 58
49496 DATA 0,208,96,173,16,208,9                 :rem 64
49504 DATA 2,141,16,208,189,224,207             :rem 196
49512 DATA 157,0,208,96,173,16,208              :rem 153
49520 DATA 41,253,141,16,208,189,224            :rem 246
```

```
49528 DATA 207,157,0,208,96,56,189          :rem 170
49536 DATA 224,207,233,25,157,228,207        :rem 47
49544 DATA 189,225,207,233,0,29,228         :rem 207
49552 DATA 207,176,13,169,24,157,224           :rem 1
49560 DATA 207,169,0,157,225,207,76         :rem 207
49568 DATA 127,193,56,189,224,207,233        :rem 63
49576 DATA 1,157,224,207,189,225,207          :rem 4
49584 DATA 233,0,157,225,207,56,189         :rem 212
49592 DATA 224,207,233,0,157,228,207        :rem 250
49600 DATA 189,225,207,233,1,29,228         :rem 201
49608 DATA 207,144,19,224,2,240,34          :rem 144
49616 DATA 173,16,208,9,1,141,16             :rem 47
49624 DATA 208,189,224,207,157,0,208        :rem 255
49632 DATA 96,224,2,240,30,173,16            :rem 94
49640 DATA 208,41,254,141,16,208,189        :rem 252
49648 DATA 224,207,157,0,208,96,173         :rem 211
49656 DATA 16,208,9,2,141,16,208             :rem 51
49664 DATA 189,224,207,157,0,208,96         :rem 216
49672 DATA 173,16,208,41,253,141,16         :rem 198
49680 DATA 208,189,224,207,157,0,208           :rem 1
49688 DATA 96,32,158,192,96,32,178          :rem 183
49696 DATA 192,96,32,198,192,96,32          :rem 182
49704 DATA 78,193,96,32,158,192,32          :rem 169
49712 DATA 78,193,96,32,178,192,32          :rem 170
49720 DATA 78,193,96,32,178,192,32          :rem 169
49728 DATA 198,192,96,32,158,192,32         :rem 225
49736 DATA 198,192,96,96,256                :rem 144
```

This machine language routine is accessed with a SYS(49152), which is where the DATA begins. You'll recognize the BASIC loader line and the sprite setup lines in the beginning of the routine. The only thing you have to do when you use this routine is select the color for each sprite. The routine asks for this information in lines 40050 and 40060. Everything else is done for you.

You can even change the sprites' colors as the program RUNs. All you have to do is press RUN/STOP and POKE a new color value into location 49228 for sprite 0, or location 49233 for sprite 1. The sprite color will instantly change. For example, POKEing 49228, 14 changes sprite 0 to light blue.

It's Not *Pong*, But Why Not Use Paddles?

A game controller often overlooked, perhaps because it has been in use so long, is a set of game paddles. First introduced as controllers for the first real videogame, *Pong*, they can be useful in many kinds of arcade-style games.

Game paddles are somewhat like the volume or tone controls on a radio. The paddle varies the resistance conducted on the wire it's connected to.

They can be used for a variety of purposes. When they are hooked up to the 64 and the right program is in place, the paddles generate numbers as they are twisted to the left or to the right. Twisting the paddle to one side results in the value of 255, while twisting to the other produces a value of 0. How these values are used depends on your decisions and the program you write or use.

The most popular use of a paddle is for character or sprite movement. A paddle is particularly suited for sprite movement, since the sprite positions, ranging from 0 to 255, are the same as the values created by the paddles.

A machine language routine is the best way to program paddle controls. Using BASIC is quite difficult when programming paddles, since they cannot be read accurately unless a machine language routine is in place. BASIC is just too slow for game paddle controllers.

Here's a machine language BASIC loader and DATA statements which will allow you to use game paddles as controllers. The routine will read one or two sets of paddles, as well as their fire buttons. After this routine has been called with a SYS(840), you'll find the following information in these memory locations:

Location	Data
830	Value (0 to 255) of position of paddle A plugged into port #1.
832	Value of position of paddle B plugged into port #1.
829	Fire button for both paddles in port #1. A value of 251 indicates paddle A's fire button is pressed, 247 means paddle B's fire button is pressed, and a value of 243 indicates both paddles' fire buttons are pressed.
831	Value of position of paddle A plugged into port #2.
833	Value of position of paddle B plugged into port #2.
828	Fire button indicator for paddles plugged into port #2. The values returned are the same as those for port #1.

Program 8-20. Paddle BASIC Loader and DATA Statements

```
5 REM BASIC LOADER FOR GAME PADDLE MACHINE LANGUAG
  E READ ROUTINE                                :rem 88
10 FOR X = 840 TO 903: READ A: POKE X, A: NEXT
                                                :rem 246
110 DATA162,1,120,173,2,220,141,70,3,169,192,141,2
    ,220,169,128,141,0                          :rem 31
120 DATA 220,160,128,234,136,16,252,173,25,212,157
    ,62,3,173,26,212,157,64                      :rem 0
130 DATA 3,173,0,220,9,128,141,60,3,169,64,202,16,
    222,173,70,3,141,2,220                      :rem 179
140 DATA 173,1,220,141,61,3,88,96                :rem 82
```

By adding this machine language routine to your own program, you can use game paddles to move characters or sprites across the screen. The following demonstration program creates a sprite, uses the SYS(840) to call the machine language routine, and uses the paddles to move the sprite about the screen. Paddle A moves the sprite horizontally, while paddle B moves it vertically. The paddles should be plugged into port #1.

Program 8-21. Paddle-Controlled Sprite Movement

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
5 REM PADDLE CONTROLLED SPRITE DEMO          :rem 181
6 REM USES PORT 1, ONE PADDLE CONTROLS X POSTION A
  ND OTHER CONTROLS Y POSTION                :rem 206
7 REM FIRE BUTTON NOT INCLUDED                :rem 86
10 POKE 53281,0:V=53248:X=120:Y=120:PRINT"{CLR}"
                                              :rem 37
15 FOR X=840 TO 903:READ A:POKE X,A:NEXT     :rem 251
20 FOR X=0 TO 63:READ A:POKE 12288+X,A:NEXT
                                              :rem 136
30 POKE 2040,192                               :rem 32
40 POKE V+21,1                                :rem 212
50 POKE V+39,10                                :rem 14
60 SYS(840)                                    :rem 82
70 POKE V,PEEK(830):POKE V+1,PEEK(832)        :rem 131
90 GOTO 60                                      :rem 8
105 REM MACHINE LANGUAGE DATA                 :rem 205
110 DATA162,1,120,173,2,220,141,70,3,169,192,141,2
    ,220,169,128,141,0                          :rem 31
120 DATA 220,160,128,234,136,16,252,173,25,212,157
    ,62,3,173,26,212,157,64                      :rem 0
```

```
130 DATA 3,173,0,220,9,128,141,60,3,169,64,202,16,
    222,173,70,3,141,2,220                    :rem 179
140 DATA 173,1,220,141,61,3,88,96              :rem 82
59999 REM SPRITE DATA                         :rem 238
60000 DATA 0,0,0,0,0,0,7,255,224,15,255,240,25,36,
      152,31,255,248,15,255                   :rem 127
60010 DATA 240,7,255,224,0,24,0,32,189,4,49,255,14
      0,43,255,212,36,60,36,40,24             :rem 187
60020 DATA 20,48,36,12,32,90,4,0,189,0,1,126,128,3
      ,0,192,0,0,0,0,0,0,36,40,24             :rem 135
```

Fast-moving sprite, isn't it? Notice that the seam is not crossed. If you wanted to move the sprite in only one direction, more like *Pong*, you could add one line and change one other.

Add this line:

```
55 POKE V,X
```

and change line 70 to:

```
70 POKE V+1,PEEK(832)
```

Now the sprite will move only vertically. The sprite jiggles less, too. This is something peculiar to game paddle controllers, since they are sometimes just between two numbers and trying to generate both numbers at once.

9 Collisions

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 rare exceptions: the overwhelming majority of events are caused by collisions.

Checking for Character Collisions

Checking for collisions between characters 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 in Chapter 5 when we created a scrolling screen display made of flickering stars, with a starship moving through? Let's turn that into a playable game now, by moving the starship around on it using the wraparound movement routine from Chapter 8. We'll use the Commodore and SHIFT keys for left/right movement, and *f5* and *f7* for up/down movement. But before we move the starship, we'll check to see if there's something in the place where the spaceship is trying to go. If we PEEK that location and get any number but 32, we'll know that the starship is colliding with *something*. Then the program will jump to a collision-handling routine and decide what to do next.

What's the story? 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 for repairs and refitting will cause the spaceship to malfunction, and the crew will have to abandon it.

How does this story translate into collision handling?

Collision handling. First, we'll keep 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 spaceship's color from red to cyan, purple, green, and so on.

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.

When the starship collides with a twirling space station, it is refitted and starts again with a red color. It gets no additional points.

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.

How Collision Handling Works:

Line	Function
230	If the character stored at the new location is something other than a blank (32), go to the Collision Routine at 700.
280	POKE the player-figure and its color into screen memory and color memory. If the color is greater than 9, the player-figure has collided with too many small stars—jump to the end routine at 900.
640	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 64 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.)
700-730	*Collision Routine.* There are four possible collisions. If the spaceship has collided with a small star (28), 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 (27), 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 (30), 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 28, a small star, so that it becomes another obstacle to the spaceship.

900 *End Routine.* This routine makes the spaceship flash through the colors, prints the final score, and exits the game.

Program 9-1. Mission: Nova!

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 DIM CC(1),CH(1),CB(1)                          :rem 175
17 REM                                             :rem 76
18 REM SET MEMORY VARIABLES                       :rem 171
19 REM                                             :rem 78
20 CC(0)=12:CC(1)=14:VB=32768:VM=VB/256:SB=128:SC=
   SB*256:CM=55296                                 :rem 166
30 FOR I=0 TO 1:CH(I)=1024*CC(I)+VB:CB(I)=CC(I)+16
   *(SB-VM):NEXT                                   :rem 229
37 REM                                             :rem 78
38 REM SET VIDEO, SCREEN, & CHAR BLOCKS           :rem 204
39 REM                                             :rem 80
40 POKE 648,SB:POKE 56578,PEEK(56578)OR 3:POKE5657
   6,(PEEK(56576)AND252)OR 1                       :rem 167
50 PRINT "{CLR}JUST A MOMENT, PLEASE . . ."
                                                   :rem 144
57 REM                                             :rem 80
58 REM SET UP CHAR SET, SCREEN, STRING            :rem 219
59 REM                                             :rem 82
60 GOSUB 1100:GOSUB 400:GOSUB 500                  :rem 69
67 REM                                             :rem 81
68 REM SET UP INITIAL SCREEN POSITIONS            :rem 117
69 REM                                             :rem 83
70 FG=31:OC=1:NC=2:Q=3:QQ=Q+1:GOSUB 800           :rem 28
80 NH=7:NV=7:C=NH+NV*40:W=32:SS=27:CS=2:TS=0
                                                   :rem 183
90 GOTO 290                                        :rem 61
97 REM                                             :rem 84
98 REM MAIN LOOP                                  :rem 180
99 REM                                             :rem 86
100 A=PEEK(653)AND3:B=PEEK(197)                   :rem 249
```

```
107 REM                                          :rem 124
108 REM TIME TO SCROLL?                           :rem 93
109 REM                                          :rem 126
110 IF VAL(TI$)>Q THEN GOSUB 600                 :rem 234
117 REM                                          :rem 125
118 REM ANIMATE THE SCREEN                        :rem 30
119 REM                                          :rem 127
120 K=-(K=0):POKE 53272,CB(K)                     :rem 10
187 REM                                          :rem 132
188 REM IF NO MOVEMENT, THEN REPEAT               :rem 56
189 REM                                          :rem 134
190 H=(A>1)-(A=1):V=(B=6)-(B=3):IF H=0 AND V=0 THE
    N 100                                         :rem 36
197 REM                                          :rem 133
198 REM MOVEMENT LOOP                             :rem 43
199 REM                                          :rem 135
200 NH=OH+H+39*((OH=38 AND H=1)-(OH=0 AND H=-1))
                                                 :rem 210
210 NV=OV+V+(OV=24 AND V=1)-(OV=1 AND V=-1):rem 74
217 REM                                          :rem 126
218 REM RESTORE VALUE AT OLD LOCATION            :rem 237
219 REM                                          :rem 128
220 POKE CM+C,OC:POKE SC+C,W:C=NH+NV*40           :rem 28
227 REM                                          :rem 127
228 REM COLLISION?                               :rem 107
229 REM                                          :rem 129
230 W=PEEK(SC+C):IF W<>32 THEN GOSUB 700         :rem 232
277 REM                                          :rem 132
278 REM POKE SHIP IN NEW LOCATION                :rem 194
279 REM                                          :rem 134
280 POKE CM+C,NC:POKE SC+C,FG:IF NC>9 THEN 900
                                                 :rem 67
290 OH=NH:OV=NV:GOTO 100                         :rem 201
397 REM                                          :rem 135
398 REM SET UP STRING OF RANDOM STARS            :rem 211
399 REM                                          :rem 137
400 T$="":FOR I=0 TO 254:T=32                    :rem 210
410 IF INT(RND(9)*7)<1 THEN T=92:IF INT(RND(9)*5)<
    1 THEN T=93                                   :rem 50
420 T$=T$+CHR$(T):NEXT                            :rem 13
430 M$=CHR$(19):FOR I=0 TO 24:M$=M$+CHR$(17):NEXT:
    RETURN                                       :rem 178
497 REM                                          :rem 136
498 REM CREATE RANDOM STARFIELD                  :rem 156
499 REM                                          :rem 138
500 POKE 53281,0:POKE 53280,0:PRINT "{WHT}{CLR}"
                                                 :rem 141
505 FOR I=CM TO CM+999:POKE I,1:NEXT              :rem 40
510 Q=100*RND(9)+20:Q1=6*RND(9)+2                :rem 254
```

```
520 FOR I=0 TO Q:X=1000*RND(9):N=28:GOSUB 540:NEXT
                                                :rem 75
530 FOR I=0 TO Q1:X=1000*RND(9):N=29:GOSUB 540:NEX
    T:RETURN                                   :rem 152
540 POKE SC+X,N:RETURN                         :rem 117
597 REM                                        :rem 137
598 REM SCROLL THE SCREEN                      :rem 250
599 REM                                        :rem 139
600 POKE SC+C,W:POKE CM+C,OC:T=1+INT(RND(9)*167)
                                               :rem 10
610 PRINT M$;MID$(T$,T,40*INT(QQ-Q)-1);:P=P-INT(7*
    Q):GOSUB 800                               :rem 74
617 REM                                        :rem 130
618 REM COLLISION?                             :rem 110
619 REM                                        :rem 132
620 OC=PEEK(CM+C):W=PEEK(SC+C):IF W<>32 THEN GOSUB
     700                                       :rem 104
627 REM                                        :rem 131
628 REM RESTORE STARSHIP TO PLACE               :rem 30
629 REM                                        :rem 133
630 POKE SC+C,FG:POKE CM+C,NC                  :rem 169
637 REM                                        :rem 132
638 REM DISPLAY SCORE                           :rem 23
639 REM                                        :rem 134
640 PRINT "{HOME}"TAB(18)STR$(P)" POINTS{3 SPACES}
    "                                          :rem 238
647 REM                                        :rem 133
648 REM TIME FOR A SPACE STATION?              :rem 170
649 REM                                        :rem 135
650 TS=TS+1:IF TS>10*(QQ-Q)THEN GOSUB 850      :rem 120
657 REM                                        :rem 134
658 REM TOO MANY SHIPS LOST?                   :rem 182
659 REM                                        :rem 136
660 IF NC>9 THEN 900                           :rem 251
690 RETURN                                     :rem 127
695 REM                                        :rem 136
696 REM COLLISION HANDLING ROUTINE             :rem 160
697 REM                                        :rem 138
698 REM SMALL STAR                              :rem 62
699 REM                                        :rem 140
700 IF W=28 THEN P=P-100:NC=NC+1:RETURN        :rem 253
707 REM                                        :rem 130
708 REM SPACE STATION                           :rem 17
709 REM                                        :rem 132
710 IF W=27 THEN NC=CS:W=30:RETURN               :rem 2
717 REM                                        :rem 131
718 REM USED SPACE STATION                      :rem 67
719 REM                                        :rem 133
720 IF W=30 THEN RETURN                         :rem 46
```

```
727 REM                                          :rem 132
728 REM NOVA!                                    :rem 218
729 REM                                          :rem 134
730 P=P+(8*NH)*INT(QQ-Q):W=28:RETURN             :rem 195
797 REM                                          :rem 139
798 REM RESET TIMER                              :rem 144
799 REM                                          :rem 141
800 TI$="000000":Q=99*(Q/100):RETURN             :rem 218
847 REM                                          :rem 135
848 REM GENERATE SPACE STATION                    :rem 97
849 REM                                          :rem 137
850 TS=0:T2=1001-INT(RND(9)*40):POKE SC+T2,SS:POKE
    CM+T2,CS:RETURN                              :rem 221
897 REM                                          :rem 140
898 REM ENDING ROUTINE                           :rem 104
899 REM                                          :rem 142
900 FOR I=0 TO 20:FOR X=0 TO 7:POKE CM+C,X:NEXT:NE
    XT                                           :rem 216
910 PRINT "{CLR}"P" POINTS":PRINT:PRINT:PRINT "THE
    END":END                                     :rem 209
1097 REM                                         :rem 181
1098 REM INITIALIZE CHARACTER SET                 :rem 33
1099 REM                                         :rem 183
1100 POKE 56334,PEEK(56334)AND 254:POKE 1,PEEK(1)A
    ND 251:POKE 657,0                             :rem 74
1110 FOR I=0 TO 1:RM=53248-CH(I)                 :rem 162
1120 FOR J=CH(I) TO CH(I)+511:POKE J,PEEK(J+RM):NE
    XT:NEXT                                       :rem 60
1130 FOR I=0 TO 1:FOR J=27 TO 31:FOR K=0 TO 7
                                                   :rem 1
1140 READ N:POKE CH(I)+8*J+K,N:NEXT:NEXT:NEXT
                                                 :rem 240
1150 POKE 1,PEEK(1)OR 4:POKE 56334,PEEK(56334)OR 1
                                                 :rem 182
1160 POKE 53272,CB(0):RETURN                      :rem 70
1297 REM                                         :rem 183
1298 REM CHARACTER DATA                           :rem 95
1299 REM                                         :rem 185
1300 DATA 24,12,14,12,48,112,48,24               :rem 117
1310 DATA 0,0,40,16,40,0,0,0                      :rem 50
1320 DATA 0,60,126,126,126,126,60,0              :rem 164
1330 DATA 0,32,116,249,159,46,4,0                 :rem 78
1340 DATA 211,137,153,255,153,24,24,36            :rem 72
1350 DATA 0,32,112,241,143,14,4,0                 :rem 56
1360 DATA 0,0,16,40,16,0,0,0                      :rem 58
1370 DATA 0,66,60,60,60,60,66,0                  :rem 233
1380 DATA 0,32,112,241,143,14,4,0                 :rem 59
1390 DATA 203,145,153,255,153,24,24,36            :rem 77
```

Star-Eater

Here's a simple gobble game. You are a hungry mouth, out to eat stars. You get points for eating them. The trouble is, each one you eat turns into a flickering fire—and if you accidentally burn yourself at a fire, you *lose* points. Also, every fire you touch turns into a boulder, 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*. You can move diagonally.

With what you already know about game programming, you should have little trouble figuring out exactly what's going on. Lines 5-55 are Initialization; lines 100-190 are the Main Loop; lines 200-290 are the Movement Loop; lines 300-350 are the End Routine; lines 600-620 are the Set Level Subroutine; and lines 700-730 are the Collision Routine. It's 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$), but when the player touches a fire, the score is *subtracted from* by an amount equal to TI$. That means that the farther you get in the game, the fewer points you get for each star you eat, and the more points you lose for each fire you touch.

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 9-2. Star-Eater

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
5 DIM CH(3),CL(3),CC(1),CB(1),CM(1):CH(Ø)=1Ø2:CH(1
  )=32:CH(2)=43:CH(3)=42                          :rem 217
1Ø CL(Ø)=Ø:CL(1)=9:CL(2)=8:CL(3)=1                :rem 27
15 SC=PEEK(648):CM=55296+256*(SC AND2):SC=SC*256:P
   OKE 53281,9:POKE 5328Ø,11                      :rem 117
2Ø PRINT "{BLK}{CLR}":FG=81:CF=4:M=Ø:GOSUB 6ØØ
                                                  :rem 59
25 FOR I=4Ø TO 999:X=INT(RND(1)*4):M=M-(X=3)
                                                  :rem 172
3Ø POKE SC+I,CH(X):POKE CM+I,CL(X):NEXT           :rem 7Ø
35 TI$="ØØØØØØ":OH=1Ø:OV=1Ø:FL=OH+OV*4Ø:OC=CL(1):W
   =CH(1)                                         :rem 169
4Ø POKE SC+FL,FG:POKE CM+FL,CF                    :rem 1Ø
45 CC(Ø)=12:CC(1)=14:CM=55296:FOR I=Ø TO 1:CM(I)=1
   Ø24*CC(I):NEXT                                 :rem 22Ø
5Ø FOR I=Ø TO 1:CM(I)=1Ø24*CC(I):CB(I)=(PEEK(53272
   )AND 24Ø)OR CC(I):NEXT                         :rem 139
55 GOSUB 11ØØ                                     :rem 172
1ØØ PRINT "{HOME}SCORE "STR$(P)" TIME "STR$(N-VAL(
    TI$))" "                                      :rem 68
11Ø A=PEEK(653)AND3:B=PEEK(197)                   :rem 25Ø
12Ø IF VAL(TI$)>N OR M=Ø THEN 3ØØ                 :rem 192
13Ø K=-(K=Ø):POKE 53272,CB(K)                     :rem 11
19Ø H=(A>1)-(A=1):V=(B=6)-(B=3):IF H=Ø AND V=Ø THE
    N 1ØØ                                         :rem 36
2ØØ NH=OH+H+4Ø*((OH=39 AND H=1)-(OH=Ø AND H=-1))
                                                  :rem 2Ø3
21Ø NV=OV+V+24*((OV=24 AND V=1)-(OV=1 AND V=-1))
                                                  :rem 43
22Ø POKE SC+FL,W:POKE CM+FL,OC:FL=NH+NV*4Ø:E=W
                                                  :rem 28
23Ø W=PEEK(SC+FL):OC=PEEK(CM+FL):IF W<>32 THEN GOS
    UB 7ØØ                                         :rem 3
28Ø POKE CM+FL,CF:POKE SC+FL,FG                   :rem 64
29Ø OH=NH:OV=NV:GOTO 1ØØ                          :rem 2Ø1
3ØØ PRINT "{CLR}":IF M=Ø THEN SH=N-VAL(TI$):P=P+1Ø
    ØØ*SH:GOTO 32Ø                                :rem 193
31Ø PRINT "TIME'S UP!":PRINT "STARS LEFT: "M:GOTO
    {SPACE}325                                    :rem 43
```

```
320 PRINT "YOU GOT ALL THE STARS!":PRINT "TIME LEF
    T: "N-VAL(TI$)                              :rem 196
325 PRINT "SCORE: "P                            :rem 113
330 PRINT"{3 DOWN}PLAY AGAIN? (Y OR N)"          :rem 8
340 B=PEEK(197):IF B=39 THEN END                :rem 231
345 IF B=25 THEN RUN                             :rem 53
350 GOTO 340                                    :rem 104
600 PRINT "{6 SPACES}STAR-EATER":PRINT "{2 DOWN}CH
    OOSE YOUR LEVEL:"                            :rem 46
605 PRINT "1 - EASY":PRINT "2 - HARD":PRINT "3 - S
    UPERHUMAN"                                  :rem 248
610 N=PEEK(197):IF N=64 THEN 610                :rem 189
620 N=400+120*(N=59)+250*(N=8):PRINT "{CLR}":RETUR
    N                                           :rem 161
700 IF W=CH(0) THEN 730                         :rem 143
710 IF W=CH(2) THEN P=P-VAL(TI$):W=CH(0):OC=CL(0):
    RETURN                                       :rem 4
720 P=P+INT(N-VAL(TI$)):W=CH(2):OC=CL(2):M=M-1:RET
    URN                                         :rem 205
730 NH=OH:NV=OV:FL=OH+OV*40:W=E:OC=PEEK(FL+CM):RET
    URN                                         :rem 187
1100 POKE 56334,PEEK(56334)AND 254:POKE 1,PEEK(1)A
     ND 251                                     :rem 227
1110 FOR I=0 TO 1:RM=53248-CM(I)                :rem 167
1120 FOR J=CM(I) TO CM(I)+511:POKE J,PEEK(J+RM):NE
     XT:NEXT                                     :rem 70
1130 FOR I=0 TO 1:J=816:GOSUB 1150:J=344:GOSUB 115
     0:J=336:GOSUB 1150                         :rem 210
1140 J=648:GOSUB 1150:NEXT:POKE 1,PEEK(1)OR 4:POKE
      56334,PEEK(56334)OR 1:RETURN               :rem 44
1150 FOR K=CM(I)+J TO CM(I)+J+7:READ SH:POKE K,SH:
     NEXT:RETURN                                :rem 120
1300 DATA 0,0,28,30,30,126,127,255              :rem 113
1310 DATA 0,0,16,106,38,28,0,0                  :rem 166
1320 DATA 0,8,28,54,28,8,0,0                     :rem 81
1330 DATA 0,0,24,126,231,24,0,0                 :rem 208
1340 DATA 0,0,28,30,30,126,127,255              :rem 117
1350 DATA 0,0,8,86,100,56,0,0                   :rem 121
1360 DATA 0,20,54,8,54,20,0,0                   :rem 118
1370 DATA 24,60,102,195,129,66,60,24            :rem 235
```

Collision Tracking

So far we have used only 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. 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 sprite collisions. One thing we haven't done yet is make a computer-controlled figure respond to other characters on the screen. In maze games like *Pac-Man*, the computer-controlled characters wander through a maze, making turns correctly and *not* blundering off through the walls of the maze. This is handled by having the computer read its own screen memory like a map. When computer-controlled figures collide with certain characters, they respond in predetermined ways.

In the game "Spark," we'll use a screen display we created in Chapter 3. The game consists of 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 100 to 190 PEEKs into screen memory in order to see

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 with character movement in BASIC games!

Program 9-3. Spark

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 DIM A$(6),S$(10):POKE 53281,0:POKE 53280,1:PRIN
   T "{CLR}{WHT}":GOSUB 500                    :rem 254
15 Z=4                                          :rem 49
20 FOR I=0 TO 9:PRINT S$(I)A$(0):NEXT I         :rem 91
30 SM=PEEK(648)*256:CM=55296                    :rem 49
35 FOR I=CM TO CM+999:POKE I,1:NEXT            :rem 246
40 OC=PEEK(CM):NC=10:OL=37:NL=77               :rem 157
50 DIM MV(4,4):GOSUB 700:FR=0:CH=4:P=300:POKE 657,
   128                                         :rem 185
60 Y=1:GOSUB 310:A=2:PL=6:GOSUB 200            :rem 250
100 IF VAL(TI$)>Z THEN GOSUB 300               :rem 239
110 POKE CM+NL-MV(CH,FR),OC:POKE CM+NL,NC      :rem 191
120 A=PEEK(653)AND3:B=PEEK(197):IF A>0 THEN GOSUB
    {SPACE}200                                 :rem 180
130 IF B<64 THEN GOSUB 250                      :rem 81
140 E=PEEK(SM+NL):GOSUB 400:CH=E               :rem 135
145 IF CH=4 AND YY<>0 THEN GOSUB 450           :rem 156
150 E=PEEK(SM+OL):GOSUB 400:FR=E:IF CH=FR THEN FR=
    4+(FR=4)+D                                 :rem 235
160 IF CH<4 THEN OL=NL                         :rem 194
170 IF CH<0 THEN 600                           :rem 227
180 NL=NL+MV(CH,FR):D=SGN(MV(CH,FR))=-1         :rem 74
190 YY=0:GOTO 100                              :rem 189
200 PH=A:IF A=3 THEN PH=0:GOTO 290             :rem 100
250 PRINT S$(PL)A$(0):YY=SGN(MV(CH,FR))         :rem 19
260 PL=PL+(B=6)-(B=3):PL=PL+10*(PL>9)-10*(PL<0)
                                                :rem 72
290 PRINT S$(PL)A$(PH):RETURN                  :rem 244
300 Y=INT(RND(9)*2):P=P-10:IF Z>1 THEN Z=Z-1
                                               :rem 228
310 PRINT "{HOME}{2 SPACES}TIME LEFT{3 SPACES}"STR
    $(P)" "                                    :rem 212
320 PRINT "{HOME}{DOWN}"A$(3+Y)S$(10)A$(6-Y)
                                               :rem 146
330 IF P=0 THEN 600                            :rem 167
```

```
340 TI$="000000":RETURN                                    :rem 19
400 E=E-128-73:IF E>2 THEN E=E-9:IF E>3 THEN E=E-7
                                                           :rem 109
410 RETURN                                                 :rem 117
450 IF YY=-1 THEN OL=CH-40*INT(CH/40)+720:RETURN
                                                           :rem 250
460 OL=CH-40*INT(CH/40)+40:RETURN                          :rem 187
500 FOR I=0 TO 6:A$(I)=CHR$(18):NEXT I                     :rem 254
520 FOR X=1 TO 38:A$(0)=A$(0)+CHR$(221):NEXT
                                                           :rem 32
530 FOR I=1 TO 5 STEP 4:FOR X=1 TO 19:A$(I)=A$(I)+
    CHR$(106)+CHR$(107):NEXT:NEXT                          :rem 2
540 FOR I=2 TO 6 STEP 4:A$(I)=CHR$(221):FOR X=1 TO
     18                                                    :rem 209
545 A$(I)=A$(I)+CHR$(106)+CHR$(107):NEXT:A$(I)=A$(
    I)+CHR$(221):NEXT                                      :rem 123
550 FOR I=0 TO 2:A$(I)=A$(I)+CHR$(13)+CHR$(18):NEX
    T                                                      :rem 193
560 FOR X=1 TO 38:A$(0)=A$(0)+CHR$(221):NEXT
                                                           :rem 36
570 FOR I=1 TO 3 STEP 2:FOR X=1 TO 19:A$(I)=A$(I)+
    CHR$(117)+CHR$(105):NEXT:NEXT                          :rem 2
580 FOR I=2 TO 4 STEP 2:A$(I)=A$(I)+CHR$(221):FOR
    {SPACE}X=1 TO 18                                       :rem 251
585 A$(I)=A$(I)+CHR$(117)+CHR$(105):NEXT:A$(I)=A$(
    I)+CHR$(221):NEXT                                      :rem 127
590 S$(0)=CHR$(18)+"{HOME}{2 DOWN}":FOR I=1 TO 10:
    S$(I)=S$(I-1)+"{2 DOWN}":NEXT:RETURN                   :rem 41
600 PRINT S$(10)TAB(17)"{DOWN}THE END":GOTO 600
                                                           :rem 242
700 FOR I=0 TO 4:FOR X=0 TO 4:READ MV(I,X):NEXT:NE
    XT:RETURN                                              :rem 235
710 DATA 0,-1,-1,40,-1,1,0,-40,1,1,-1,-40,0,-1,-1
                                                           :rem 247
720 DATA 40,1,1,0,1,40,-40,-40,40,0                        :rem 128
```

Checking for Collisions—Sprites

Characters are not the only thing you can place on the 64's screen. In many games, you'll want to include sprites. Checking for sprite collisions is even easier than testing for character collisions. This is because the 64 has a built-in system for detecting when a sprite bumps into another sprite, or even into a character.

Two memory locations are set aside for sprite collision detection in your computer. They are addresses 53278 and 53279, usually referred to as registers V+30 and V+31, when V=53248. Each bit in each of the two registers is assigned to a

particular sprite. For instance, bit 3 in each of these registers represents the collision status of sprite 3.

Usually, all bits in both registers are set to 0, or *off*. When a sprite bumps into another sprite, its bit in register V + 30 is set to 1. If that sprite collides with a screen character, its bit in the other register, V + 31, is set to 1.

By PEEKing at these registers' locations, you can find out what sprites, if any, have collided. PEEKing(V + 30), for example, might give you a value of 72. That means that sprites 3 and 6 have bumped into each other. When a sprite's bit is turned *on* as it collides, the bit's value is placed in the register. Bit 3's value is 8, while bit 6's value is 64. 8 + 64 = 72.

Collisions between sprites. So you can easily find out which sprites have collided simply by PEEKing location V + 30. Once you've PEEKed this location, all bits are automatically reset to 0, or turned *off*.

If your game has only two sprites, all you need do is PEEK(V + 30) to see if its value is greater than 0. If it is, the sprites, no matter what their numbers, have collided. When using more than two sprites, however, you'll have to use the logical AND to mask the unwanted sprites.

```
IF PEEK(V+30) AND 16=16 THEN GOSUB 300
```

This would check to see if sprite 4 (bit value of 16) was involved in a collision.

This program is a variation of a machine language joystick-controlled sprite routine from Chapter 8. It's not quite a game, but with some work on your part, it could be.

Here's how the program works.

Line	Function
10	Set screen background color, establish variable V, and set X and Y coordinate positions for both sprites. *X* and *Y* are sprite 0's starting location, while *X2* and *Y2* are the coordinates for sprite 1.
15	READ DATA for machine language movement subroutine.
20	READ DATA for sprite creation. Since both sprites use identical DATA statements, information is only POKEd into addresses 12288 + X. Both sprites will take their information from this area.

30	Set sprite pointers so that both sprites look to 12288 for DATA information.
40	Enable both sprite 0 and sprite 1.
50	Set color of sprite 0 to light red (V+39,10) and sprite 1 to yellow (V+40,7).
60-70	POKE in location of both sprites.
80	Access the machine language subroutine for faster joystick movement.
90-100	Move sprite 1 vertically down the screen, using wraparound.
110	PRINT the value read from PEEK(V+30) at the top-left corner of the screen. As you move the light red sprite across the screen, this number will change to a 3 when the sprites collide.
150	End of main loop.
990-1010	DATA statements for machine language subroutine.
60000-60020	DATA statements for sprite creation.

Program 9-4. Sprite-to-Sprite Collisions

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 POKE 53281,0:PRINT"{CLR}":V=53248:X=120:Y=120:X
   2=150:Y2=50                                    :rem 35
15 FOR X=828 TO 949:READ A:POKE X,A:NEXT          :rem 11
20 FOR X=0 TO 63:READ A:POKE 12288+X,A:NEXT
                                                  :rem 136
30 POKE 2040,192:POKE 2041,192                     :rem 24
40 POKE V+21,3                                    :rem 214
50 POKE V+39,10:POKE V+40,7                       :rem 191
60 POKE V+0,X:POKE V+2,X2                         :rem 156
70 POKE V+1,Y:POKE V+3,Y2                         :rem 161
80 SYS(937)                                        :rem 91
90 Y2=Y2+3:IF Y2>255 THEN Y2=0                    :rem 111
100 POKE V+3,Y2                                    :rem 43
110 PRINT"{HOME}"PEEK(V+30)                       :rem 208
150 GOTO 80                                        :rem 55
990 REM DATA FOR JS ROUTINE                        :rem 74
995 DATA 173,1,220,74,176,3,206,1,208,74,176,3,238
    ,1,208,74,176,42,173                          :rem 134
996 DATA 0,208,208,31,173,16,208,41,1,208,16,173,1
    6,208,9,1,141,16,208                          :rem 114
997 DATA 169,80,141,0,208,96,234,234,173,16,208,41
    ,254,141,16,208,206                           :rem 87
```

```
998 DATA 0,208,96,234,234,74,176,32,238,0,208,240,
    30,169,80,205,0,208                          :rem 82
999 DATA 208,20,173,16,208,41,1,240,13,173,16,208,
    41,254,141,16,208,169                        :rem 172
1000 DATA 0,141,0,208,96,234,234,173,16,208,9,1,14
     1,16,208,96,234,234                         :rem 54
1010 DATA 32,60,3,32,60,3,32,60,3,32,60,3,96
                                                 :rem 83
59999 REM SPRITE DATA                            :rem 238
60000 DATA0,0,0,0,0,0,7,255,224,15,255,240,25,36,1
      52,31,255,248,15,255                       :rem 127
60010 DATA 240,7,255,224,0,24,0,32,189,4,49,255,14
      0,43,255,212,36,60,36,40,24                :rem 187
60020 DATA 20,48,36,12,32,90,4,0,189,0,1,126,128,3
      ,0,192,0,0,0,0,0,0,36,40,24                :rem 135
```

Notice the number changing from 0 (indicating no collision) to 3 (indicating a collision between sprite 0 and sprite 1) as the sprites bump into each other. You can keep the 3 displayed by keeping in contact with the computer-controlled sprite.

Even though this was the fast version of joystick movement, note how much it slowed down with the addition of only a few lines in the main loop. Keep this in mind when you decide how complicated to make your own game's main loop.

Making this program display a score is only a matter of a few changes.

Insert these changes and additions to Program 9-5:

```
110 PRINT"{HOME}"PEEK(V+30);"{9 SPACES}SCORE="SC
                                                 :rem 158
120 IF PEEK(V+30)=3 THEN SC=SC+1                 :rem 224
```

You could just as easily subtract points as add them.

Collisions between sprites and characters. The second collision register, V+31, is reserved for collisions between sprites and other screen characters. The value stored in the register will tell you which sprite was involved in the collision, but it will not tell you which screen character was bumped into. Still, this gives you a number of uses for this register.

A sprite could collide with character missiles, for instance. It wouldn't be important which character the sprite collided with, only that it was hit. To test for a particular sprite's collision with screen characters, you again need to mask the other sprites with logical AND. For example:

IF PEEK(V+31) AND 8=8 THEN GOSUB 300

would test to see if sprite 3 was involved in a collision with a screen character.

These program changes show both sprite-to-sprite and sprite-to-character collisions. Add them to Program 9-4.

Program 9-5. Sprite-to-Character Collisions

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
7 L=1Ø24:C=42:CM=54272                          :rem 178
11Ø PRINT "{HOME}{WHT}SPRITE TO SPRITE="PEEK(V+3Ø)
    ; "(V+3Ø)                                   :rem 245
115 PRINT "{2 DOWN}SPRITE TO CHARACTER= "PEEK(V+31
    ); "(V+31)                                  :rem 188
12Ø L=L+1:IF L>2Ø23 THEN L=1Ø24                  :rem 93
13Ø POKE L,C:POKE L+CM,5:POKE L-1,32            :rem 243
```

Both register values will print on the screen with this program. Although there are two sprites on the screen, you control only the light red one. A moving character zips across the screen as well. The register values change each time the player-controlled sprite collides with the other sprite, the moving character, or the text at the top of the screen. (Remember that the text is really just standard characters.)

Different kinds of collisions can occur on the screen at the same time with the 64. Keeping track of these collisions is relatively easy, especially when you're using sprites.

Invisible Characters

Remember when you created multicolored characters? Not only are they more attractive on the screen, but they also have a very useful purpose when you want to detect only certain screen characters.

A collision between a sprite and a character causes bits to be set in sprite register V + 31. This makes it easy to detect collisions between important characters on the screen. However, if you want to have some characters that would set off the collision register and others that didn't, you may think that it's impossible. It's not.

A game like *Pac-Man*, for example, has some characters that set registers, and others that don't. You certainly wouldn't want the ghost-like sprites to set bits in a collision register when moving over the dots in the maze. You *would* want these same sprites to detect the walls of the maze, though.

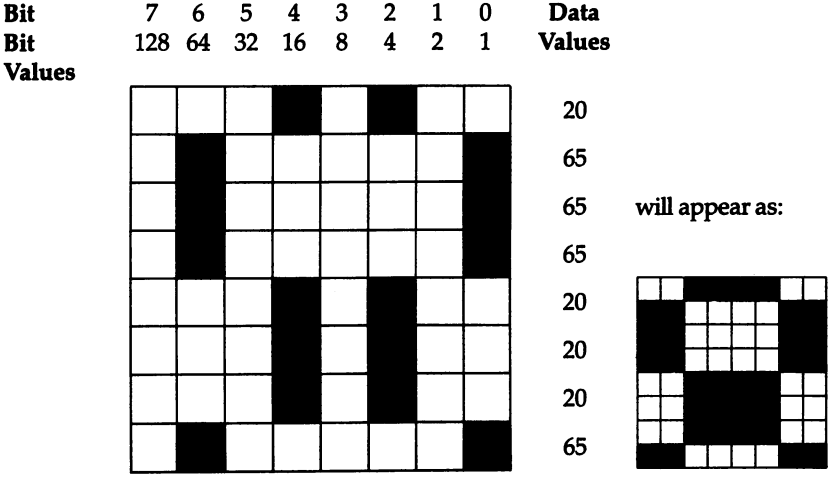
You can use multicolored characters to distinguish between characters that will set bits in the V+31 register and those that do not. Using this technique, you can place characters on the screen that will be ignored by the sprite detection register.

You'll recall that multicolored characters are made up of bits paired in four possible combinations. These combinations are: 00, 01, 10, and 11. Each of these bit-pair combinations produce a different color. Characters completely made up of 00 and 01 paired bits will not be detected by the sprite collision register.

Creating background characters on the screen becomes simple. As long as the characters are multicolored and made up only of 00 and 01 bit-pairs, the sprites moving across the screen will not detect a collision. Other characters, perhaps multicolored characters made up of 10 and 11 bit-pairs, would cause the register to be set.

This program creates a multicolored character made up only of 00 and 01 bit-pairs, moves it about the screen, and allows you to see a sprite bump into it, but *not* set the collision register. The multicolored character is drawn in Figure 9-1, showing you how it was designed using only those two bit-pairs.

Figure 9-1. Multicolor Custom Character



Program 9-6. Invisible Character Detection

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
8 GOTO 500                                              :rem 6
10 POKE 53281,0:PRINT"{CLR}":V=53248:X=120:Y=120:X
   2=150:Y2=50                                          :rem 35
11 POKE 53282,15:POKE 53283,2:POKE 53270,PEEK(5327
   0) OR 16                                             :rem 8
15 FOR X=828 TO 949:READ A:POKE X,A:NEXT   :rem 11
20 FOR X=0 TO 63:READ A:POKE 12608+X,A:NEXT
                                                        :rem 132
25 L=1520:C=28:CM=54272                                 :rem 231
30 POKE 2040,197:POKE 2041,197                          :rem 34
40 POKE V+21,3                                          :rem 214
50 POKE V+39,10:POKE V+40,7                             :rem 191
60 POKE V+0,X:POKE V+2,X2                               :rem 156
70 POKE V+1,Y:POKE V+3,Y2                               :rem 161
80 SYS(937)                                             :rem 91
90 Y2=Y2+3:IF Y2>255 THEN Y2=0                          :rem 111
100 POKE V+3,Y2                                         :rem 43
110 PRINT"{HOME}{2 SPACE}SPRITE TO SPRITE={2 SPACES}"P
    EEK (V+30);" (V+30)                                 :rem 133
115 PRINT"{2 DOWN}SPRITE TO CHARACTER={2 SPACES}"P
    EEK (V+31); " (V+31)                                :rem 188
120 L=L+1:IF L>2023 THEN L=1024                         :rem 93
130 POKE L+CM,15:POKE L,C:POKE L-1,32                   :rem 36
150 GOTO 80                                             :rem 55
500 PRINT CHR$(142)                                     :rem 11
510 POKE 52,48:POKE 56,48:CLR                           :rem 75
520 POKE 56334,PEEK(56334) AND 254                      :rem 224
530 POKE 1,PEEK(1) AND 251                              :rem 54
540 FOR I=0 TO 511:POKE I+12288,PEEK(I+53248):NEXT
                                                        :rem 231
550 POKE 1,PEEK(1) OR 4                                 :rem 162
560 POKE 56334,PEEK(56334) OR 1                         :rem 72
570 POKE 53272,(PEEK(53272) AND 240)+12   :rem 187
580 FOR X=12512 TO 12519:READ A:POKE X,A:NEXT
                                                        :rem 247
590 GOTO 10                                             :rem 56
595 REM DATA FOR CUSTOM CHARACTER                       :rem 240
600 DATA 20,65,65,65,20,20,20,65                        :rem 24
990 REM DATA FOR JS ROUTINE                             :rem 74
995 DATA 173,1,220,74,176,3,206,1,208,74,176,3,238
    ,1,208,74,176,42,173                                :rem 134
996 DATA 0,208,208,31,173,16,208,41,1,208,16,173,1
    6,208,9,1,141,16,208                                :rem 114
997 DATA 169,80,141,0,208,96,234,234,173,16,208,41
    ,254,141,16,208,206                                 :rem 87
```

```
998 DATA 0,208,96,234,234,74,176,32,238,0,208,240,
    30,169,80,205,0,208                          :rem 82
999 DATA 208,20,173,16,208,41,1,240,13,173,16,208,
    41,254,141,16,208,169                        :rem 172
1000 DATA 0,141,0,208,96,234,234,173,16,208,9,1,14
     1,16,208,96,234,234                         :rem 54
1010 DATA 32,60,3,32,60,3,32,60,3,32,60,3,96
                                                 :rem 83
59999 REM SPRITE DATA                            :rem 238
60000 DATA0,0,0,0,0,0,7,255,224,15,255,240,25,36,1
      52,31,255,248,15,255                       :rem 127
60010 DATA 240,7,255,224,0,24,0,32,189,4,49,255,14
      0,43,255,212,36,60,36,40,24                :rem 187
60020 DATA 20,48,36,12,32,90,4,0,189,0,1,126,128,3
      ,0,192,0,0,0,0,0,0,36,40,24                :rem 135
```

Only the routine from line 500 to line 600 is new. This sets up the custom multicolored character, and is completely explained in Chapter 4, Custom Characters.

The first 64 standard characters were copied and placed in memory locations starting at 12288. The custom character was located where the pound sign is normally placed.

As you move the light red sprite across the screen, watch the values change at the top. When the sprites collide, a value of 3 is still placed in register V+30. Each sprite also sets bits in the V+31 register as it collides with the text. Sprite 0 sets a value of 1 in the register, while the computer-controlled sprite, sprite 1, sets a value of 2. But neither sprite detects a collision with the multicolored character.

10 Sounds and Music

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 importantly, 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 you 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 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.

The SID Chip

Creating sounds on the Commodore 64 is very different from other computers which use controllable oscillators to produce sound. The VIC-20, for example, creates sounds quite simply. The 64, however, has a built-in music and sound synthesizer. This allows you to create much more complicated sounds, but the process of creating those sounds is also more complex.

The 64's SID (Sound Interface Device) chip creates sounds and music by using 29 different memory locations, or registers. These registers can be divided into five sections. These sections and the registers in each are:

Section	Registers	Control
Section 1	0-6	Voice Number 1
Section 2	7-13	Voice Number 2
Section 3	14-20	Voice Number 3
Section 4	21-24	Filter Controls
Section 5	25-28	Read-Only Controls

The first four sections are sometimes referred to as *write-only* registers. You can POKE, or *write* new values to these locations, but you can't PEEK there to find out the values in a particular register. Because they are *write-only*, you can't use the logical ANDs and ORs to turn certain bits *on* or *off*. You can only POKE a number directly into a register.

The last section of registers is *read-only*. It's just the opposite: you can PEEK, but you can't POKE. The four registers in this section are useful as random number generators when you create sound effects. For instance, register 27 could be used to generate random numbers to represent a static or white noise sound.

The first three sections, registers 0-20, are the ones you'll find the most use for when creating sound effects for your games. Each section is one *voice* of the 64. They can be played

separately or they can be combined. Each voice is capable of creating any of the notes within the human auditory range.

Getting Ready

Using the SID chip in the 64 to create game sound effects may seem complex to you, especially if you're just beginning. Don't worry. As long as you follow the steps outlined, you'll quickly be creating your own game sounds and music.

To create a single sound or note, you'll have to do five things. Each step is explained below.

1. Frequency. You have to select a frequency for your sound or note. The first two registers in each voice control this. These registers combine their values to form one number, which is the frequency of the note. This is not important unless you want specific notes played, as when you're creating a tune. It is the high frequency register, register 1 in voice 1, for instance, which has the most effect on the pitch of a note. Most of the time you'll use register 1 to change a note's pitch. Just remember that a high pitch is soprano, a low pitch is bass, and you'll be able to control a note's frequency.

2. Wave form. You also need to select a wave form for the note you're creating. This is chosen by POKEing the appropriate value into the fifth register of each voice. You have four wave forms to choose from. The triangle wave form produces a flute-like sound, while the sawtooth wave form creates a more twangy, bright sound. The pulse wave form can produce a variety of sounds because other registers in each voice control its frequency separately. The noise wave form creates "white noise," ranging from a hiss to a rumble.

3. Attack, Delay, Sustain, and Release (ADSR). The sixth register in each of the three voices controls this. The ADSR is somewhat complicated, and will be explained in more detail later in the chapter.

4. Volume. Register 24 controls the volume, which has a range from 0 to 15. Zero is *off*, while 15 is the loudest volume. If this is not set, you won't hear anything when the sound effect runs.

5. Gate bit. Unless the gate bit is turned on, the sound will not play. This bit is in the same register as the wave form values, so it's often set at the same time.

The SID Register

It's difficult trying to memorize the registers, what each does, and which bit controls each register function. Refer to the SID Register Table whenever you need this information.

Table 10-1. SID Register

Register	128 7	64 6	32 5	16 4	8 3	4 2	2 1	1 0	Bit Value Bit
0	FL 7	FL 6	FL 5	FL 4	FL 3	FL 2	FL 1	FL 0	Note Frequency (Low Half)
1	FH 15	FH 14	FH 13	FH 12	FH 11	FH 10	FH 9	FH 8	Note Frequency (High Half)
2	PWL 7	PWL 6	PWL 5	PWL 4	PWL 3	PWL 2	PWL 1	PWL 0	Pulse Width (Low)
3 Voice 1	Not Used	Not Used	Not Used	Not Used	PWH 11	PWH 10	PWH 9	PWH 8	Pulse Width (High)
4	Noise Waveform	Pulse Waveform	Sawtooth Waveform	Triangle Waveform	Test Bit	Ring Mod. on/off	Sync. on/off	Gate Bit	Misc. Controls
5	Attack 3	Attack 2	Attack 1	Attack 0	Decay 3	Decay 2	Decay 1	Decay 0	Attack & Decay
6	Sustain 3	Sustain 2	Sustain 1	Sustain 0	Release 3	Release 2	Release 1	Release 0	Sustain & Release
7	FL 7	FL 6	FL 5	FL 4	FL 3	FL 2	FL 1	FL 0	Note Frequency (Low)
8	FH 15	FH 14	FH 13	FH 12	FH 11	FH 10	FH 9	FH 8	Note Frequency (High)
9	PWL 7	PWL 6	PWL 5	PWL 4	PWL 3	PWL 2	PWL 1	PWL 0	Pulse Width (Low)
10 Voice 2	Not Used	Not Used	Not Used	Not Used	PWH 11	PWH 10	PWH 9	PWH 8	Pulse Width (High)
11	Noise Waveform	Pulse Waveform	Sawtooth Waveform	Triangle Waveform	Test Bit	Ring Mod. on/off	Sync. on/off	Gate Bit	Misc. Controls
12	Attack 3	Attack 2	Attack 1	Attack 0	Decay 3	Decay 2	Decay 1	Decay 0	Attack & Decay
13	Sustain 3	Sustain 2	Sustain 1	Sustain 0	Release 3	Release 2	Release 1	Release 0	Sustain & Release

Register	128 7	64 6	32 5	16 4	8 3	4 2	2 1	1 0	Bit Value Bit
14	FL 7	FL 6	FL 5	FL 4	FL 3	FL 2	FL 1	FL 0	Note Frequency (Low)
15	FH 15	FH 14	FH 13	FH 12	FH 11	FH 10	FH 9	FH 8	Note Frequency (High)
16	PWL 7	PWL 6	PWL 5	PWL 4	PWL 3	PWL 2	PWL 1	PWL 0	Pulse Width (Low)
17 Voice 3	Not Used	Not Used	Not Used	Not Used	PWH 11	PWH 10	PWH 9	PWH 8	Pulse Width (High)
18	Noise Waveform	Pulse Waveform	Sawtooth Waveform	Triangle Waveform	Test Bit	Ring Mod. on/off	Sync. on/off	Gate Bit	Misc. Controls
19	Attack 3	Attack 2	Attack 1	Attack 0	Decay 3	Decay 2	Decay 1	Decay 0	Attack & Decay
20	Sustain 3	Sustain 2	Sustain 1	Sustain 0	Release 3	Release 2	Release 1	Release 0	Sustain & Release
21	Not Used	Not Used	Not Used	Not Used	Not Used	FFL 2	FFL 1	FFL 0	Filter Frequency (Low)
22	FFH 10	FFH 9	FFH 8	FFH 7	FFH 6	FFH 5	FFH 4	FFH 3	Filter Frequency (High)
23 Filters, etc.	Resonance 3	Resonance 2	Resonance 1	Resonance 0	External Filter	Filter Voice 3	Filter Voice 2	Filter Voice 1	Resonance & Filter Controls
24	Turn off Voice 3	High Pass Filter	Band Pass Filter	Low Pass Filter	Volume 3	Volume 2	Volume 1	Volume 0	Filter Type & Volume Control
25	PX 7	PX 6	PX 5	PX 4	PX 3	PX 2	PX 1	PX 0	Potentiometer X Read Bits
26 Misc.	PY 7	PY 6	PY 5	PY 4	PY 3	PY 2	PY 1	PY 0	Potentiometer Y Read Bits
27	03 7	03 6	03 5	03 4	03 3	03 2	03 1	03 0	Oscillator 3 Read Bits
28	EG3 7	EG3 6	EG3 5	EG3 4	EG3 3	EG3 2	EG3 1	EG3 0	Envelope Generator 3 Read Bits

Making a Simple Sound

You're ready to create your first note on the 64. Look at this program, which plays a high note and then turns it off, to see how each step was followed. Refer to the SID Register Table to check the POKEs used.

Line	Function
10	Establish variable *S* equal to register 0. Beginning memory location of SID registers is 54272.
20	Set the frequency of the note.

30 Set ADSR. Bit 3 in register S + 5 is set by POKEing in a value of 8. This sets the Decay to level 8. The Sustain is set to level 15 and the Release is set to level 8 by POKEing 248 into register S + 6 (248 = 16 * 15 + 8).
40 Turn on volume and set it to high.
50 Set wave form as the triangle wave form. Gate bit is also turned on. POKEing 16 turns on bit 4 (triangle wave form) and 1 is added to the value to turn on bit 0 (Gate bit).
60 Delay loop so that sound will play longer.
70 Set wave form again, this time with the gate bit turned off.
80 Another delay loop.
90 Set volume level to 0.

Not all of the SID register controls were used in creating this note, but it's a beginning.

Program 10-1. One Note

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 S=54272                                :rem 245
20 POKE S,150:POKE S+1,200                :rem 124
30 POKE S+5,8:POKE S+6,248                :rem 144
40 POKE S+24,15                             :rem 9
50 POKE S+4,17                            :rem 218
60 FOR T=0 TO 1000:NEXT                   :rem 235
70 POKE S+4,16                            :rem 219
80 FOR T=0 TO 1000:NEXT                   :rem 237
90 POKE S+24,0                            :rem 216
```

Attack, Decay, Sustain, and Release (ADSR)

Don't let the title frighten you. ADSR is actually little more than an expanded volume control. Although register 24 turns the volume on and off, as well as sets its level, ADSR is something each sound effect routine must have.

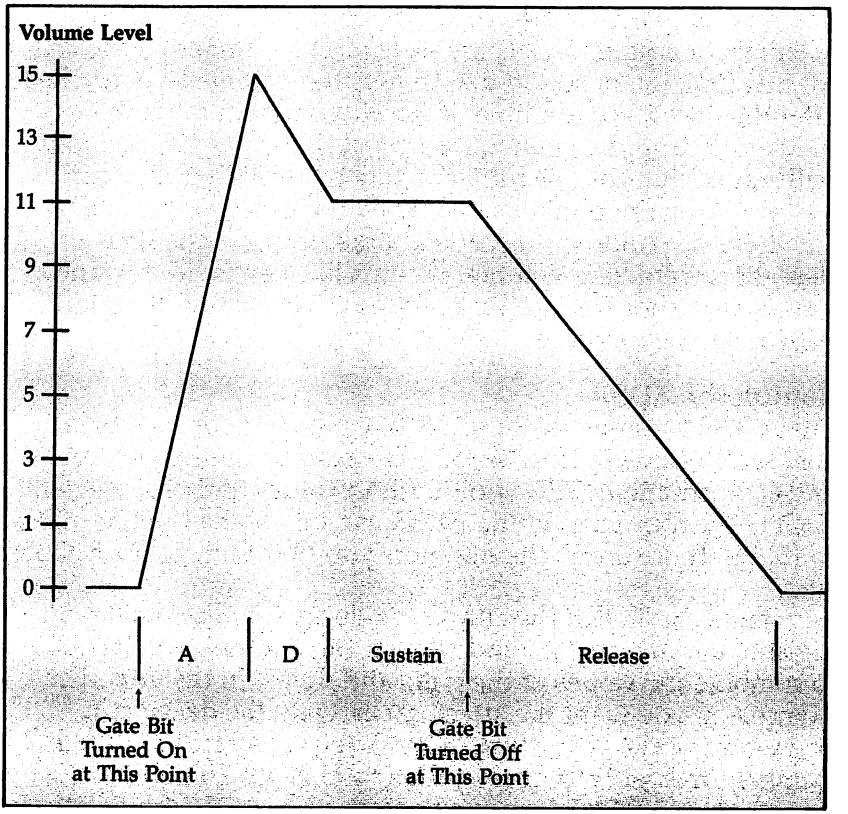
Think of what an explosion sounds like. It begins quite loud, and then dies down quickly. You might think of it as starting at full volume, keeping that level for a second or so, and then falling off rapidly. The entire sound might last only a few seconds. We can call this process of getting loud, then eventually getting softer, as the ADSR of a note. Sometimes it's

called the sound envelope.

Each natural and artificial sound has its own peculiar envelope, or ADSR. It's really what makes one sound different from another. By adjusting the ADSR of a sound you can imitate many natural sounds, and also produce completely new sounds.

ADSR breaks every note into four parts. By altering one or more of these parts, you control the note's sound. Take a look at Figure 10-1, ADSR Envelope.

Figure 10-1. ADSR Envelope



Notes: 1. Sustain can be maintained as long as desired, i.e. indefinitely.
2. Volume level of sustained note determined by sustain setting.
3. Attack, Decay and Release relate time elapsed, whereas Sustain is an actual loudness or volume setting.

Attack. As the sound begins, the volume rises rapidly. This section of the envelope is called the *attack*. When you create sound effects, you can set the attack to last anywhere from two milliseconds to eight seconds. Sounds like bangs, explosions, taps, or peeps have a short attack. An attack of 0 means the sound starts at full volume.

Decay. After this initial attack, the volume decreases during the *decay* section of the envelope. This decay of volume lasts a specified amount of time before the next level is reached, the *sustain*. In this way decay is like attack: it is a function of time expired. The number you select for this function determines how long the decay process will take.

Sustain. After the decrease in volume, the sound will stabilize for a time. This is the *sustain*. The number you select for this function of sound determines the volume level reached after the decay. For example, if the sustain is set at 12, the decay will drop the volume from level 15 to level 12. The note will stay at this volume until the gate bit is cleared. You can sustain a note indefinitely, simply by not clearing the gate bit.

Release. Another function of time, *release* is the final descent toward zero volume. Sounds like beeps, for instance, have a very short release time. Others, like a gong-like sound, have long release times.

Setting the ADSR

The ADSR parameters are set by two registers for each of the computer's three voices. Voice 1, for example, uses registers 5 and 6. To set the ADSR, you POKE in values to those registers. Each of the four parts of the ADSR can be set at a level from 0 to 15. But those aren't the numbers you POKE in. Look at Table 10-2.

Since you POKE the attack and decay into the same register, and the sustain and release into another, you'll need to combine the values of the pairs and POKE in the total. For example, if you want the attack set at 3 and the decay at 12, you would add 48 (the number for level 3 of attack) to 12 (the number for level 12 of decay), for a total of 60. That's the number you'd POKE into register 5 to set voice 1 with an attack of 3 and decay of 12.

If you take Program 10-1, One Note, and alter the ADSR values, you can change the sound of the note. Line 30 held the ADSR settings. You could change it to:

```
30 POKE S+5,204:POKE S+6,255
```

The attack was set at 12 (192) and the decay at 12 (12). The S+5 register was then POKEd with 204 (192+12). Sustain and release were both set to maximum (240 and 15, respectively), so 255 was POKEd into register S+6. Use your own ADSR values to experiment with the envelope parameters.

Table 10-2. ADSR Values and POKEs

Value	Attack—Decay (register 5, 12, or 19)		Sustain—Release (register 6, 13, or 20)	
0	0	0	0	0
1	16	1	16	1
2	32	2	32	2
3	48	3	48	3
4	64	4	64	4
5	80	5	80	5
6	96	6	96	6
7	112	7	112	7
8	128	8	128	8
9	144	9	144	9
10	160	10	160	10
11	176	11	176	11
12	192	12	192	12
13	208	13	208	13
14	224	14	224	14
15	240	15	240	15

Dynamic Changes

Using the SID chip, you can create single notes, and control its ADSR. You could certainly use single notes in a game program. If a character dashes across the screen, for instance, you might want to have a simple beeping sound playing at the same time. This can be easily done with single notes. You would set a short attack, decay, and release, and set the sustain to maximum. Let's modify that first sound effect program to see an example. Adding lines and making changes to Program 10-1 will create the following new program.

Program 10-2. Beeps

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 S=54272                                  :rem 245
20 POKE S,150:POKE S+1,200                  :rem 124
30 POKE S+5,0:POKE S+6,240                  :rem 128
40 POKE S+24,15                               :rem 9
50 POKE S+4,17                              :rem 218
60 FOR T=0 TO 200:NEXT                      :rem 188
70 POKE S+4,16                              :rem 219
80 FOR T=0 TO 200:NEXT                      :rem 190
90 GOTO 50                                    :rem 7
```

All we've done is turn the sound on and off every one-fifth of a second. You can vary the time each beep takes by changing the 200 in lines 60 and 80.

You just created a sound effect with *dynamic change*. Instead of simply turning the note on and letting it continue, you alternated the note with periods of silence. This is the key to sound effects. Change something. Since the SID chip has many parameters, you have numerous places to practice this technique of dynamic change.

Instead of using silence to change a note, let's change the pitch, or frequency. Altering the original version of Program 10-1 changes the sound effect to this:

Program 10-3. Frequency Change

```
10 S=54272:P=10                             :rem 29
20 POKE S,150:POKE S+1,P                    :rem 58
30 POKE S+5,0:POKE S+6,240                  :rem 128
40 POKE S+24,15                               :rem 9
50 POKE S+4,17                              :rem 218
60 FOR P=10 TO 250 STEP 2                   :rem 227
70 POKE S+1,P:NEXT                            :rem 58
80 FOR T=0 TO 1000:NEXT                     :rem 237
90 GOTO 60                                    :rem 8
```

What you've done is set the high frequency register to equal P, which changes in line 60. The sound will continue until you press the RUN/STOP and RESTORE keys. Quite a different sound, isn't it?

You can change the sound effect even more by experimenting with some of the parameters. For example, to increase the pitch jump each time through the FOR-NEXT loop in line

60, change it to:

```
60 FOR P=10 TO 250 STEP 6                    :rem 231
```

Or you can reduce the range of the pitch change by altering the values in line 60's FOR-NEXT loop. Try:

```
60 FOR P=10 TO 60 STEP 2                     :rem 178
```

or

```
60 FOR P=180 TO 250 STEP 5                    :rem 30
```

You can even reverse the pitch scale by changing line 60 to:

```
60 FOR P=250 TO 10 STEP -2                    :rem 16
```

Or you can create another line to make the sound first rise in pitch, then lower. Add these lines to the original Program 10-3:

```
75 FOR P=250 TO 10 STEP -2                    :rem 22
76 POKE S+1,P:NEXT                            :rem 64
```

Shortening the delay loop also creates a new sound. Try:

```
80 FOR T=0 TO 50:NEXT                        :rem 145
```

As you can see, changing the FOR-NEXT loop and the STEP instruction creates the most dynamic changes. By altering just a few things, you can make the computer play totally different sounds.

Random Notes

Register 27 in the SID chip can generate random numbers from 0 to 255 at a very rapid pace. By selecting the noise wave form for voice 3 (register 18, POKE 128), you can create these numbers. PEEKing register 27 lets you read the values there, which can then be POKEd into the frequency register of another voice to produce a series of random notes. It's useful and very easy to do.

Program 10-4. Random Notes

```
10 S=54272                                   :rem 245
20 POKE S+18,128:POKE S,75                   :rem 147
30 POKE S+5,0:POKE S+6,240                   :rem 128
40 POKE S+14,12:POKE S+15,250                 :rem 21
60 POKE S+24,207                              :rem 62
70 FOR L=0 TO 25:POKE S+4,17:POKE S+1,PEEK(S+27)
                                              :rem 97
```

```
75 FOR T=Ø TO 1ØØ:NEXT:NEXT:POKE S+4,Ø     :rem 177
8Ø FOR T=Ø TO 5ØØ:NEXT:GOTO 7Ø             :rem 155
```

The important lines in this program are 20 and 70. Line 20 sets the noise wave form of voice 3 (POKE S+18, 128) so that register 27 will generate random numbers. Line 70 sets up a FOR-NEXT loop that plays 26 random notes, the pitch of each determined by PEEKing register 27 (S+27). These random values are then POKEd into register S+1.

A random sound like this could be used in several types of games, perhaps as a background when characters or screen is set up.

Ring modulation. Another technique you can use when creating random sounds is *ring modulation*. This produces harmonic structures that are not even and smooth as in normally generated notes. This lets you create sounds like bells and gongs.

To use ring modulation, you have to turn on bit 2 of register 4 (for voice 1) at the same time you select the triangle wave form and set the gate bit. In other words, by POKEing S+4, 21, you'll turn on the ring modulation ability of the SID chip's voice 1.

Ring modulation creates different sounds depending on the frequencies of voices 1 and 3. The overtone structure of voice 3 is added to that of voice 1. All you really have to know, however, is that it makes an interesting sound.

Making a few changes to the random note program can add ring modulation.

Program 10-5. Ring Modulation

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
1Ø S=54272                                  :rem 245
2Ø POKE S+18,128:POKE S,75                  :rem 147
3Ø POKE S+5,Ø:POKE S+6,24Ø                  :rem 128
4Ø POKE S+14,12:POKE S+15,5                 :rem 179
6Ø POKE S+24,2Ø7:POKE S+4,21                :rem 232
7Ø FOR L=Ø TO 15:POKE S+1,PEEK(S+27) AND 224
                                             :rem 28
75 FOR T=Ø TO 1ØØ:NEXT:NEXT:POKE S+4,Ø     :rem 177
8Ø FOR T=Ø TO 5ØØ:NEXT:GOTO 6Ø              :rem 154
```

Look at line 70. The PEEK (S+27) AND 224 creates a lower frequency. No matter what the value received from PEEK (S+27), the frequency value will never be more than 224.

You can hear even stranger sound effects using ring modulation by changing the frequencies of voice 1 and 3. Here's an example:

Program 10-6. Changing Ring Modulation

```
10 S=54272                                          :rem 245
15 POKE S+14,134:POKE S+15,10                        :rem 22
16 POKE S+3,8:POKE S+2,0:POKE S+14,255:POKE S+15,3
   :POKE S+18,32                                    :rem 193
18 POKE S+24,15                                      :rem 14
20 POKE S+5,15:POKE S+6,240                         :rem 181
30 POKE S+4,19                                      :rem 218
50 FOR T=1 TO 80:POKE S+1,T+0:NEXT                  :rem 133
55 FOR T=80 TO 1 STEP-1:POKE S+1,T+50:NEXT          :rem 89
70 GOTO 30                                           :rem 3
```

Lines 50 and 55 are where this frequency change is executed. Line 50 causes voice 1's pitch to rapidly go from 1 to 80. Line 55 brings it back down again, but adds a value of 50, so actually it drops from a value of 130 to 51.

Ring modulation is an interesting technique of producing sounds. However, since voice 3 cannot be used for another purpose if ring modulation is operating, you're restricted to only two voices. If you need more than two voices at a time, you'll have to compromise.

Sound Effects

Here are two sound effects which use ring modulation. Although you may find use for these in your own game, their true purpose is to show you what can be created with the SID chip and ring modulation. Don't be afraid of changing values in either of these programs. Oftentimes that's the best way to learn.

Program 10-7. Raygun

```
5 PRINT"{CLR}{WHT}{5 DOWN}{YEL}RAYGUN{WHT} SOUND D
  EMO----- TO HEAR THE NEXT{DOWN}                   :rem 205
6 PRINT"RAYGUN SOUNDS, JUST HIT ANY KEY. THE
                                                    :rem 206
7 PRINT"{DOWN}SOUNDS ARE DONE WITH RING MOD. AND B
  Y                                                 :rem 185
8 PRINT"{DOWN}HITTING ANY KEY YOU ARE CHANGING THE
                                                    :rem 213
```

```
9 PRINT"{DOWN}FREQUENCY OF VOICE 3 TO A HIGHER PIT
  CH.                                              :rem 42
10 POKE 53281,0:S=54272:F=0                        :rem 170
15 POKE S+14,250:POKE S+18,17                      :rem 31
20 POKE S,1:POKE S+4,16:POKE S,15:POKE S+6,16:POKE
   S+24,143                                        :rem 254
25 POKE S+4,21:POKE S+18,17:POKE S+15,F            :rem 122
30 FOR X=200 TO 100 STEP-5:POKE S+1,X:NEXT         :rem 88
35 FOR X=100 TO 0 STEP-1:POKE S+1,X:NEXT           :rem 247
40 POKE S+4,20:POKE S+18,16                        :rem 182
50 FOR T=0 TO 100:NEXT:F=F+5:IF F=150 THEN F=1
                                                   :rem 226
55 GET A$:IF A$="" THEN 55                         :rem 247
60 GOTO 25                                          :rem 6
```

Program 10-8. Ring Modulation Demo

```
1 PRINT"{CLR}{WHT}{2 DOWN}{YEL}"SPC(10)"RING MODUL
  ATION DEMO{WHT}                                  :rem 106
2 PRINT"{3 DOWN}CONTROLS:{5 SPACES}{RVS}F1{OFF}
  {2 SPACES}TURN OFF VOLUME                        :rem 217
3 PRINTSPC(14)"{DOWN}{RVS}F3{OFF}{2 SPACES}RAISE F
  REQ. VOICE 3                                     :rem 37
4 PRINTSPC(14)"{DOWN}{RVS}F4{OFF}{2 SPACES}LOWER S
  AME                                              :rem 93
8 PRINTSPC(14)"{DOWN}{RVS}F5{OFF}{2 SPACES}RAISE F
  REQ. VOICE 1                                     :rem 42
9 PRINTSPC(14)"{DOWN}{RVS}F6{OFF}{2 SPACES}LOWER S
  AME                                              :rem 100
10 PRINTSPC(14)"{DOWN}{RVS}F7{OFF}{2 SPACES}INCREA
   SE INTERVAL                                     :rem 141
11 PRINTSPC(14)"{DOWN}{RVS}F8{OFF}{2 SPACES}DECREA
   SE INTERVAL                                     :rem 129
14 POKE 53281,0:S=54272:P=6:P1=100:X1=100:X2=150
                                                   :rem 111
15 POKE S+14,20:POKE S+15,P1:POKE S+18,17          :rem 228
20 POKE S,1:POKE S+4,16:POKE S+5,15:POKE S+6,240:P
   OKE S+24,143                                    :rem 141
25 POKE S+4,21:POKE S+18,17                        :rem 187
30 FOR X=X1 TO X STEP P:POKE S+1,X:NEXT             :rem 4
35 FOR X=X2 TO X1 STEP -P:POKE S+1,X:NEXT          :rem 104
36 GET A$:IF A$ ="{F1}" THEN POKE S+24,0           :rem 128
40 IF A$="{F3}" THEN P1=P1+5:IF P1>250 THEN P1=250
                                                   :rem 208
45 IF A$="{F4}" THEN P1=P1-5:IF P1<5 THEN P1=5
                                                    :rem 21
50 POKE S+15,P1                                     :rem 37
60 IF A$="{F7}" THEN P=P+1:IF P>20 THEN P=20
                                                   :rem 162
```

```
70 IF A$="{F8}" THEN P=P-1:IF P<1 THEN P=1 :rem 69
80 IF A$="{F5}" THEN X1=X1+5:X2=X2+5:IF X2>250 THE
   N X1=200:X2=250                          :rem 116
90 IF A$="{F6}" THEN X1=X1-5:X2=X2-5:IF X1<25 THEN
    X1=25:X2=75                             :rem 244
200 GOTO 30                                  :rem 46
```

Filtering Sounds

Another powerful and useful feature of the SID chip is its variable filter controls. As with any filter, which lets some things pass through and stops others, the filters in the SID chip stop certain frequencies and allow others to be heard. Filtering sounds on the 64 creates purer sounds, sounds without overtones. But the most important thing to remember is the effect that filters can have on your game sounds.

When you use a filter, you'll have to decide two things. First, you have three types of filters to choose from. They are the High Pass filter, the Band Pass filter, and the Low Pass filter. All three are found in register S + 24. By setting different bits, you choose the filter. Refer to Table 1, SID Register Table, for the bit values. Second, filters are set by selecting a frequency cutoff point. This is similar to adjusting the filters' weave. The tighter the weave, the smaller the holes, and the fewer sounds let through.

If you're using the High Pass filter, for example, and set the frequency cutoff point, all frequencies higher than that point are allowed to pass through unchanged. The opposite is true if you use the Low Pass filter. Only frequencies *below* the cutoff point pass unchanged. The Band Pass filter, on the other hand, lets only those frequencies close to the cutoff point through.

Registers 21 and 22 set the frequency of the cutoff point. Register 23 allows you to switch sounds from any of the 64's voices to pass through the filters. Register 24 selects the filter. Using these registers, you can choose a cutoff point, select a filter, and channel a voice through that filter. By sweeping the cutoff point from its lowest to its highest value, you can create interesting sound effects.

Program 10-9 is an example of filtered sound. The program creates a rushing sound by using the noise generator of voice 1. The High Pass filter is used, while the frequency cutoff point is changed with a FOR-NEXT loop.

Here's how the program works:

Line	Function
10	Set variable *S* to equal register 0.
20	Set note frequency in voice 1. Switch on filter for voice 1 (S+23,1).
30	Select filter (High Pass filter), as well as turn all four volume control bits on. This is done with S+24,79. Other two POKEs set ADSR for note.
40	Set note frequency in voice 3.
50	Set wave form to noise wave form, turn gate bit on.
60	Establish variable F. F will decrease, starting at 250 and ending with 2.
70	Set frequency cutoff point as F. Set note frequency as F.
80	Turn gate bit off.
90	Wait for key to be pressed.
100	Return to line 50 and execute main loop again.

Program 10-9. Torpedo

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 S=54272                                          :rem 245
20 POKE S,20:POKE S+1,5:POKE S+23,1                 :rem 148
30 POKE S+24,79:POKE S+5,15:POKE S+6,246            :rem 165
40 POKE S+15,30:POKE S+14,30                        :rem 225
50 POKE S+4,131                                       :rem 7
60 FOR F=250 TO 2 STEP-1.5                           :rem 57
70 POKE S+22,F:POKE S+1,F:NEXT                      :rem 237
80 POKE S+4,130                                       :rem 9
90 GET A$:IF A$="" THEN 90                          :rem 245
100 GOTO 50                                          :rem 47
```

The next program is simply a demonstration of the filtering capabilities of the 64. After you've entered and RUN the program, you can hear the differences as you change from one filter to another. It begins with the High Pass filter. Once you've heard what it can do, experiment and change some of the program's parameters.

Program 10-10. Multiple Filters

```
5 REM MULTI- FILTER DEMO                            :rem 188
10 GOSUB 200                                        :rem 115
18 POKE 53281,0:S=54272:X=0:F=35:RM=140             :rem 148
```

```
20 POKE S,18:POKE S+1,F:POKE S+18,16       :rem 230
25 POKE S+15,RM:POKE S+14,200:POKE S+23,129:rem 99
30 POKE S+4,32:POKE S+5,15:POKE S+6,248    :rem 106
40 POKE S+21,8:POKE S+129,33:POKE S+24,79  :rem 214
50 POKE S+4,21:POKE S+22,X                 :rem 164
60 FOR F=150 TO 200 STEP 5:POKE S+1,F:NEXT :rem 15
70 X=X+8:IF X>255 THEN X=0                 :rem 166
80 GET A$:IF A$="{F7}" THEN RM=RM+5:IF RM>200 THEN
    RM=2                                   :rem 99
85 POKE S+15,RM                            :rem 75
90 IF A$="{F8}" THEN POKE S+24,0:END       :rem 25
100 IF A$="{F1}" THEN POKE S+24,79         :rem 105
110 IF A$="{F3}" THEN POKE S+24,47         :rem 102
120 IF A$="{F5}" THEN POKE S+24,31         :rem 97
150 GOTO 50                                :rem 52
200 PRINT"{CLR}{DOWN}"SPC(11)"{YEL}MULTI FILTER DE
    MO{WHT}                                :rem 219
210 PRINT"{2 DOWN}{2 SPACES}THIS DEMO PLAYS AN OSC
    ILLATING, RING                         :rem 110
220 PRINT"{DOWN}MODULATED SOUND AS THE FILTER CUTO
    FF IS{DOWN}                            :rem 43
230 PRINT"SWEPT FROM BOTTOM TO TOP. CONTROLS ARE:
    {2 DOWN}                               :rem 76
240 PRINT"{5 SPACES}{RVS}F1{OFF}{2 SPACES}HIGH PAS
    S FILTER{DOWN}                         :rem 142
250 PRINT"{5 SPACES}{RVS}F3{OFF}{2 SPACES}BAND PAS
    S FILTER{DOWN}                         :rem 134
260 PRINT"{5 SPACES}{RVS}F5{OFF}{2 SPACES}LOW PASS
     FILTER{DOWN}                          :rem 102
270 PRINT"{5 SPACES}{RVS}F7{OFF}{2 SPACES}CHANGE R
    ING MOD. FREQUENCY{DOWN}               :rem 16
280 PRINT"{5 SPACES}{RVS}F8{OFF}{2 SPACES}END DEMO
                                           :rem 103
300 RETURN                                 :rem 115
```

You now have the background needed to create sound effects on the Commodore 64. What do you do with them? Place them in a game program, *your* game program.

Background Sound

One possible use for sound in a game program is as a background, which can be used as a mood producer. Sound can influence the way the game is played, even making it more enjoyable. Although the sounds you choose for your own game may be different than the following example, once you've looked through this section, you should be able to apply its techniques.

If you want a sound to be continuous throughout the game, the routine must be included as part of the main loop. If you wanted an arcade-like sound, for example, you might set up something like this:

Program 10-11. Background Sound

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 S=54272:RM=20                                    :rem 109
20 POKE S+1,15:POKE S+5,16:POKE S+6,240              :rem 96
30 POKE S+15,RM:POKE S+24,31                         :rem 30
40 POKE S+4,21                                      :rem 212
50 RM=RM-1:IF RM=10 THEN RM=20                      :rem 177
60 GOTO 30                                            :rem 2
```

This routine creates a ring modulated effect.

You still have to add this to the main loop of a program. All you have to do is change the line numbers to fit your program, and eliminate the last line of the routine. By adding this sound routine to the main loop of the "Mission: Nova!" game program from Chapter 9, you can hear a constant background sound as you play.

Program 10-12. Main Loop Background Sound

```
25 S=54272:MR=20                                    :rem 115
127 REM                                             :rem 126
128 REM BACKGROUND SOUND ROUTINE                     :rem 14
129 REM                                             :rem 128
130 POKE S+1,15:POKE S+5,16:POKE S+6,240            :rem 146
140 POKE S+15,MR:POKE S+24,24:POKE S+4,21           :rem 252
150 MR=MR-1:IF MR=10 THEN MR=20                     :rem 226
```

The variables *S* and *MR* are established in the initialization section of the program, before the main loop begins. Line 100 starts the main loop of the Mission: Nova! program. Lines 130-140 POKE in the values for the sound we want, while line 150 varies the frequency of the note in voice 2. Each time the main loop executes, this sound is played.

Try it. You'll notice that the movement slows down somewhat. This is unavoidable when you place a sound in the main loop. It's a good idea to keep any background sound simple, for the more complicated it is, the more the main loop is slowed.

Sound Subroutines

You probably won't want every sound in the game to play all the time. Instead, you'll want to hear certain sounds when certain events occur on the screen. The sound you hear when a player-controlled character is lost, for instance, would fit in this category. Perhaps you want a sound when the joystick fire button is pressed, or a loud tone when a target is hit. In situations like these, the sounds are called by 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 noticeable, and you'll see the characters stop on the screen as the sound is executed. Yet many games use subroutines, simply because they are easier to incorporate into a program. In some games, the delay is not apparent. You must make the decision. Will the delay decrease the game's playability? Will the player enjoy it less? If you think not, try the subroutine technique.

Let's add some sound subroutines to our Mission: Nova! program. We'll want sounds for each of the four things the starship bumps into. Not only will this make the game more entertaining, but it'll also provide the player with feedback as the game is played. The sounds will tell the player what is happening without him or her having to visually check each movement.

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

Line 700 tests to see if the starship is in contact with a small star. If it is, the score falls and the ship suffers damage. That's indicated by changing the ship's color.

Line 710 checks if the ship is on the large space station. Remember that the ship is then renewed to full power.

Line 720 tests to find out if the ship is on the small space station. Nothing is added or subtracted for this.

Line 730 checks for the only remaining possibility, that the ship is in contact with a large, nova star. Points are added for this.

Each sound should indicate whether the collision is positive or negative. This is how you can provide feedback for the player through sound.

All four lines must include a GOSUB command, so that

the 64 refers to the right subroutine. They could look like this.

```
698 REM SMALL STAR                                    :rem 62
699 REM                                              :rem 140
700 IF W=28 THEN P=P-100:NC=NC+1:GOSUB 1500:RETURN
                                                     :rem 125
707 REM                                              :rem 130
708 REM SPACE STATION                                 :rem 17
709 REM                                              :rem 132
710 IF W=27 THEN NC=CS:W=30:GOSUB 1550:RETURN
                                                     :rem 135
717 REM                                              :rem 131
718 REM USED SPACE STATION                            :rem 67
719 REM                                              :rem 133
720 IF W=30 THEN GOSUB 1600:RETURN                   :rem 175
727 REM                                              :rem 132
728 REM NOVA!                                        :rem 218
729 REM                                              :rem 134
730 P=P+(8*NH)*INT(QQ-Q):W=28:GOSUB 1650:RETURN
                                                      :rem 73
```

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

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

Program 10-13. Small Star Collision

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
1500 FOR I=0 TO 24:POKE S+I,0:NEXT                   :rem 113
1510 POKE S+24,15:POKE S+12,160:POKE S+13,252
                                                     :rem 137
1520 POKE S+8,80:POKE S+7,40:POKE S+11,129
1530 FOR I=0 TO 100:NEXT                              :rem 19
1540 POKE S+11,128:RETURN                            :rem 186
```

This subroutine creates a small explosion sound to indicate that points have been lost and the starship has been damaged. Notice that line 1500 sets all the sound registers back to 0, so that previous POKEs to the memory locations don't disrupt the sound. This is good practice when using the 64. If you don't do this, at times the sound will come out different than you expected, or perhaps not be heard at all.

Line 1540 includes a RETURN command so that the program shifts back to line 700 after the sound subroutine executes.

Program 10-14. Meeting the Large Space Station

```
155Ø FOR I=Ø TO 24:POKE S+I,Ø:NEXT:SP=1Ø  :rem 241
156Ø POKE S,15Ø:POKE S+1,SP               :rem 247
157Ø POKE S+5,Ø:POKE S+6,24Ø:POKE S+24,15:POKE S+4
     ,17                                  :rem 12Ø
158Ø FOR SP=1Ø TO 25Ø STEP 2:POKE S+1,SP:NEXT
                                          :rem 254
159Ø FOR T=Ø TO 1ØØ:NEXT:RETURN           :rem 62
```

As the starship meets the large space station, this subroutine plays a quick sound which rises in pitch. The routine is almost identical to Program 10-3, Frequency Change. Because it rises in pitch, it sounds like something filling up, which is what the station does for the starship, returning it to its original power level.

Program 10-15. Landing on the Small Space Station

```
16ØØ FOR I=Ø TO 24:POKE S+I,Ø:NEXT         :rem 114
161Ø POKE S,15Ø:POKE S+1,2ØØ:POKE S+5,8:POKE S+6,2
     48                                     :rem 73
162Ø POKE S+24,15:POKE S+4,17              :rem 29
163Ø FOR T=Ø TO 1ØØ:NEXT                   :rem 31
164Ø POKE S+4,16:FOR T=Ø TO 1ØØ:NEXT:RETURN
                                           :rem 232
```

Since there is no effect when the starship touches the small station, this blip sound works well. It's identical to the One Note program used earlier in the chapter. This sound is useful, if only to tell the player that the small station does not give the starship an increase in power.

Program 10-16. Meeting a Nova Star

```
165Ø FOR I=Ø TO 24:POKE S+I,Ø:NEXT:SP=1Ø  :rem 242
166Ø POKE S,15Ø:POKE S+1,SP               :rem 248
167Ø POKE S+5,Ø:POKE S+6,24Ø:POKE S+24,15:POKE S+4
     ,17                                  :rem 121
168Ø FOR SP=1Ø TO 25Ø STEP 4:POKE S+1,SP:NEXT
                                            :rem 1
169Ø FOR T=Ø TO 1ØØ:NEXT:RETURN            :rem 63
```

This subroutine creates a sound almost identical to the one used when the starship lands on the large space station. The major difference is that in line 1680, the STEP 4 command is used. This simply shortens the sound to half of what it was in the large space station subroutine. Although the sound is again one of filling something up, making it only half as long is

appropriate, for the increase in points is not as positive as renewing the starship's power supply.

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

Sound Effect Samples

To get you started in creating your own sound effects on the 64, here are six sample sound effects. Using the SID Register Table, you should be able to figure out what registers are used, and the values POKEd to those registers, for each of the effects. You can use these effects as they are now, if they'll fit into your game design, or you can change some of the sound parameters to create an entirely different sound.

Program 10-17. Train

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 S=54272                                  :rem 245
100 POKE S+24,15 :REM SET VOLUME HIGH        :rem 56
110 POKE S+1,80 :REM HIGH FREQUENCY         :rem 244
120 POKE S+5,80 :REM A/D                    :rem 219
130 POKE S+6,245 :REM S/R                    :rem 48
140 FORSP=10000TO25STEP-400                  :rem 84
150 Q=SQR(SP)                                :rem 14
160 GOSUB500 :REM DO AN ON/OFF              :rem 147
180 GETA$:IFA$=""THENNEXT                   :rem 251
190 Q=25:GOSUB500                           :rem 222
200 GETA$:IFA$=""THEN190                     :rem 79
210 END                                     :rem 106
500 POKE S+4,129 :REM START NOISE           :rem 104
510 FORI=1TO10+Q:NEXT :REM DELAY            :rem 186
520 POKE S+4,128 :REM BEGIN RELEASE         :rem 195
530 FORI=1TO50+Q:NEXT:RETURN                 :rem 77
```

Program 10-18. Whistling Bomb

```
5 S=54272                                   :rem 201
10 FORI=0TO22:POKE S+I,0:NEXT                :rem 10
100 POKE S+24,15 :REM SET VOLUME HIGH        :rem 56
110 POKE S+5,80 :REM A/D                    :rem 218
115 POKE S+12,160 :REM A/D                   :rem 60
120 POKE S+6,255 :REM S/R                    :rem 48
```

```
125 POKE S+13,252 :REM S/R                    :rem 96
130 POKE S+4,17                               :rem 9
140 POKE S+4,16                               :rem 9
150 FORI=255TO50STEP-1:POKE54273,I            :rem 36
155 FORJ=1TO5:NEXT:NEXT                       :rem 4
160 POKE S+1,10                               :rem 2
170 POKE S+8,1                                :rem 218
180 POKE S+5,112:POKE S+6,252                 :rem 29
190 POKE S+4,129:POKE S+11,129                :rem 84
200 FORI=1TO200:NEXT                          :rem 222
210 POKE S+4,128:POKE S+11,128                :rem 75
```

Program 10-19. Klaxon

```
5 S=54272                                     :rem 201
10 FORI=0TO22:POKE S+I,0:NEXT                 :rem 10
100 POKE S+24,15 :REM SET VOLUME HIGH         :rem 56
110 POKE S+5,80 :REM A/D                      :rem 218
120 POKE S+6,243 :REM S/R                     :rem 45
130 POKE S+3,4                                :rem 212
140 POKE S+4,65                               :rem 13
150 FORI=20TO140STEP5:POKE S+1,I:NEXT         :rem 20
155 POKE S+4,64:FORI=1TO50:NEXT               :rem 107
170 GETA$:IFA$=""THEN140                      :rem 80
180 POKE S+4,64                               :rem 16
```

Program 10-20. Landing Saucer

```
5 S=54272                                     :rem 201
10 FORI=0TO22:POKE S+I,0:NEXT                 :rem 10
100 POKE S+24,15 :REM SET VOLUME HIGH         :rem 56
110 POKE S+5,80 :REM A/D                      :rem 218
120 POKE S+6,243 :REM S/R                     :rem 45
130 POKE S+3,7                                :rem 215
140 FORJ=50TO17STEP-1:POKE S+4,65             :rem 191
150 POKE S+1,J:FORI=1TO200:NEXT               :rem 112
155 POKE S+4,64:FORI=1TO50:NEXT               :rem 107
170 GETA$:IFA$=""THENNEXT                     :rem 250
```

Program 10-21. Gong

```
5 S=54272                                     :rem 201
10 FORI=0TO22:POKE S+I,0:NEXT                 :rem 10
100 POKE S+24,143 :REM SET VOLUME HIGH        :rem 106
110 POKE S+5,16 :REM A/D                      :rem 217
115 POKE S+19,16 :REM A/D                     :rem 19
120 POKE S+6,252 :REM S/R                     :rem 45
125 POKE S+20,249 :REM S/R                    :rem 100
140 POKE S+4,21:POKE S+18,17                  :rem 233
```

```
150 POKE S+1,68:POKE S+15,42:FORI=1TO200:NEXT
                                              :rem 115
155 POKE S+4,20:POKE S+18,16:FORI=1TO400:NEXT
                                              :rem 117
170 GETA$:IFA$=""THEN140                       :rem 80
```

Program 10-22. Siren

```
5 S=54272                                     :rem 201
10 FORI=0TO22:POKE S+I,0:NEXT                  :rem 10
100 POKE S+24,143 :REM SET VOLUME HIGH        :rem 106
110 POKE S+5,16 :REM A/D                      :rem 217
115 POKE S+19,16 :REM A/D                      :rem 19
120 POKE S+6,252 :REM S/R                      :rem 45
125 POKE S+20,249 :REM S/R                    :rem 100
140 POKE S+4,21:POKES+18,17                   :rem 233
150 POKE S+1,2:POKE S+15,34:FORI=1TO200:NEXT
                                               :rem 56
151 POKE S+4,20:POKE S+18,16:FORI=1TO300:NEXT
                                              :rem 112
152 POKE S+4,21:POKE S+18,17                  :rem 236
153 POKE S+1,2:POKE S+15,30:FORI=1TO200:NEXT
                                               :rem 55
155 POKE S+4,20:POKE S+18,16:FORI=1TO300:NEXT
                                              :rem 116
160 GETA$:IFA$=""THEN140                       :rem 79
```

All but the Whistling Bomb sound effect can be stopped by pressing any key.

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's worth the time and effort to include sound, simply for the impact it has on a game. A game without sound is just not a game to most players. Something vital is missing.

11
Introductions, Instructions, and Farewells

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. 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.

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. 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 9, "Mission: Nova!," it would look like this:

```
5 GOSUB 1700                                    :rem 125
```

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 11-1. Introduction

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
1700 PRINT"{CLR}":POKE 53281,15              :rem 42
1705 PRINT "{2 DOWN}{16 RIGHT}{BLK}STARMOVE"
                                             :rem 145
1710 PRINT "{2 DOWN}{2 RIGHT}{BLU}MANEUVER YOUR ST
     AR SHIP ACROSS THE"                     :rem 225
1720 PRINT "{2 RIGHT}GALAXY."                :rem 201
1730 PRINT "{2 DOWN}{2 RIGHT}AVOID THE STARS AND R
     EFUEL AT NOVAS."                        :rem 185
1740 PRINT "{2 DOWN}{2 RIGHT}LAND AT THE SPACE STA
     TION FOR POWER."                        :rem 190
1750 PRINT "{6 DOWN}{2 RIGHT}HIT ANY KEY TO CONTIN
     UE"                                     :rem 252
1755 GET A$:IF A$="" THEN 1755               :rem 199
```

As the program runs, notice that the screen changes color. This is a nice effect that is done simply by POKEing a different value into location 53281. 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, then the words will seem invisible.

It is also simple to change the character color of the title, as was done in this sample subroutine in line 1705. 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 1710 this was done by hitting CTRL and BLUE at the same time.

It's a good idea to double-space between each line, or between your short paragraphs, as was 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 either 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 1710 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 of print separately. At the end of each screen, as in Program 11-1, you could add the line:

PRINT "HIT ANY KEY TO CONTINUE"

followed by:

GET A$:IF A$=" "THEN xxx

where xxx is the line number of the GET A$ command.

This holds the screen until the player presses a key, then it displays 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's all you're using, to return to the game program you need only add a line such as this:

```
175Ø FOR T1=Ø TO 5ØØØ:NEXT:RETURN
```

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

Introduction sound. Adding sound is quite easy. Before the program RETURNs, insert the POKE statements for whatever effect you want. The sound then plays before the game begins and while the introduction is on the screen.

Here's an example of a sound effect you can add to Mission: Nova!

Program 11-2. Introduction Sound

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E

```
19ØØ FOR I=Ø TO 24:POKE S+I,Ø:NEXT:F=Ø      :rem 98
1915 POKE S+14,25Ø:POKE S+18,17             :rem 137
```

```
1920 POKE S,1:POKE S+4,16:POKE S,15:POKE S+6,16:PO
     KE S+24,143                                  :rem 104
1925 POKE S+4,21:POKE S+18,17:POKE S+15,F :rem 228
1930 FOR X=200 TO 100 STEP-5:POKE S+1,X:NEXT
                                                  :rem 194
1935 FOR X=100 TO 0 STEP-1:POKE S+1,X:NEXT :rem 97
1940 POKE S+4,20:POKE S+18,16                     :rem 32
1950 FOR T=0 TO 100:NEXT:F=F+5:IF F=150 THEN F=1
                                                  :rem 76
1960 GET A$:IF A$="" THEN 1525                   :rem 192
1970 RETURN                                      :rem 177
```

This is a version of the raygun sound in Chapter 10. All you need to do is add this sound subroutine, along with this line at the end of the introduction routine:

```
1755 GOSUB 1900                                   :rem 28
```

Now the sound routine will play until a key is pressed, which will then send it back to the program.

Instructions

Before the game starts, the player must know how to play the game. On some arcade games, this is done with printed documentation on the game machine. Since you can't print instructions on every player's Commodore 64, the method you'll want to use is documentation within the game program. This is where the player finds out such things as:

- How to start the game
- How to move the player-controlled characters or sprites
- The game scenario, or story line
- How points are awarded or subtracted
- How extra turns of player characters are earned
- Explanations of the different game levels
- Details of any special game features

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 11-3. Instructions

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

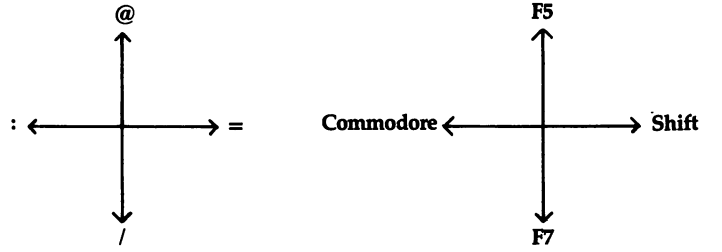
```
1760 PRINT "{CLR}{3 DOWN}{2 RIGHT}HIT F5 KEY FOR '
     UP'"                                         :rem 194
```

```
1770 PRINT "{2 DOWN}{2 RIGHT}HIT F7 KEY FOR 'DOWN'
     "                                        :rem 180
1780 PRINT "{2 DOWN}{2 RIGHT}HIT COMMODORE KEY FOR
      'LEFT'"                                 :rem 208
1790 PRINT "{2 DOWN}{2 RIGHT}HIT SHIFT KEY FOR 'RI
     GHT'"                                    :rem 253
1800 PRINT "{6 DOWN}{2 RIGHT}HIT ANY KEY TO CONTIN
     UE"                                      :rem 248
1820 GET A$:IF A$="" THEN 1820                :rem 185
```

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 11-1 on the screen, showing the corresponding directions and making it easier for the player to understand.

Figure 11-1. 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 11-4. More Instructions

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
1830 PRINT "{CLR}{2 DOWN}{2 RIGHT}SCORE POINTS BY
     {SPACE}REFUELING AT NOVAS."              :rem 107
```

```
1840 PRINT "{2 DOWN}{2 RIGHT}POINTS DEDUCTED FOR C
     RASHING INTO"                                  :rem 137
1850 PRINT "{2 RIGHT}STARS AND TAKING TOO MUCH TIM
     E."                                            :rem 115
1860 PRINT "{2 DOWN}{2 RIGHT}YOU CAN CRASH INTO EI
     GHT STARS"                                     :rem 116
1870 PRINT "{2 RIGHT}BEFORE LANDING ON THE SPACE S
     TATION."                                       :rem 197
1880 PRINT "{2 DOWN}{2 RIGHT}THE GAME ENDS WHEN YO
     UR SHIP'S POWER"                               :rem 223
1890 PRINT "{2 RIGHT}IS GONE."                      :rem 208
1895 PRINT ?"{6 DOWN}{2 RIGHT}HIT KEY TO START."
                                                    :rem 117
1896 GET A$:IF A$="" THEN GOTO 1896                  :rem 12
1897 RETURN                                         :rem 185
```

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 1897, 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.

Again, a subroutine called at the program's outset is one way to do this. The game "Spark" from Chapter 9 is one in which levels could be set using the statement:

```
5 GOSUB 800                                         :rem 77
```

You must also eliminate the *P=300* expression from line 50 of the original version of the Spark program. If you don't, the following subroutine will not work properly.

Line	Function
800	Clear the screen and POKE in a screen color change.
810-840	Print the instructions for setting the different levels of play. Note the {BLU} 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.

Program 11-5. Spark Levels

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
800 PRINT "{CLR}":POKE 53281,7              :rem 203
810 PRINT "{3 DOWN}{2 RIGHT}{BLU}CHOOSE LEVEL OF P
    LAY BY HITTING KEY"                     :rem 149
815 PRINT "{2 RIGHT}AND PRESSING {RVS}RETURN{OFF}"
                                            :rem 107
820 PRINT "{2 DOWN}{2 RIGHT}1"TAB(8)"EASIEST"
                                            :rem 170
830 PRINT "{2 DOWN}{2 RIGHT}2"TAB(8)"HARDER"
                                             :rem 84
840 PRINT "{2 DOWN}{2 RIGHT}3"TAB(8)"DIFFICULT":PR
    INT:PRINT:PRINT                         :rem 143
860 OPEN 1,0                                 :rem 93
870 INPUT#1,A$:CLOSE 1,0                     :rem 81
880 IF A$="1" THEN P=300:RETURN             :rem 175
890 IF A$="2" THEN P=200:RETURN             :rem 176
900 IF A$="3" THEN P=100:RETURN             :rem 168
```

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

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 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:

```
17Ø IF CH<Ø THEN 61Ø                    :rem 228
```

and

```
33Ø IF P=Ø THEN 6ØØ                     :rem 167
```

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:

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 four 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 64 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.

Program 11-6. Spark End

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
600 PRINT:PRINT "{CLR}{3 DOWN}{2 RIGHT}TOO LATE!
    {2 RIGHT}SCORE IS 0":GOTO 620          :rem 244
610 PRINT:IF P>0 THEN PRINT "{CLR}{3 DOWN}
    {2 RIGHT}YOU MADE IT!{2 RIGHT}SCORE IS"P
                                            :rem 31
620 PRINT "{4 DOWN}{2 RIGHT}TO PLAY AGAIN, HIT ANY
     KEY"                                    :rem 2
625 POKE 198,0                             :rem 202
630 GET A$:IF A$="" THEN 630                :rem 85
640 GOTO 5                                  :rem 8
```

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 another subroutine called when the game is over. The sample game Mission: Nova! could use this technique, and the program could appear as:

Line	Function
905-979	The landing saucer sound effect is created. This is identical to the effect included in Chapter 10.
980	Print final score and ask the player whether another game is to be played.
985	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 *not* wanted, then the program will end in line 990 after the delay.

Program 11-7. Mission: Nova! End

```
905 S=54272                                 :rem 50
910 FORI=0TO22:POKE S+I,0:NEXT              :rem 67
920 POKE S+24,15 :REM SET VOLUME HIGH       :rem 66
```

```
930 POKE S+5,80 :REM A/D                          :rem 228
940 POKE S+6,243 :REM S/R                          :rem 55
950 POKE S+3,7                                    :rem 225
960 FORJ=50TO17STEP-1:POKE S+4,65                 :rem 201
970 POKE S+1,J:FORI=1TO200:NEXT                   :rem 122
975 POKE S+4,64:FORI=1TO50:NEXT                   :rem 117
979 NEXT                                          :rem 232
980 PRINT "{CLR}{2 DOWN}{2 RIGHT}"P" POINTS":PRINT
    :PRINT"{2 DOWN}{2 RIGHT}PRESS ANY KEY TO PLAY
    {SPACE}AGAIN"                                 :rem 151
985 FOR I=0 TO 3000:A=PEEK(197):IF A<>64 THEN GOSU
    B 450:GOTO 280                                :rem 187
990 NEXT I:END                                     :rem 59
```

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. 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 64. 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.

12
The Shape of the Game

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 example, 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, 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? If you use L for the player-figure's location, then use 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 of our example 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),OC
```

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, 498
```

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 reaches this line only 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 happen only 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 900,700,900,800,900,500,900,8
    00: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 and Video Bank Selection

Memory's not something you often worry about with the Commodore 64, but it's nice to know it's there. Unlike other, smaller computers, the 64 has enough memory to satisfy even the most advanced game programmer. You'll seldom find that you're running out of memory space when you create your own game programs.

However, the 64 can give you problems when you're designing that especially long game. Although the computer has 64K of RAM (Random Access Memory), the VIC-II chip can read only one bank of 16K at a time. This is an important concept, for in order to operate properly, the VIC-II chip must be able to find a number of different things in the block it is reading. To perform all its assigned functions, the VIC-II chip must be able to find:

- A character set to use. This is normally provided by the ROM character generator.
- An area to use as screen memory. Normally, this is at locations 1024-2023.
- An area for color memory. Fortunately, this never changes. It is always located at addresses 55296-56295.
- If sprites are used, the sprite DATA must be in this same 16K bank.
- When custom characters are used, the DATA used to create them must also be located in this bank.

The VIC-II chip can access only one 16K bank at a time. When the computer is first turned on, the bank located at addresses 0 to 16383 is accessed. For an explanation of the other banks, and their locations, refer to Chapter 1.

Why would you want to switch so that the VIC-II chip could read another bank? If your program was quite long, over 14K, or if you are designing a custom character set and have a lengthy game, you'll want to use bank switching. This was first explained in Chapter 1, so if you've forgotten how to switch banks, or skipped over that section earlier, refer to it now. You've also seen how to relocate custom character sets, a technique demonstrated in Chapter 4.

Another use for bank switching is when you're planning on using high-resolution graphics. We haven't looked at this excellent tool of the 64, primarily because of its complexity, but there are references available which will give you more information. There is a short list of other resources, including those

which explain high-resolution graphics, in the back of this book.

Video Banks

Bank 0, the normal section of the computer that the chip reads, sees RAM from 0 to 4095 and from 8192 to 16383. The ROM character generator is in the area from 4096 to 8191. Screen memory normally is located from addresses 1024 to 2023.

Bank 1 sees RAM from 16384 to 32767. There is no character generator in this bank.

Bank 2 sees RAM in locations 32768 to 36863 and from 40960 to 49151. The ROM character generator is located at addresses 36864 to 40959.

Bank 3 is similar to Bank 1. It sees RAM from locations 49152 to 65535. There is no character generator in this bank.

It is possible to use all four banks during the course of a program simply by switching from one to another as the program executes. There are a number of things you should be aware of, however, before you begin switching banks on the 64.

Character sets. To use Banks 1 or 3, you'll have to create your own character set, or copy the existing one into an area of memory within that bank. If you don't, you won't have a character set to use, since neither of these banks includes a ROM character generator.

Screen memory. If you change banks, remember that you will also have to relocate screen memory. This is true if you use any bank but Bank 0. Refer to Chapter 1 for an explanation on how to relocate this section of memory.

Character and sprite DATA. If you are using custom characters, or sprites, or both, you should make sure that the DATA information which creates those figures is included in the bank you're working with.

13
Missiles

Missiles

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 be allowed to control the ball's actual trajectory, its path across the screen. That is almost always controlled by the computer. The *launch* is player-controlled, but from then on the object continues to go where it was sent.

The same thing happens with missiles in *Asteroids*, bombs and bullets in *Scramble*, the power 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 launch subroutine. A missile-figure (either a character or a sprite) is put on the screen a short distance away from the player-figure in the direction in which the player is shooting or throwing. Then the missile is moved across the screen as far as the program allows it to go.

The problem with missiles is that they use up time. Having a lot of them on the screen has the same effect as having a quantity of other characters or sprites. Everything that must be moved often slows down everything else. In machine language, this usually makes no difference. But in BASIC, the difference can be crucial.

One solution is to stop everything else while the missile fires. This is the way the laser beam is handled in *Gorf*. When the laser-firing ship is shooting, it does not move. It won't seem to slow down the game much, since things are as fast as when the missile is not being fired.

Another solution is the one we'll use in this example program. Program 13-1 is based on Program 5-7, which moved a series of playing card symbols (and other graphics characters) in a block back and forth across the screen. Now, by adding missiles, we can turn it into a game.

The story. You have been dealt a hand of 56 cards. The cards are moving back and forth across the screen, gradually coming down at you. Your job is to get them all with your dart before they can move down to the bottom row of the screen.

You can move your player-figure back and forth across the bottom of the screen by using the two cursor-movement keys. The UP/DOWN key moves you left; the LEFT/RIGHT key moves you right. Pressing the space bar launches your dart. You can retrieve your dart at any time just by launching another.

How the missile is handled. Both player movement and missile firing happen more often than other screen movements. They are placed in a loop-within-a-loop, so that missile movements and player movements take place many times for each time the rest of the screen display is moved. It gives the feeling of fast missile movement.

It also gives us a way of adding levels to a program. The game will be much easier when the player can move ten times and the missile can move ten steps between each movement of the deck of cards. Later, when the missile and player move only twice between each movement of cards, the player has to be absolutely magnificent to clear the screen. The game is just too fast for most players.

How the Program Works

Line	Explanation
10-90	**Initialization.** This is pretty much the same as the initialization in Program 5-7. D$ is the HOME and CURSOR-DOWN characters that place the player-figure, P$, on the correct row. The variable P controls the horizontal position of P$. The string TR$, for Top Row, contains the current score, the number of cards remaining, and the number of rows that the cards can move down before the player loses the game.
100	Begin the movement loop. (Since movement is handled entirely within FOR-NEXT loops, the program will reach the beginning of line 100 only when the game starts or the player has got rid of all the cards. Therefore, the GOSUB to 300, where the FG$ (*n*) string array is set up, takes place here. The same routine is used to set up each new block of cards.)
	Decrease the value of T by 1 each time through the loop. (We're adding a *true* value, which is the same as subtracting 1.)
	Begin the X, or row, or loop. (Each time through this

loop, the deck of cards drops down one row. When the cards have crossed the screen for the ninth time, the game ends.)

110 Begin the *I* loop. (This loop is identical to the loop at 150, except that the loop at 110 moves from left to right and the loop at 150 moves from right to left. The J loop PRINTs each line of the deck by PRINTing each FG$(*n*) string followed by a semicolon.)

115-120 After the J loop is complete, the K loop is executed T times. This loop contains player and missile movement. When T is a high number early in the game, player-figure and dart can both move T times for each time the deck of cards moves. At the lowest allowable value of T, 2, the player-figure and dart can only move twice for each movement of the cards. Set variable D to the direction of player movement and check for the edge of the player's movement range.

130 PRINT player-figure P$ at row D$ and column P. Then subtract 40 from M, the current address of the missile. To avoid extra arithmetic during the movement loop, M is always set to the absolute address of the missile. If the missile is off the screen, the value of M will be less than 1024, the start of screen memory.
Then, IF the space bar is pressed, jump to the missile launch subroutine at 400. (Notice that the missile address is decremented whether the missile is on the screen or not, and whether it is launched or not.)

135 If M>SM (the start of screen memory), the missile is on the screen. If it is, PEEK its new location and put what is there in the variable F . POKE the missile (MX) into screen memory and its color into color memory; erase the old missile position (M+40). Then, if F is the screen code of a graphics character, go to the collision subroutine at 500. If EG is set (1), it means all the cards have been hit; jump to the restart routine at 990. (The leftward movement loop is identical, except that the EG jump is to 995.)

400-410 **Missile Launch Subroutine**

400 Erase the last missile fired, if it is still onscreen. (Note that this allows you to fire a new missile before the

old one has reached the top of the screen. The dart keeps moving even after it has hit and removed one card. If the cards didn't move, it would be able to wipe out an entire column of cards at a time. Missile firing will seem sluggish at higher levels of play, but the player can always hold down the space bar until a missile appears sitting on top of the player-figure. Then releasing the space bar will cause the missile to launch.)

410 Set the horizontal position of the missile (MH) to P+1. (Since the player position marks the blank to the left of the player character, the actual horizontal position of the player-figure is P+1.) Set M to screen memory (SM) plus 840 (the row *above* the player-figure) plus MH. (That's all that it takes to launch the missile—movement is now handled automatically.)

500-570 **Collision-Handling Subroutine**

500 If the dart is in the top row, ignore the collision. Jump to 560 to update the score and return.

510 Calculate *which* character you hit by comparing it with each of the F$(*n*) characters. If the character collided with matches the character F$(3), the collision was in row FG$(3).

520 Calculate which character in the string was collided with by comparing MH with the horizontal position of the deck of cards. Set W to the length of the string FG$(*n*) to the left of the collision character, and set WW to the length of the string segment to the right of the collision character.

530 Reconstruct FG$(*n*) by setting it to everything left of the collision character plus a blank plus everything to the right of the collision character. (The string is now as long as it was before, but now the collision character is gone.)

540 Set the points and calculate the new value of L, which maintains the total of cards left.

550 If L now equals 0, the last card has been hit. Add the bonus to the score and set the EG flag to 1.

560 Update the string TR$ with the new score, row, and cards left.

600 **The Ending Routine.** This is where the program jumps if the cards have finished the bottom row

without the player getting all of them.

990-995 **Loop Breakout Routines.** These lines close each FOR-NEXT loop and jump to line 100 to refresh the screen and start a new deck of cards.

Program 13-1. Card Invaders

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
10 DIM FG$(7),F$(6),V$(9),P(8),PP(8)          :rem 27
20 B$=" ":BC=1:DI=1:T=8:SM=1024:CH=55296-SM:MX=30:
   B=32:EG=0                                  :rem 54
40 POKE 53281,BC:POKE 53280,BC:PRINT "{CLR}":POKE
   {SPACE}657,0                               :rem 104
60 V$(0)="{HOME}{BLU}":FOR I=1 TO 9:V$(I)=V$(I-1)+
   "{DOWN}":NEXT                              :rem 190
70 D$="{HOME}{22 DOWN}":P$="{BLK} {E} ":P=20
                                              :rem 114
80 PP=0:FOR I=0 TO 8:P(I)=50-(5*I):PP(I)=9-I:NEXT
                                              :rem 236
90 TR$="{HOME}"+STR$(PP)+" POINTS "+STR$(L)+" CARD
   S LEFT "+STR$(8-X)+" ROWS LEFT"            :rem 115
100 GOSUB 300:T=T+(T>0):FOR X=0 TO 8:PRINT "{CLR}"
                                              :rem 96
110 FOR I=0 TO 15:PRINT TR$V$(X)TAB(I);:FOR J=0 TO
     6:PRINT FG$(J);:NEXT J                   :rem 90
115 FOR K=0 TO T                              :rem 45
120 Q=PEEK(197):D=(Q=7)-(Q=2):P=P+D+((P>34 AND D=1
    )-(P<1 AND D=-1))                          :rem 1
130 PRINT D$TAB(P)P$:M=M-40:IF Q=60 THEN GOSUB 400
                                              :rem 23
135 IFM>SM THENF=PEEK(M):POKEM,MX:POKECH+M,4:POKEM
    +40,B:IFF>63 THENGOSUB 500                :rem 200
140 IF EG=1 THEN EG=0:GOTO 990                :rem 91
145 NEXT K:GOSUB 560:NEXT I                   :rem 59
150 FORI=14 TO 0 STEP-1:PRINT TR$V$(X)TAB(I);:FORJ
    =0 TO 6:PRINT FG$(J);:NEXT J              :rem 247
155 FOR K=0 TO T                              :rem 49
160 Q=PEEK(197):D=(Q=7)-(Q=2):P=P+D+((P>34 AND D=1
    )-(P<1 AND D=-1))                          :rem 5
170 PRINT D$TAB(P)P$:M=M-40:IF Q=60 THEN GOSUB 400
                                              :rem 27
180 IFM>SM THENF=PEEK(M):POKEM,MX:POKECH+M,4:POKEM
    +40,B:IFF>63 THENGOSUB 500                :rem 200
185 IF EG=1 THEN EG=0:GOTO 995                :rem 105
190 NEXT K:GOSUB 560:NEXT I:NEXT X:GOTO 600:rem 21
300 F$(0)="Q":F$(1)="S":F$(2)="A":F$(3)="Z":F$(4)=
    "X":F$(5)="{B}":F$(6)="£"                 :rem 119
```

```
305 FOR I=0 TO 6 STEP 2:FG$(I)="{BLU}":FG$(I+1)="
    {RED}":NEXT                                    :rem 152
310 FORI=0 TO 6:FG$(I)=FG$(I)+"{DOWN}"             :rem 151
320 FORJ=0 TO 7:FG$(I)=FG$(I)+B$+F$(I)+B$:NEXT:NEX
    T                                              :rem 93
330 FOR I=0 TO 5:FG$(I)=FG$(I)+"{16 SPACES}":NEXT
390 L=56:RETURN                                    :rem 170
400 IF M>984 THEN POKE M+40,32                      :rem 30
410 MH=P+1:M=SM+MH+840:RETURN                       :rem 24
500 IF M<1064 THEN 560                              :rem 66
510 F=F+128+64*(F>100):FOR II=0 TO 6:IF F=ASC(F$(I
    I))THEN F=II                                   :rem 72
520 NEXT II:W=MH-I+2:WW=41-W+16*(F=6)              :rem 114
530 FG$(F)=LEFT$(FG$(F),W)+B$+RIGHT$(FG$(F),WW)
                                                   :rem 89
540 PP=PP+INT(P(F)*PP(X)):L=L-1                    :rem 68
550 IF L=0 THEN L=56:PP=PP+1000*(8-T):EG=1         :rem 16
560 TR$="{HOME}"+STR$(PP)+" POINTS "+STR$(L)+" CAR
    DS "+STR$(8-X)+" ROWS{5 SPACES}"               :rem 79
570 RETURN                                         :rem 124
600 PRINT "{CLR}":FOR I=0 TO 6:PRINT "{RED}":PRINT
     F$(I):PRINT:NEXT                              :rem 220
610 PRINT STR$(PP)" POINTS":PRINT                   :rem 26
620 PRINT "PRESS A KEY TO PLAY AGAIN":PRINT:rem 32
630 FOR I=0 TO 3000                                :rem 156
640 IF PEEK(197)<64 THEN 660                       :rem 177
650 NEXT:PRINT "THANKS FOR PLAYING!":END           :rem 219
660 I=3000:NEXT I:PP=0:T=8:GOTO 100                :rem 245
990 K=T:NEXT:I=15:NEXT:X=8:NEXT:GOTO 100            :rem 26
995 K=T:NEXT:I=0:NEXT:X=8:NEXT:GOTO 100            :rem 233
```

You'll notice that there is no sound in this game. It would certainly be better with sound—feel free to add your own sound routines. Also, the game would be more enjoyable if the graphics symbols that are not hearts, diamonds, spades, or clubs were redefined as custom characters for kings, queens, and jacks. These could even be animated using character flipping.

It's often a good idea to practice game programming by modifying or building on a game like this one. You know that this games runs as is; you can now improve it and alter it until it exactly suits you. My guess is that by the time you got through altering the game, very little would be left of the original—it would have become your game entirely. But at every step along the way, you'd have a completed, working game to let you see how each change you've made works with the rest of the program.

Other Missiles

There are many other ways that missiles can behave. For instance, you could use a projectile that stays attached to the player-figure, like the air hose in *Dig-Dug*. This is done by simply not erasing each old missile position. Then, when the shot is finished, the entire line is erased all at once. While the hose is extended, the player-figure can't move.

Another possibility is a rebounding missile, like the power ball in *Mr. Do*. It keeps bouncing around until it hits a target or is retrieved by the player-figure.

Missiles can go a set distance and then fade, like the missiles in *Asteroids*. *Asteroids* also allows four missiles at once, which is easy enough in machine language, but very slow in BASIC.

You can also allow the player a limited number of missiles, the way *Missile Command* does. When the missiles are gone, they're gone, and the player becomes a helpless target.

You can allow computer-launched missiles to attack the player, forcing the player to dodge.

Missiles can be boomerangs, rebounding to the launch position where the player-figure must pick them up before it can launch again.

Missiles can be short extensions of the player-figure, like the kangaroo's arm and boxing glove in *Kangaroo* and the hammer in *Donkey Kong*.

But in all cases, missiles follow the same general pattern. The player's command launches the missile, the player-figure's location determines the missile's starting position and direction, and from then on the computer controls missile movement like any other programmed object on the screen.

14
Your First Game—and Mine

Your First Game—and Mine

We've been through all the principles of programming arcade games in BASIC on the 64. Using other reference books, you'll be able to pick up some more sophisticated 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 carefully, leaving plenty of line numbes for adding new subroutines to improve the game, your first game will turn out pretty well. And with patience, you'll gain skill and confidence.

But no matter how good you eventually become, you'll probably always feel proud of your first few games. I know I feel that way about the games I first designed and wrote. The two games in this chapter, "Crusade" and "Magneto," were the result of trying out techniques I'd first learned on the VIC. The 64 is so different, though, that my initial versions were primitive, and didn't fully use the 64's abilities. So I changed and rewrote and debugged these games until they worked to my satisfaction. I finally settled on the versions you see here.

There are some features for each of these games that you may find interesting, and which also show how I learned to program games in BASIC.

Crusade

Crusade, for instance, is an all-BASIC game. I didn't even use a machine language subroutine to move the player-controlled character around the screen. I wanted to see what I could do using only BASIC. I think you'll agree that it worked out well.

Writing games in BASIC often means walking a thin line between speed and complexity. Games like *Defender* or *Robotron* cannot be written in BASIC. The games need to move much faster than BASIC allows. Other games, such as *Pac-Man* and *Berserk*, are a better model for those of us who are working solely in BASIC. For instance, this game began with an idea to include a fair amount of missile firing. The player's character would fire missiles at the infidels; the infidels would fire back. But I discovered while programming that BASIC could not

execute all this and still keep playing at a reasonable speed. Once I realized that, I eliminated the feature and made the game what it is now. It became playable.

The flashy title screen was harder to program than it was to think of. The letters of "Crusade" are spelled out in silhouette, and each row was assigned to an array variable and then PRINTed out in a different color each time to create the rainbow effect.

In my attempt to make the game play faster, I discovered something. By moving everything that fell before the main loop to a far-distant subroutine, I increased the game's speed. This increased the speed of the backward GOTOs, which have to hunt from the beginning of the program forward to find the line they're going to. I also sped up the game by making all the constants in the main loop into variables defined at the beginning of the program. Although this reduces the readability of the program, I found it to be the fastest and easiest way of speeding up a game.

The program is long. There are only about 100 bytes free while the program executes. I could have relocated some things to another 16K bank of memory, but I wanted to see if I could write the game using only one video bank. I did it, but it was close. The graphic characters are stored in the top 2K block of the bank. Most were copied from the ROM character set (see Chapter 4 for explanations on how to do this), but there are some custom characters.

I used sprites for the infidel figures. Sprites worked well here, for I could animate them by blinking lights on and off in their chests. A single POKE of new values into the sprite DATA block was all it took. Since all the infidels share the same sprite DATA, they appear identical. This is another way to save memory space if you find yourself running short, as I did.

I also included a pair of convenience controls in the game. Hitting the f1 key during the main part of the game will reset the program (although it will take only a moment for the game to be ready again). Pressing the f7 key stops the program (and the time clock,too), so that you can analyze the game, or take a break from it.

This was one of my first games on the 64. I hope you like it. Maybe I'll get to play yours someday.

Program 14-1. Crusade

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
100 POKE56,53:POKE55,128:CLR                    :rem 116
110 GOSUB1290                                   :rem 222
120 GOSUB1340                                   :rem 219
130 GOSUB990                                    :rem 182
140 :                                           :rem 207
150 POKEV0,SX:POKEV1,SY                         :rem 234
160 A=PEEK(V4)+PEEK(V5):IF(PEEK(V4)ANDZ1)+(PEEK(V5
    )ANDZ1)THEN860                               :rem 97
170 GETA$:IFA$=""THEN230                         :rem 80
180 IFA$="{F1}"THENPOKE53269,0:GOTO120           :rem 92
190 IFA$<>"{F7}"THEN230                         :rem 152
200 T$=TI$                                        :rem 8
210 GETA$:IFA$=""THEN210                         :rem 73
220 TI$=T$                                       :rem 10
230 K=ZB-PEEK(JS):IFK=Z0THEN330                 :rem 153
240 IFKANDZATHEN500                             :rem 162
250 IFKANDZ1THENSY=SY-Z4:GOTO270                 :rem 90
260 IFKANDZ2THENSY=SY+Z4                         :rem 78
270 IF(KANDZ4)=Z0THEN300                        :rem 174
280 SX=SX-Z4:IFDR=Z0THENSX=SX-Z6:POKEV0,SXANDZD:DR
    =Z4                                          :rem 95
290 GOTO320                                     :rem 105
300 IF(KANDZ8)=Z0THEN320                        :rem 174
310 SX=SX+Z4:IFDR=Z4THENSX=SX+Z6:POKEV0,SXANDZD:DR
    =Z0                                          :rem 85
320 POKEPN,SB+WK+DR:WK=(WK+Z1)ANDZ3             :rem 211
330 I=PEEK(V2):IFSX>ZDTHENSX=SX-ZE:I=IORZ1:POKEV2,
    I:POKEV0,SX                                  :rem 89
340 IFSX<Z0THENSX=SX+ZE:I=IANDZC:POKEV2,I:POKEV0,S
    X                                           :rem 200
350 IFSY<M0ORSY>M1OR(SX<M2AND(IANDZ1)=Z0)OR(SX>M3A
    ND(IANDZ1))THEN580                          :rem 113
360 IFLRTHENLR=LR+1+15*(LR>14):POKE53294,LR:rem 81
370 POKE14223+BL,72:BL=BL+3:IFBL>6THENBL=0:rem 249
380 POKE14223+BL,75                              :rem 23
390 FORI=Z2TO12STEPZ2:IFAL%(RM,I/Z2)=Z0THEN440
                                                :rem 140
400 J=PEEK(V0+I):J=J+SR*SGN((PEEK(V2)ANDZ1)*ZE+SX-
    J):K=PEEK(V1+I)                             :rem 191
410 K=K+SR*SGN(SY-K)                            :rem 191
420 IFJ<ZEANDJ>M2THENPOKEV0+I,J                  :rem 242
430 IFK<M1ANDK>M0THENPOKEV1+I,K                  :rem 212
440 NEXT:IFPEEK(V4)ANDZ1THEN860                 :rem 203
450 IFPEEK(V4)+PEEK(V5)=Z0THEN150               :rem 224
```

```
460 FORI=Z1TOZ6:IF((PEEK(V4)ORPEEK(V5))ANDZ2↑I)=Z0
    THENNEXT:GOTO150                          :rem 22
470 POKE53287+I,1:FORJ=1TO100:NEXT:POKE53287+I,I
                                              :rem 133
480 AL%(RM,I)=Z0:POKEV3,PEEK(V3)ANDZD-Z2↑I:NEXT:GO
    TO150                                     :rem 234
490 :                                         :rem 215
500 IFCS=0THEN390                             :rem 242
510 CS=CS-1:PRINT"{HOME}"TAB(10)CS$(CS)" ":rem 233
520 POKE53285,1:POKE54273,10:POKE54277,11:POKE5427
    8,0:FORI=1TO3                             :rem 162
530 POKE54276,8:POKE54276,129:FORJ=4TO0STEP-1:POKE
    53280,CL(J):POKE53281,CL(J)               :rem 56
540 FORK=0TO50:NEXT:NEXT:FORK=0TO99:NEXT:NEXT:POKE
    54276,8                                   :rem 231
550 FORI=1TO6:AL%(RM,I)=0:POKE53269,PEEK(53269)AND
    (255-2↑I)                                 :rem 178
560 FORJ=1TO100:NEXT:NEXT:POKE53285,12:GOTO390
                                              :rem 110
570 :                                         :rem 214
580 IF(SY<M0ANDD%(RM,1)=0)OR(SY>M1ANDD%(RM,1)=1)TH
    EN610                                     :rem 239
590 IFSX<M2ANDD%(RM,1)=2THEN610               :rem 81
600 IFSX<M3OR(PEEK(V2)ANDZ1)<>Z1ORD%(RM,1)<>3THEN1
    30                                        :rem 166
610 PRINT"{HOME}{7 DOWN}"TAB(11)"THERE IS NO RETUR
    N!{UP}":FORJ=1TO500:NEXT                  :rem 141
620 PRINTTAB(11)"{19 SPACES}":IFD%(RM,1)AND2THENSX
    =SX(D%(RM,1)):GOTO640                     :rem 77
630 SY=SY(D%(RM,1))                           :rem 53
640 IFD%(RM,1)>1THENDR=-4*(D%(RM,1)=3):POKE2040,SB
    +WK+DR                                    :rem 208
650 GOTO150                                   :rem 106
660 :                                         :rem 214
670 PRINT"{CLR}{7 DOWN}"TAB(14)"YOU HAVE WON!":T$=
    TI$:T=TI                                  :rem 151
680 POKE53294,8:POKE53248,PEEK(53248)-11:POKE53249
    ,PEEK(53249)-6                            :rem 90
690 POKE53262,PEEK(53262)-11:POKE53263,PEEK(53263)
    -6                                        :rem 119
700 POKE53271,129:POKE53277,129               :rem 201
710 FORI=1TO5000:NEXT:PRINT"{CLR}":POKEV3,0:POKE53
    270,8:POKE53272,23                        :rem 206
720 POKE53280,12:POKE53281,12                 :rem 84
730 IFT<MTTHENMT=T:MT$=T$:MR=RR               :rem 41
740 PRINTTAB(15)"{WHT}{3 DOWN}{9 @}":PRINTTAB(15
    )"{RVS} CRUSADE "                         :rem 40
750 A$=T$:GOSUB830:PRINTTAB(5)"{3 DOWN}{BLU}YOUR T
    IME (FOR"RR"ROOM";                        :rem 50
```

```
760 IFRR<>1THENPRINT"S";                          :rem 9
770 PRINT") WAS{DOWN}[43]":PRINTA$                :rem 87
780 A$=MT$:GOSUB830:PRINTTAB(4)"{3 DOWN}{RED}BEST
    {SPACE}TIME YET (FOR"MR"ROOM";                :rem 74
790 IFMR<>1THENPRINT"S";                          :rem 7
800 PRINT") IS{DOWN}[43]":PRINTA$                 :rem 2
810 FORI=1TO7500:IFPEEK(JS)AND16THENNEXT          :rem 42
820 GOTO120                                       :rem 102
830 B$=A$:A$=" "+LEFT$(B$,2)+" HOURS, "+MID$(B$,3,
    2)+" MINUTES, AND "                           :rem 174
840 A$=A$+RIGHT$(B$,2)+" SECONDS.":RETURN         :rem 75
850 :                                             :rem 215
860 FORI=1TO200:NEXT:IF(PEEK(V4)AND128)=128THEN670
                                                  :rem 22
870 POKE54272,10:POKE54277,0:POKE54278,245:POKE542
    75,1:POKE54276,65                             :rem 60
880 FORI=1TO100:NEXT:POKE54276,64                 :rem 242
890 FORI=1TO161:POKE53287,I:NEXT:FORI=3TO0STEP-1:P
    OKE53287,CL(I):FORJ=0TO400                    :rem 247
900 NEXT:NEXT:POKEV3,0:POKE53280,1:POKE53281,1:RS=
    RS-1:IFRS<0THEN940                            :rem 245
910 POKE53282,1:POKE53283,1:POKE53287,1           :rem 196
920 PRINT"{HOME}"TAB(37)RS:FORI=0TO48:POKE53280,I:
    POKE53281,I:FORJ=1TO50:NEXT:NEXT              :rem 48
930 DR=-4*((D%(RM,1)AND1)=1):GOSUB1220:GOTO150
                                                  :rem 76
940 POKE53280,0:POKE53281,0                       :rem 242
950 PRINT"{CLR}{WHT}{6 DOWN}"TAB(11)"ALAS, THE BLE
    SSING":PRINTTAB(13)"IS NO MORE ...            :rem 43
960 FORI=1TO7500:IFPEEK(JS)AND16THENNEXT          :rem 48
970 PRINT"{CLR}":POKE53270,8:POKE53272,21:GOTO120
                                                  :rem 211
980 :                                             :rem 219
990 T$=TI$:RM=RM+1:POKEV3,0                       :rem 119
1000 POKE53280,1:POKE53281,1:PRINT"{CLR}":POKE5327
     2,23                                         :rem 179
1010 PRINT"{11 DOWN}"TAB(5)"{WHT}** PREPARE TO ENT
     ER ROOM"RM"**                                :rem 74
1020 FORI=4TO0STEP-1:POKE53281,CL(I):POKE53280,CL(
     I):FORJ=1TO200:NEXT:NEXT                     :rem 82
1030 POKE54280,0:POKE54281,0:POKE54282,244:POKE542
     82,1:POKE54283,65                            :rem 32
1040 FORI=1TO39:PRINT"{UP} "CHR$(20):FORJ=1TO50:NE
     XT:POKE54280,100-I*2:NEXT                    :rem 144
1050 PRINT"{HOME}"TAB(11)"{11 DOWN}{BLK}PRAY FOR Y
     OUR SOUL.{WHT}":POKE54283,0                  :rem 246
1060 FORI=0TO4:POKE53281,CL(I):POKE53280,CL(I):FOR
     J=1TO200:NEXT:NEXT                           :rem 188
```

```
107Ø PRINT"{HOME}{WHT} ROOM"RMTAB(1Ø)CS$(CS)TAB(24
     )"RESURRECTIONS"RS;                          :rem 97
108Ø PRINTSPC(16)LEFT$(T$,2)":"MID$(T$,3,2)":"RIGH
     T$(T$,2)                                     :rem 179
109Ø POKE53282,1:POKE53283,1:FORI=ØTO11:IFRM%(RN%(
     RM),I)=ØTHENNEXT:GOTO112Ø                    :rem 27
11ØØ A=WS%(I):POKEWB%(I)-A,69:POKEWE%(I)+A,69
                                                  :rem 62
1111 FORJ=WB%(I)TOWE%(I)STEPA:POKEJ,65-(A>1):POKEJ
     +54272,9:NEXT:NEXT                           :rem 181
112Ø FORI=ØTO39:POKE11Ø4+I,64:POKE55376+I,9:POKE19
     84+I,64:POKE56256+I,9:NEXT                    :rem 1
113Ø FORI=4ØTO84ØSTEP4Ø:POKE11Ø4+I,64:POKE55376+I,
     9:POKE1143+I,64:POKE55415+I,9                :rem 124
114Ø NEXT:J=D%(RM,1):FORI=DB%(J)TODE%(J)STEPDS%(J)
     :POKEI,67-(DS%(J)>1)                         :rem 119
115Ø POKEI+54272,9:NEXT                           :rem 76
116Ø J=D%(RM,2):FORI=DB%(J)TODE%(J)STEPDS%(J):POKE
     I,67-(DS%(J)>1):POKEI+54272,9                :rem 71
117Ø NEXT:SR=INT(1.5+RM/2Ø*3):PRINT"{HOME}"TAB(11)
     "{11 DOWN}{19 SPACES}"                       :rem 83
118Ø POKE53272,31:POKE54291,Ø:POKE54292,25Ø:POKE54
     287,5Ø:POKE54299,129                         :rem 193
119Ø POKE54299,128:FORI=4TOØSTEP-1:POKE53281,CL(I)
     :POKE53280,CL(I)                             :rem 142
12ØØ FORJ=1TO2ØØ:NEXT:NEXT:TI$=T$                 :rem 57
121Ø :                                            :rem 254
122Ø J=1:LR=Ø:IFRM=RRTHENJ=129:POKE53262,178:POKE5
     3263,15Ø:LR=1                                 :rem 8
123Ø FORI=1TO6:POKEVØ+I*2,RND(1)*215+4Ø:POKEV1+I*2
     ,RND(1)*13Ø+75                               :rem 134
124Ø IFAL%(RM,I)THENJ=J+2↑I                       :rem 113
125Ø NEXT:K=D%(RM,1):POKEV2,(PEEK(V2)ANDZC)-(K=3):
     SX=SX(K):SY=SY(K)                            :rem 1Ø4
126Ø WK=Ø:POKEVØ,SX:POKEV1,SY:POKEPN,SB+DR:POKEV3,
     J                                             :rem 87
127Ø POKE53283,(RMAND3)+1:POKE53282,SR:RETURN
                                                  :rem 212
128Ø :                                             :rem 5
129Ø K=Ø:I=Ø:J=Ø:SX=Ø:SY=Ø:DR=Ø:WK=Ø:BL=Ø:REM THIS
      IS TO ORDER THE VAR.TABLE                   :rem 54
13ØØ ZØ=Ø:Z1=1:Z2=2:Z3=3:Z4=4:Z6=6:Z8=8:ZA=16:ZB=1
     27:ZC=254:ZD=255:ZE=256                      :rem 74
131Ø PN=2Ø4Ø:VØ=53248:V1=53249:V2=53264:V3=53269:V
     4=53278:V5=53279:JS=5632Ø                    :rem 164
132Ø SB=214:MØ=66:M1=229:M2=16:M3=69:MT=9999999:GO
     SUB2Ø4Ø:POKE54296,15                         :rem 16
133Ø DIM AL%(4Ø,6),D%(4Ø,2),RM%(7,11),RN%(4Ø),WB%(
     11),WE%(11),WS%(11):RETURN                   :rem 195
```

```
1340 PRINT"{CLR}":POKE53280,0:POKE53281,0:POKE5327
     2,21:POKE53271,0:POKE53277,0          :rem 74
1350 POKE53270,8:A$(0)=" {RVS}M [*]{RIGHT}M
     {3 SPACES}[*]{RIGHT}M[*]{2 RIGHT}M[*]
     {RIGHT}M{2 SPACES}[*]{2 RIGHT}M{2 SPACES}
     [*]{RIGHT}M{3 SPACES}[*]{RIGHT}M
     {3 SPACES}[*]"                        :rem 117
1360 A$(1)="MM{2 RIGHT}[*] {4 RIGHT}[*]
     {3 RIGHT} {RIGHT}MM{3 RIGHT}[*]MM{3 RIGHT}
     [*] {4 RIGHT}[*] {OFF}{4 SPACES}"     :rem 49
1370 A$(2)="{RVS} {2 RIGHT}{OFF}[*]{RVS}{RIGHT}
     {SPACE}{3 RIGHT} {RIGHT} {3 RIGHT} {RIGHT}
     {3 RIGHT}{OFF}[*]{RVS}{RIGHT} {3 RIGHT}
     {RIGHT} {3 RIGHT} {RIGHT} {OFF}{4 SPACES}"
                                           :rem 121
1380 A$(3)="{RVS} {4 RIGHT} {RIGHT}{2 SPACES}M
     {RIGHT} {3 RIGHT} {RIGHT}{OFF}[*]{RVS}
     {RIGHT}{2 SPACES}[*]{RIGHT} {RIGHT}
     {2 SPACES}M{RIGHT} {3 RIGHT} {RIGHT} {RIGHT}
     {SPACE}[*]{OFF} "                     :rem 220
1390 A$(4)="{RVS} {4 RIGHT} {5 RIGHT} {3 RIGHT}
     {5 RIGHT} {RIGHT} {3 RIGHT} {RIGHT} {3 RIGHT}
      {RIGHT} {OFF}{4 SPACES}"             :rem 206
1400 A$(5)="{RVS} {2 RIGHT}M[*] {3 RIGHT}
     {RIGHT} {3 RIGHT} {RIGHT}M[*]{2 RIGHT}
     {RIGHT} {3 RIGHT} {RIGHT} {3 RIGHT} {RIGHT}
     {OFF}{4 SPACES}"                      :rem 142
1410 A$(6)="[*]{RVS}{RIGHT} M{RIGHT} {3 RIGHT}
     {RIGHT}{OFF}[*]{RIGHT}{RVS}{2 SPACES}M
     {RIGHT}{OFF}[*]{RIGHT}{RVS}{2 SPACES}M
     {RIGHT} {3 RIGHT} {RIGHT} {RIGHT}{2 SPACES}M
     {RIGHT} {RIGHT}{2 SPACES}[*]"         :rem 42
1420 A$(7)="{OFF} [*]{3 SPACES}[*]{3 SPACES}
     [*]{2 SPACES}[*]{5 SPACES}[*]{4 SPACES}
     [*]{3 SPACES}[*] [*]{5 SPACES}[*]"
                                           :rem 158
1430 CL$="{RED}[1]{YEL}{GRN}{CYN}{BLU}{PUR}{WHT}
     "                                     :rem 180
1440 FORT=1TO30:PRINT"{HOME}{6 DOWN}"      :rem 252
1450 FORLN=0TO7:PRINTMID$(CL$,CL+1,1)A$(LN);:CL=(C
     L+1)AND7:NEXT:CL=(CL+1)AND7           :rem 107
1460 IFPEEK(JS)AND16THENNEXT                :rem 21
1470 PRINT"{HOME}{6 DOWN}{WHT}":FORI=0TO7:PRINTA$(
     I);:NEXT                              :rem 70
1480 CL(1)=11:CL(2)=12:CL(3)=15:CL(4)=1     :rem 21
1490 FORI=1TO4:POKE53280,CL(I):POKE53281,CL(I):FOR
     J=1TO200:NEXT:NEXT                    :rem 196
1500 PRINT"{CLR}{WHT}":POKE53272,23         :rem 44
```

```
1510 CR$="-------- CRUSADE --------":PRINTTAB(7)CR
     $                                    :rem 55
1520 PRINT"{DOWN}{4 SPACES}YOU ARE ON A CRUSADE TO
      RECOVER THE"                        :rem 126
1530 PRINT"CROSS OF GOLD, REVERED FOR CENTURIES BY
     "                                    :rem 76
1540 PRINT"YOUR ANCESTORS, BUT STOLEN BY INFIDELS.
     "                                    :rem 159
1550 PRINT"YOU MUST FIND YOUR WAY THROUGH THEIR"
                                          :rem 96
1560 PRINT"CATACOMBS, AVOIDING THEIR TOUCH AT ALL"
                                          :rem 20
1570 PRINT"COSTS, UNTIL YOU FIND THE CROSS."
                                          :rem 153
1580 PRINT"{DOWN}{4 SPACES}THEIR TOUCH SPELLS DOOM
      TO YOU, BUT                         :rem 198
1600 PRINT"YOU CAN USE THE POWERS GRANTED YOU TO"
                                          :rem 186
1610 PRINT"MAKE THESE BEINGS VANISH.{2 SPACES}YOU
     {SPACE}CAN USE"                      :rem 27
1620 PRINT"THIS POWER ONLY THREE TIMES.{2 SPACES}Y
     OUR"                                 :rem 152
1630 PRINT"POWER ALSO BREATHES LIFE INTO YOU,"
                                          :rem 40
1640 PRINT"RAISING YOU FROM THE DEAD THREE TIMES."
                                          :rem 241
1650 PRINT"{DOWN}{4 SPACES}TO USE YOUR POWERS, SIM
     PLY"                                 :rem 247
1660 PRINT"DEPRESS THE BUTTON ON JOYSTICK 2."
                                          :rem 222
1665 PRINT"{DOWN}{5 SPACES}MAY YOUR CRUSADE GO WEL
     L."                                  :rem 105
1670 FORI=4TO0STEP-1:POKE53280,CL(I):POKE53281,CL(
     I):FORJ=1TO200:NEXT:NEXT             :rem 93
1680 POKE54277,0:POKE54278,15*16+4:POKE54272,0
                                          :rem 33
1690 PRINT"{DOWN}{2 SPACES}NUMBER OF ROOMS (JOYSTI
     CK) --":RR=20                        :rem 56
1700 POKE214,21:PRINT:B$=RIGHT$("0"+MID$(STR$(RR),
     2),2):PRINTTAB(32)B$                 :rem 42
1710 IFNOTPEEK(JS)AND16THEN1760           :rem 147
1720 IFNOTPEEK(JS)AND4THENRR=RR+(RR>1):GOTO1750
                                          :rem 231
1730 IFNOTPEEK(JS)AND8THENRR=RR-(RR<40):GOTO1750
                                          :rem 31
1740 PRINT"{HOME}{DOWN}"TAB(7)MID$(CL$,CL+1,1)CR$:
     CL=(CL+1)AND7:GOTO1700               :rem 214
1750 POKE54276,8:POKE54273,50+RR*2:POKE54276,17:PO
     KE54276,16:GOTO1740                  :rem 218
```

```
1760 POKE54278,249:POKE54276,8:POKE54273,10:POKE54
     276,129:POKE54276,128                    :rem 19
1770 PRINT"{UP}{WHT}"TAB(32)B$"{HOME}{DOWN}"TAB(7)
     CR$                                      :rem 169
1780 FORJ=1TO6:FORI=1TORR:AL%(I,J)=1:NEXT:POKEPN+J
     ,222:POKE53287+J,J:NEXT                  :rem 94
1790 POKEPN+7,223:POKE53287+7,7:POKE53287,1
                                              :rem 201
1800 RESTORE:FORI=0TO7:FORJ=0TO11:READRM%(I,J):NEX
     T:NEXT                                   :rem 151
1810 D%(0,2)=3:DR=0:FORI=1TORR:RN%(I)=INT(RND(1)*8
     )                                        :rem 184
1820 D%(I,1)=(D%(I-1,2)AND254)OR(NOTD%(I-1,2)AND1)
                                              :rem 140
1830 D%(I,2)=INT(RND(1)*4):IFD%(I,1)=D%(I,2)THEN18
     30                                       :rem 44
1840 NEXT:D%(RR,2)=D%(RR,1):FORI=0TO3:READDB%(I),D
     E%(I),DS%(I):NEXT                        :rem 177
1850 FORI=0TO11:READWB%(I),WE%(I),WS%(I):NEXT
                                              :rem 179
1860 FORI=0TO3:READSX(I),SY(I):NEXT:POKE53270,24:P
     OKE53283,1:POKE53284,1:RM=1              :rem 162
1870 RS=3:CS=3:POKE53280,11:POKE53281,11:POKE53276
     ,254:POKE53275,255                       :rem 134
1880 FORI=1TORR:FORJ=1TO6:AL%(I,J)=1:NEXT:FORI=0TO
     3:READCS$(I):NEXT                        :rem 27
1890 IFPEEK(13697)<>96ORPEEK(14337)<>102ORPEEK(148
     48)<>170THEN1910                         :rem 28
1900 FORI=1TO3:POKE53280,CL(I):POKE53281,CL(I):FOR
     J=1TO200:NEXT:NEXT:GOTO1990               :rem 5
1910 POKE56334,PEEK(56334)AND254:POKE53274,0
                                              :rem 222
1920 POKE53280,12:POKE53281,12:SP=13696       :rem 170
1930 READA:IFA<0THENFORI=1TO-A:POKESP,0:SP=SP+1:NE
     XT:GOTO1930                              :rem 184
1940 IFA<ZETHENPOKESP,A:SP=SP+1:GOTO1930       :rem 64
1950 POKE53280,15:POKE53281,15                :rem 144
1960 POKE1,51:C1=55296:C2=14336               :rem 165
1970 FORI=0TO511:POKEC2+I,PEEK(C1+I):NEXT:POKE1,55
     :POKE56334,PEEK(56334)OR1                :rem 20
1980 C2=14848:FORI=0TO47:READA:POKEC2+I,A:NEXT
                                              :rem 65
1990 POKE53285,12:POKE53286,1                 :rem 102
2000 POKE657,128:POKE53272,31:POKE53280,1:POKE5328
     1,1                                      :rem 231
2010 POKE54273,20:POKE54277,0:POKE54278,247:POKE54
     280,10:POKE54284,0                       :rem 84
2020 POKE54285,247:TI$="000000":RM=0:RETURN
                                              :rem 192
```

```
2030 :                                                  :rem 255
2040 FORI=0TO23:POKE54272+I,0:NEXT                      :rem 33
2050 POKE54276,8:POKE54283,8:POKE54290,8:RETURN
                                                        :rem 32
2060 DATA 1,0,0,1,0,0,0,0,1,0,1,0                       :rem 10
2070 DATA 0,0,0,1,0,0,1,1,0,0,0,0                       :rem 10
2080 DATA 1,1,0,0,0,0,0,0,0,0,1,1                       :rem 12
2090 DATA 1,0,0,0,0,0,1,1,1,0,0,0                       :rem 13
2100 DATA 0,0,0,1,0,0,0,0,1,0,0,0                        :rem 3
2110 DATA 0,0,0,0,0,1,1,0,1,0,0,0                        :rem 5
2120 DATA 0,0,0,0,0,1,0,0,0,1,1,0                        :rem 6
2130 DATA 1,0,0,0,0,1,1,0,0,0,0,1                        :rem 8
2140 DATA 1120,1127,1,2000,2007,1,1424,1664,40,146
     3,1703,40,1152,1352,40                             :rem 186
2150 DATA 1175,1375,40,1385,1391,1,1393,1414,1,141
     6,1422,1,1432,1672,40                              :rem 171
2160 DATA 1455,1695,40,1705,1711,1,1713,1734,1,173
     6,1742,1,1752,1952,40                              :rem 187
2170 DATA 1775,1975,40                                  :rem 74
2180 DATA 173,72,173,226,31,147,59,147                  :rem 92
2190 DATA " NO POWERS"," ONE POWER"," TWO POWERS",
     "THREE POWERS{2 SPACES}"                           :rem 152
2200 DATA 0,96,0,0,208,0,0,240,0,0,224,0,0,192,0,1
     ,192,0,3,224,0,3,60,0,3,0,0                        :rem 93
2210 DATA 3,0,0,3,128,0,6,192,0,12,96,0,24,48,0,48
     ,48,0,112,56,-17                                   :rem 130
2220 DATA 0,96,0,0,208,0,0,240,0,0,224,0,0,192,0,1
     ,192,0,3,224,0,3,60,0,3,0,0                        :rem 95
2230 DATA 3,0,0,07,128,0,12,192,0,12,192,0,24,96,0
     ,24,96,0,28,112,-17                                :rem 17
2240 DATA 0,96,0,0,208,0,0,240,0,0,224,0,0,192,0,1
     ,192,0,3,224,0,3,48,0,3,24                         :rem 65
2250 DATA 0,3,0,0,3,128,0,3,192,0,3,128,0,3,0,0,7,
     0,0,15,128,-17                                     :rem 253
2260 DATA 0,96,0,0,208,0,0,240,0,0,224,0,0,192,0,1
     ,192,0,3,224,0,3,60,0,3,0,0                        :rem 99
2270 DATA 3,0,0,7,128,0,12,192,0,12,192,0,24,96,0,
     24,96,0,28,112,-17                                 :rem 229
2280 DATA 0,6,0,0,11,0,0,15,0,0,7,0,0,3,0,0,3,128,
     0,7,192,0,60,192,0,0,192,0,0                       :rem 141
2290 DATA 192,0,1,192,0,3,96,0,6,48,0,12,24,0,12,1
     2,0,28,14,-16                                      :rem 234
2300 DATA 0,6,0,0,11,0,0,15,0,0,7,0,0,3,0,0,3,128,
     0,7,192,0,60,192,0,0,192,0,0                       :rem 134
2310 DATA 192,0,1,224,0,3,48,0,3,48,0,6,24,0,6,24,
     0,14,56,-16                                        :rem 131
2320 DATA 0,6,0,0,11,0,0,15,0,0,7,0,0,3,0,0,3,128,
     0,7,192,0,12,192,0,24,192,0                        :rem 95
2330 DATA 0,192,0,1,192,0,3,192,0,1,192,0,0,192,0,
     0,224,0,1,240,-16                                  :rem 150
```

```
2340 DATA 0,6,0,0,11,0,0,15,0,0,7,0,0,3,0,0,3,128,
     0,7,192,0,60,192,0,0,192,0                :rem 46
2350 DATA 0,192,0,1,224,0,3,48,0,3,48,0,6,24,0,6,2
     4,0,14,56,-16                             :rem 227
2360 DATA 3,0,0,15,192,0,63,240,0,15,192,0,106,164
     ,0,72,132,0,72,132,0,72,132               :rem 146
2370 DATA 0,69,68,0,10,128,0,8,128,0,8,128,0,8,128
     ,0,8,128,0,20,80,0,20,80,-17              :rem 212
2380 DATA 0,40,0,0,40,0,0,235,0,2,170,128,2,150,12
     8,2,170,128,0,235,0,0,40                  :rem 236
2390 DATA 0,0,40,0,0,40,0,0,40,0,0,40,0,0,40,0,0,4
     0,0,0,40,-19,256                           :rem 64
2400 DATA 170,170,170,170,170,170,170,170,0,0,0,17
     0,170,0,0,0                               :rem 132
2410 DATA 40,40,40,40,40,40,40,40,0,0,0,85,85,0,0,
     0                                         :rem 143
2420 DATA 4,4,4,4,4,4,4,255,255,255,255,255,255,
     255,255                                   :rem 246
```

Magneto

This game uses somewhat different programming techniques. It's not a completely BASIC game, for the multicolored player-controlled character is moved with a machine language routine. The keyboard routine you saw in Chapter 8 is the one I POKEd into the cassette buffer to move the character. I found uses for some things in this game that I didn't in *Crusade*, such as stationary sprites, random sounds, ring modulation, and multicolored characters. It made it more interesting as I programmed when I tried to find ways to fit these things in.

This game gets more difficult as you play, just as *Crusade* does. I sped up the play of the game by using some of the same methods found in *Crusade*. Accessing subroutines at the very beginning of the program and using variables instead of constants are two ways to speed up BASIC's execution. Modularizing the program also helped, for I could work on one aspect of the game, then add it to the main program before moving on to the next piece. That way, it seemed like I was making progress as I wrote the game, and gave me some incentive to continue. It's a tip you might want to try yourself.

The program itself is pretty easy to understand and follow. It uses a lot of the techniques and methods you've seen in the book, and puts them together to create a complete game. It's something I enjoy playing, and I hope you will, too.

Program 14-2. Magneto

Remember, do not type the checksum number at the end of each line. For example, do not type ":rem 123." Please read the article about the "Automatic Proofreader" in Appendix E.

```
4 GOSUB1000:PRINT"{BLU}{HOME}{23 DOWN}"SPC(22);:FO
  RX=OTO13:PRINT"N{UP}";                          :rem 192
10 PRINT "{CLR}" : GOSUB 10000: GOSUB 5000: GOSUB
   {SPACE}6000: GOSUB 4700                        :rem 244
15 D = 5: GD = 25: DL = 1: SL = 5                  :rem 8
20 PRINT "{CLR}" : GOSUB 5500: GOSUB 4000 :rem 200
40 GOSUB 1010: GOSUB 200: GOTO 210                 :rem 248
45 S = 54272: FOR X = 0 TO 24: POKE S + X, 0: NEXT
   : RETURN                                        :rem 88
50 GOSUB 1000: PRINT"{YEL}{HOME}{10 DOWN}{4 RIGHT}
   {32 Y}": RETURN                                 :rem 122
55 PRINT"{YEL}{HOME}{10 DOWN}{4 RIGHT}{32 SPACES}"
   : RETURN                                        :rem 36
60 GOSUB 1000:PRINT"{BLU}{HOME}{18 DOWN}{4 RIGHT}D
   DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD"              :rem 10
65 RETURN                                          :rem 75
70 PRINT"{BLU}{HOME}{18 DOWN}{4 RIGHT}{32 SPACES}"
   : RETURN                                        :rem 42
75 GOSUB 1000:PRINT"{RED}{HOME}{7 DOWN}"SPC(Q);:FO
   RX=OTOW:PRINT"{J}{DOWN}{LEFT}";:NEXT:RETURN
                                                   :rem 92
80 PRINT"{RED}{HOME}{7 DOWN}"SPC(Q);:FORX=OTOW:PRI
   NT" {DOWN}{LEFT}";:NEXT:RETURN                  :rem 40
85 GOSUB 1000:PRINT"{YEL}{HOME}{7 DOWN}"SPC(Z2);:F
   ORX=OTOW:PRINT"{N}{DOWN}{LEFT}";:NEXT:RETURN
                                                   :rem 15
90 PRINT"{YEL}{HOME}{7 DOWN}"SPC(Z2);:FORX=OTOW:PR
   INT" {DOWN}{LEFT}";:NEXT:RETURN                 :rem 230
95 GOSUB 1000:PRINT"{PUR}{HOME}{10 DOWN}{4 RIGHT}"
   ;:FORX=OTOU:PRINT"M{DOWN}";:NEXT:RETURN:rem 118
100 PRINT"{PUR}{HOME}{10 DOWN}{4 RIGHT}";:FORX=OTO
    U:PRINT" {DOWN}";:NEXT:RETURN                  :rem 81
110 GOSUB 1000:PRINT"{3}{HOME}{17 DOWN}{4 RIGHT}
    ";:FORX=OTOU:PRINT"N{UP}";:NEXT:RETURN:rem 140
115 PRINT"{3}{HOME}{17 DOWN}{4 RIGHT}";:FORX=OTO
    U:PRINT" {UP}";:NEXT:RETURN                    :rem 72
120 GOSUB 1000:PRINT"{RED}{HOME}{4 DOWN}"SPC(Z1);:
    FORX=OTOU:PRINT"M{DOWN}";:NEXT:RETURN          :rem 3
125 PRINT"{RED}{HOME}{4 DOWN}"SPC(Z1);:FORX=OTOU:P
    RINT" {DOWN}";:NEXT:RETURN                     :rem 192
130 GOSUB1000:PRINT"{BLU}{HOME}{23 DOWN}"SPC(22);:
    FORX=OTO13:PRINT"N{UP}";                       :rem 32
135 NEXT:RETURN                                    :rem 242
140 PRINT"{BLU}{HOME}{23 DOWN}"SPC(22);:FORX=OTO13
    :PRINT" {UP}";:NEXT:RETURN                     :rem 107
```

```
145 GOSUB 1000: PRINT "{GRN}{HOME}{6 DOWN}"SPC(W);
    :FORX=OTOY: PRINT "M{DOWN}";: NEXT: RETURN
                                              :rem 254
150 PRINT "{GRN}{HOME}{6 DOWN}"SPC(W);:FORX=OTOY:
    {SPACE}PRINT " {DOWN}";: NEXT: RETURN :rem 178
155 GOSUB 1000:PRINT"{GRN}{HOME}{22 DOWN}"SPC(W);:
    FORX=OTOY:PRINT"N{UP}";                   :rem 253
160 NEXT:RETURN                               :rem 240
165 PRINT"{HOME}{22 DOWN}"SPC(W);:FORX=OTOY:PRINT"
     {UP}";:NEXT:RETURN                        :rem 42
170 GOSUB 1000: PRINT"{CYN}{HOME}{5 DOWN}"SPC(P);:
    FORX=OTOY:PRINT"M{DOWN}";:NEXT:RETURN :rem 101
175 PRINT"{CYN}{HOME}{5 DOWN}"SPC(P);:FORX=OTOY:PR
    INT" {DOWN}";:NEXT:RETURN                 :rem 34
180 GOSUB 1000:PRINT"{YEL}{HOME}{23 DOWN}"SPC(P);:
    FORX=OTOY:PRINT"N{UP}";                   :rem 133
185 NEXT:RETURN                               :rem 247
190 PRINT"{HOME}{23 DOWN}"SPC(P);:FORX=OTOY:PRINT"
     {UP}";: NEXT: RETURN                      :rem 50
195 GOSUB 1000:PRINT"{HOME}{14 DOWN}"SPC(Z);"{YEL}
    **":RETURN                                 :rem 8
200 PRINT"{HOME}{YEL}{RVS} **** MAGNETO **** {OFF}
    {8 SPACES}{WHT}{RVS}LEVEL{OFF}"DL         :rem 196
205 PRINT"{DOWN}{RVS}{WHT}SCORE{OFF}"SC;"
    {2 SPACES}{RVS}SHIPS LEFT{OFF}"SL;"{2 SPACES}
    {RVS}HIGH{OFF}"HS: RETURN                 :rem 160
210 FOR X2 = O TO GD                          :rem 190
215 SYS(B):ZT = PEEK(M)                        :rem 18
220 R=INT(RND(K)*D)+K                         :rem 195
230 ON R GOSUB 50,60,75,85,195                :rem 247
240 L= PEEK(C)+(PEEK(E)*F4)                   :rem 179
250 IF PEEK (L) <> G THEN GOSUB 3000           :rem 27
260 IF PEEK (H) = J THEN GOSUB 4900           :rem 232
265 NEXT: GOSUB 55: GOSUB 70: GOSUB 80: GOSUB 90
                                              :rem 102
310 FOR X2 = O TO GD                          :rem 191
315 SYS(B):ZT = PEEK(M)                        :rem 19
320 R=INT(RND(K)*D)+K                         :rem 196
330 ON R GOSUB 95,110,120,130,195             :rem 123
340 L= PEEK(C)+(PEEK(E)*F4)                   :rem 180
350 IF PEEK (L) <> G THEN GOSUB 3000           :rem 28
360 IF PEEK (H) = J THEN GOSUB 4900           :rem 233
365 NEXT: GOSUB 110: GOSUB 115: GOSUB 125: GOSUB 1
    40                                         :rem 27
410 FOR X2 = O TO GD                          :rem 192
415 SYS(B):ZT = PEEK(M)                        :rem 20
420 R=INT(RND(K)*D)+K                         :rem 197
430 ON R GOSUB 145,155,170,180,195            :rem 187
440 L= PEEK(C)+(PEEK(E)*F4)                   :rem 181
```

```
450 IF PEEK (L) <> G THEN GOSUB 3000        :rem 29
460 IF PEEK (H) = J THEN GOSUB 4900        :rem 234
465 NEXT: GOSUB 150: GOSUB 165: GOSUB 175: GOSUB 1
    90                                      :rem 47
999 GOTO 210                               :rem 119
1000 POKE S2 , R + F: RETURN               :rem 126
1010 REM SET UP FOR GAME SOUND             :rem 193
1015 GOSUB 45                              :rem 176
1020 POKE S + 5, 15: POKE S + 6, 255        :rem 28
1025 POKE S, K: POKE S + 24, 6             :rem 112
1030 POKES + 4, 21: POKE S + 15, 200        :rem 63
1035 RETURN                                :rem 169
3000 IF PEEK(L) <> C2 THEN 3010            :rem 246
3004 IF CH <> 0 THEN RETURN                :rem 185
3006 GOTO 3100                             :rem 198
3010 REM SOUND FOR CHARACTER COLLISION TO BEAMS
                                           :rem 92
3030 GOSUB 45: RM = 31                      :rem 41
3040 POKE S + K, 15 : POKE S + 5 , 16 : POKE S + 6
     , 240                                 :rem 223
3050 FOR X = 0 TO 60                       :rem 156
3060 POKE S + 15 , RM : POKE S + 24 , 31   :rem 132
3065 POKE 53281, X/4                       :rem 231
3070 POKE S + 4 , 21                        :rem 58
3080 RM = RM -.5                           :rem 214
3090 NEXT: GOSUB 4000 : POKE 53281 , 0      :rem 81
3092 SL = SL - K: IF SL = 0 THEN 3500      :rem 172
3095 GOSUB 1010: CH = 0: GOTO 200           :rem 74
3100 REM JEWEL CAPTURE SOUNDS               :rem 15

3104 CH = CH + 1: POKE 833, 1:SC = SC + 100:rem 58
3105 GOSUB 45: RM = 6 : C3 = .75           :rem 133
3110 POKE S + K ,3: POKE S + 5 ,16: POKE S + 6 , E
                                           :rem 89
3115 POKE S + 15 , RM: POKE S + 23 , K: POKE S + 2
     4 , 31: POKE S + Z1, 90               :rem 79
3120 POKE S + 4 , 21                       :rem 54
3130 RM = RM + C3: IF RM > 100 THEN 3140    :rem 17
3135 POKE S + 15 , RM: GOTO 3130           :rem 228
3140 GOSUB 1010: GOTO 200                   :rem 15
3500 REM END GAME SEQUENCE                 :rem 246
3510 PRINT "{CLR}": POKE V + 21, 0: POKE S + 24, 0
                                           :rem 127
3520 PRINT"{2 DOWN}{YEL}SORRY CAPTAIN, YOU HAVE RU
     N OUT OF SHIPS"                       :rem 80
3530 PRINT"{4 DOWN}{CYN}HOWEVER, YOU WERE DOING VE
     RY WELL. YOU"                         :rem 17
3540 PRINT"{DOWN} MANAGED TO GET"SC;"POINTS AND MA
     KE IT"                                :rem 158
```

```
3550 PRINTSPC(13) "{DOWN} TO LEVEL"DL :PRINT"
     {2 DOWN}{WHT}{4 SPACES}HIT THE {RVS}SPACE BAR
     {OFF} TO";                                  :rem 176
3560 PRINT" TRY AGAIN":PRINT"{DOWN}{7 SPACES}OR AN
     Y OTHER KEY TO END"                         :rem 136
3565 IF SC > HS THEN HS = SC                     :rem 110
3570 GET A$: IF A$ THEN 3570                      :rem 64
3580 GET A$: IF A$ = "" THEN 3580                :rem 195
3590 IF A$ = " " THEN SC = 0: CH = 0: SP = 0: DL =
      1: GD = 25: SL = 5: GOTO 20                :rem 243
3600 PRINT"{CLR}{3 DOWN}{12 SPACES}{WHT}{RVS}
     {2 SPACES}GOOD BYE{2 SPACES}{OFF}"          :rem 18
3610 FOR T = 0 TO 3000: NEXT: END                :rem 98
4000 REM VARIABLE SET UP - FOR CKX AND OTHERS
                                                 :rem 33
4010 POKE 828,46: POKE 829 ,60: POKE 830 ,45: POKE
      831 ,53: POKE 832 , 86                     :rem 156
4020 POKE 833, 15: POKE 251, 49: POKE 252 , 6: POK
     E 253, 49: POKE 254 , 218                   :rem 138
4030 P=2040: RG = 192: RO= 193 : S= 54272        :rem 73
4040 B=834: C=251: E=252: F4=256: G = 86: H = 197:
      J = 10: K = 1: M = 53279                   :rem 138
4050 P = 8: Q= 12: U = 13: W = 14: Y = 17: Z = 19:
      Z1 = 22: C2 = 42                           :rem 71
4060 Z2 = 27: S2 = 54273: F = 220                :rem 78
4100 RETURN                                      :rem 165
4700 REM SET UP ALL COLORS FOR MULTI-COLOR MODE
                                                 :rem 46
4710 POKE 53281,0: REM SCREEN BLACK              :rem 149
4720 POKE 53282,14: REM BACKGROUND #1 TO LIGHT BLU
     E                                           :rem 38
4730 POKE 53283,4: REM BACKGROUND #2 TO PURPLE
                                                 :rem 48
4740 POKE 53270, PEEK (53270) OR 16: REM TURN ON M
     ULTICOLOR MODE                              :rem 221
4750 RETURN                                      :rem 176
4900 REM DISSOLVE SPRITE ROUTINE                  :rem 23
4902 IF PEEK (V+31) = 0 THEN RETURN               :rem 53
4904 IF CH = O THEN RETURN                       :rem 134
4905 RM = 4: GOSUB 45: SP = SP + PEEK (V+31): SC =
      SC + (100 * SL)                            :rem 193
4910 POKE S + K ,RM: POKE S + 5 ,16: POKE S + 6 ,
     {SPACE}E                                    :rem 206
4915 POKE V + 29, SP: POKE V + 23, SP            :rem 219
4920 POKE S + 15 , 5: POKE S + 24 , 31            :rem 32
4925 POKE S + 4 , Z                               :rem 59
4930 FOR T = 4 TO 80 STEP .25: POKE S + K , T: NEX
     T                                           :rem 131
4935 FOR X = O TO 7: POKE P + X, RO : NEXT: POKE V
      + 28, SP                                   :rem 31
```

```
4940 FOR T = 81{2 SPACES}TO 150 STEP .25: POKE S +
     K , T : NEXT                              :rem 231
4945 GOSUB 1010                                :rem 24
4950 FOR X = 0 TO 7: POKE P + X, RG : NEXT:rem 241
4955 POKE V + 21, (255-SP): POKE V + 29, 0: POKE V
     + 23, 0                                   :rem 109
4958 IF PEEK (V+21) = 0 THEN 7000              :rem 38
4960 CH = 0: POKE 833, 15: GOTO 200            :rem 136
5000 REM BASIC LOADER FOR FAST(CKX) KEYBOARD CONTR
     OLLED CHARACTER ROUTINE                    :rem 8
5010 FOR X = 834 TO 1014: READ A: POKE X , A: NEXT
                                                :rem 136
5020 DATA 32,176,3,165,197,205,60,3,208,14,32,183,
     3,32,228,3,176,63,32                      :rem 111
5030 DATA 160,3,76,147,3,205,61,3,208,14,32,160,3,
     32,237,3,144,44,32                        :rem 249
5040 DATA 183,3,76,147,3,205,62,3,208,14,32,199,3,
     32,228,3,176,25,32,217                    :rem 214
5050 DATA 3,76,147,3,205,63,3,208,14,32,217,3,32,2
     37,3,144,6,32,199,3                       :rem 57
5060 DATA 76,147,3,160,0,173,64,3,145,251,173,65,3
     ,145,253,96,24,165                        :rem 28
5070 DATA 251,105,40,133,251,133,253,144,4     :rem 5
5080 DATA 230,252,230,254,96,160,0,169,32,145,251,
     96,56,165,251,233,40                      :rem 124
5090 DATA 133,251,133,253,176,4,198,252,198,254,96
     ,165,251,208,9,198,251                    :rem 2
5100 DATA 198,253,198,252,198,254,96,198,251,198,2
     53,96,230,251,230,253,208                 :rem 158
5110 DATA 4,230,252,230,254,96,165,251,233,160,165
     ,252,233,4,96,165,251                     :rem 169
5120 DATA 201,232,165,252,233,7,96,234         :rem 76
5130 RETURN                                    :rem 169
5500 REM SCREEN AND VARIABLE SET UP ROUTINE:rem 62
5505 V = 53248:POKE 53280, 0: POKE 53281 , 0: PRIN
     T"{HOME}{WHT}"                            :rem 26
5510 POKEV,28:POKEV+1,120:POKEV+2,110:POKEV+3,80:P
     OKEV+16,136:POKEV+4,235                   :rem 70
5515 POKEV+5,80:POKEV+6,60:POKEV+7,120:POKEV+8,28:
     POKEV+9,186:POKEV+10,110                  :rem 138
5520 POKEV+11,226:POKEV+12,235:POKEV+13,226:POKEV+
     14,60:POKEV+15,186                        :rem 199
5540 POKE V + 21, 255: POKE V + 23,0:POKE V + 28,
     {SPACE}0: POKE V + 29 , 0                 :rem 181
5550 FOR X = 0 TO 7: POKE 2040 + X , 192: POKE V +
      39 + X , 3:NEXT                          :rem 128
5570 PRINT"{2 DOWN}EEEEEEEEEEEEEEEEEEEEEEEEEEEEEEE
     EEEEEEEEE"                                :rem 140
5610 RETURN                                    :rem 172
```

```
6000 REM READ SPRITE DATA                    :rem 183
6005 FOR X = 0 TO 63: READ A:POKE X + 12288, A: NE
     XT                                      :rem 241
6010 FOR X = 0 TO 63: READ A:POKE X + 12352, A: NE
     XT                                      :rem 229
6020 REM POD 1 DATA                          :rem 218
6030 DATA 0,16,0,0,56,0,0,124,0,1,255,0,2,0,128,2,
     0,128,15,255,224,20                     :rem 251
6040 DATA 130,80,25,1,48,18,56,144,242       :rem 73
6050 DATA 108,158,242,108,158,18,56,144,25,1,48,20
     ,130,80,15,255,224,2,0                  :rem 206
6060 DATA 128,1,255,0,0,198,0,1,131,0,3,1,128,0
                                             :rem 225
6070 REM POD 2 DATA                          :rem 224
6080 DATA 0,16,0,0,56,0,0,124,0,1,255,0,2,0,128,2,
     0,128,15,255,224,17                      :rem 6
6090 DATA 1,16,18,56,144,20,68,80,244,146,94,244,1
     46,94,20,68,80,18,56                    :rem 142
6100 DATA 144,17,1,16,15,255,224,2,0,128,1,255,0,0
     ,198,0,1,131,0,3,1,128,0                :rem 253
6110 RETURN                                  :rem 168
7000 REM GOT THEM ALL ROUTINE                :rem 194
7005 SC = SC + 500                           :rem 245
7010 PRINT"{CLR}": POKE V + 21, 0: POKE S + 24 , 0
                                             :rem 126
7020 PRINT"{2 DOWN}{CYN} CONGRATULATIONS!!
     {2 SPACES}YOU HAVE ELIMINATED           :rem 245
7030 PRINT"{DOWN}{5 SPACES}ALL OF THE DEADLY {WHT}
     {RVS} MAGNETOS {OFF}{CYN}.               :rem 96
7040 PRINT"{PUR}{2 DOWN}{12 SPACES}**************
     {CYN}"                                  :rem 111
7050 PRINT"{2 DOWN}{4 SPACES}YOUR SCORE IS"SC;". W
     ELL DONE !!"                             :rem 5
7060 PRINT"{DOWN}{2 SPACES}NOW YOU MUST ATTEMPT TH
     E NEXT LEVEL"                           :rem 160
7070 PRINT"{2 DOWN}{YEL}{7 SPACES}HIT ANY KEY TO C
     ONTINUE{3 SPACES}"                       :rem 29
7085 IF SC > HS THEN HS = SC                 :rem 111
7090 GET A$: IF A$ THEN 7090                  :rem 66
7100 GET A$: IF A$ = "" THEN 7100            :rem 179
7110 CH = 0: SP = 0                           :rem 11
7140 GD = GD - 5: IF GD < 5 THEN GD = 5      :rem 114
7145 DL = DL + 1                             :rem 138
7150 GOTO 20                                 :rem 104
9999 REM INTRO AND INSTRUCTION ROUTINES       :rem 2
10000 POKE 53280, 3:POKE 53281 , 1            :rem 74
10005 PRINT"{CLR}{2 DOWN}{7}{RVS}{40 SPACES}";
                                             :rem 99
10010 PRINT"{RVS}{6 SPACES}{N}{M}{N}{H}{N}{M
      {2 SPACES}{O}{P} {N}{M}{2 SPACES}{H}{O}{Y}
      {2 Y}{O}{Y} {O}{2 Y}{P}{6 SPACES}";    :rem 209
```

```
10015 PRINT"{2 SPACES}*** {N}{2 SPACES}{H}
      {N} M {H} {P}{N} M {H}O{4 SPACES}
      {H}{2 SPACES}{H}{2 SPACES}{N}
      {2 SPACES}*** ";                          :rem 67
10020 PRINT"{6 SPACES}{N}{2 SPACES}{H}{N}
      {Y}P L{P}@{N}{2 SPACES}M{H}L{2 P}
      {2 SPACES}{H}{2 SPACES}L{2 P}@{6 SPACES}
      ";                                        :rem 176
10025 PRINT"{40 SPACES}";                        :rem 4
10030 PRINT"{40 SPACES}";                        :rem 0
10035 PRINT"{40 SPACES}";                        :rem 5
10040 PRINT"{40 SPACES}{OFF}"                    :rem 88
10045 PRINT"{6 DOWN}{BLU}{5 SPACES}INSTRUCTIONS ??
      {2 SPACES}( {RVS}Y{OFF} OR {RVS}N{OFF} )
                                                 :rem 66
10050 S = 54272: D = 4                           :rem 121
10055 POKE S + 1 ,10: POKE S + 5 ,16: POKE S + 6 ,
       252                                       :rem 247
10060 POKE S + 15 , D : POKE S + 23 , 1          :rem 35
10065 POKE S + 4 , 19: POKE S + 24 , 31          :rem 80
10070 FOR X=0 TO 30 STEP.5:POKE S+1,X*2:POKE S+22,
      FF:FF=FF+3:NEXT                            :rem 170
10075 FOR X=30 TO 0 STEP-.5:POKE S+1,X*2:POKE S+22
      ,FF:FF=FF-3:NEXT                           :rem 222
10080 D = D + 2                                  :rem 27
10085 IF D > 20 THEN POKE S + 24 , 0: GOTO 10105
                                                 :rem 121
10090 GET A$: IF A$ = "N" THEN POKE S + 24, 0: RET
      URN                                        :rem 244
10095 IF A$ = "Y" THEN POKE S + 24, 0 : GOTO 10125
                                                 :rem 215
10100 GOTO 10060                                 :rem 34
10105 GET A$: IF A$ = "" THEN 10105              :rem 17
10110 IF A$ = "N" THEN RETURN                    :rem 197
10115 IF A$ = "Y" THEN POKE S + 24, 0 : GOTO 10125
                                                 :rem 208
10120 GOTO 10105                                 :rem 36
10125 PRINT"{CLR}{12 SPACES}*** MAGNETO ***"
                                                 :rem 100
10130 PRINT"{2 DOWN}{2 SPACES}THE COSMIC BALANCE O
      F THE GALAXY HAS"                          :rem 133
10135 PRINT"{DOWN}BEEN UPSET BY A GROUP OF 8 DEADL
      Y ATOMIC"                                  :rem 45
10140 PRINT"GENERATORS WHOSE A.I. CIRCUITS HAVE BE
      EN"                                        :rem 209
10145 PRINT"TAMPERED WITH. YOU MUST DESTROY ALL 8
      {SPACE}TO"                                 :rem 172
10150 PRINT"SAVE OUR CIVILIZATION. THEY HAVE DEADL
      Y"                                         :rem 193
```

```
10155 PRINT"{DOWN}BEAMS TO PROTECT THEIR POWER CRY
      STALS."                                  :rem 182
10160 PRINT"{DOWN}TO DESTROY THEM YOU MUST REACH T
      HE POWER"                                :rem 236
10165 PRINT"CRYSTALS AND GET ENERGIZED BY THE ATOM
      IC"                                      :rem 204
10170 PRINT"FORCE. THE BEAMS WILL DESTROY YOU IF Y
      OU"                                      :rem 155
10175 PRINT"HIT THEM WHEN THE FORCE IS IN USE.
      {DOWN}"                                  :rem 195
10180 PRINT"{6 SPACES}{RVS}HIT ANY KEY TO CONTINUE
      {OFF}"                                    :rem 45
10185 GET A$: IF A$ THEN 10185                 :rem 160
10190 GET A$: IF A$ = "" THEN 10190             :rem 25
10195 PRINT"{CLR}{12 SPACES}*** MAGNETO ***"
                                               :rem 107
10200 PRINT"{2 DOWN} HOWEVER, THE MAGNETOS HAVE ON
      LY ENOUGH"                               :rem 157
10205 PRINT"{DOWN}{2 SPACES}POWER TO ENERGIZE ONE
      {SPACE}OF THEIR BEAMS"                   :rem 190
10210 PRINT"{DOWN}AT A TIME SO THE FORCE IS FOUND
      {SPACE}RANDOMLY"                          :rem 75
10215 PRINT" IN THE BEAMS AND POWER CRYSTALS. WITH
       "                                       :rem 233
10220 PRINT"{DOWN}CARE YOU CAN DESTROY THEM ALL. G
      OOD LUCK"                                :rem 120
10225 PRINT"{5 DOWN}{7 SPACES}{RVS}HIT ANY KEY TO
      {SPACE}CONTINUE{OFF}"                    :rem 130
10230 GET A$: IF A$ = "" THEN 10230             :rem 15
10235 PRINT"{CLR}{12 SPACES}*** MAGNETO ***"
                                               :rem 102
10240 PRINT"{2 DOWN}{10 SPACES}{RVS} KEYBOARD CONT
      ROLS {OFF}"                               :rem 83
10245 PRINT"{3 DOWN}{5 SPACES}{RVS}@{OFF} MOVES SH
      IP UP":PRINT"{DOWN}{5 SPACES}{RVS}={OFF} MOV
      ES SHIP RIGHT"                           :rem 128
10250 PRINT"{DOWN}{5 SPACES}{RVS}:{OFF} MOVES SHIP
       LEFT"                                   :rem 161
10253 PRINT"{DOWN}{5 SPACES}{RVS} SPACE BAR {OFF}
      {SPACE}MOVES SHIP DOWN"                  :rem 184
10255 PRINT"{DOWN}{4 SPACES}{RVS} A {OFF} DESTROYS
       MAGNETOS WHEN YOU ARE "                 :rem 166
10260 PRINT"{DOWN}{5 SPACES}ENERGIZED AND POSITION
      ED DIRECTLY "                            :rem 169
10265 PRINT"{DOWN}{5 SPACES}UNDERNEATH{2 SPACES}TH
      EM."                                      :rem 42
10270 PRINT"{PUR}{2 DOWN}{5 SPACES}{RVS}HIT ANY KE
      Y TO BEGIN THE GAME{OFF}"                :rem 230
10275 GET A$: IF A$ = "" THEN 10275             :rem 33
10280 RETURN                                   :rem 219
```

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 this book are written in a computer language called BASIC. BASIC is easy to learn and is built into all Commodore 64s.

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 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 your machine'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 video, lowercase, and control characters? It's all explained in your computer's manuals.

A Quick Review

1) Type in the program a line at a time, in order. Press RETURN at the end of each line. Use backspace or the back arrow to correct 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.

How to Type In Programs

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, Commodore 64 program listings will contain words within 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. These characters can be entered by holding down the CTRL key while typing the letter in the braces. For example, {A} would indicate that you should press CTRL-A.

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 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.

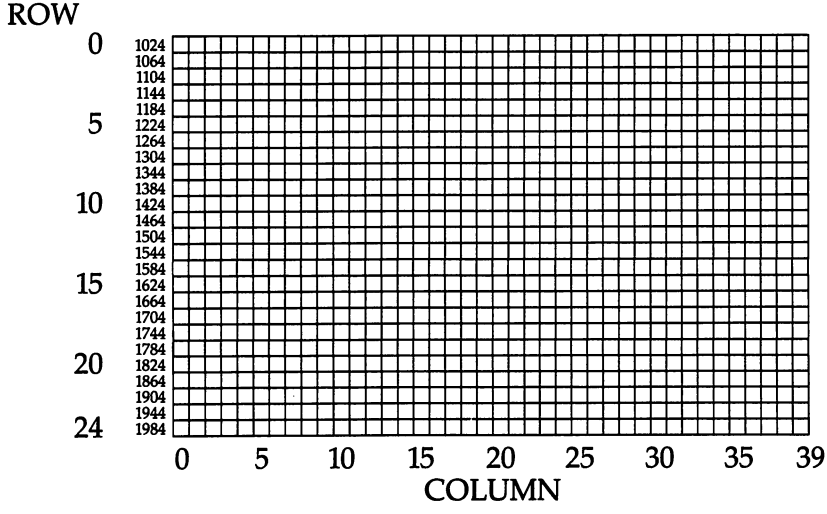
You also go into quote mode when you INSerT spaces into

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.

When You Read:	Press:	See:
{CLR}	SHIFT + CLR/HOME	
{HOME}	CLR/HOME	
{UP}	SHIFT + CRSR ↑	
{DOWN}	CRSR ↑	
{LEFT}	SHIFT + CRSR →	
{RIGHT}	CRSR →	
{RVS}	CTRL + 9	
{OFF}	CTRL + 0	
{BLK}	CTRL + 1	
{WHT}	CTRL + 2	
{RED}	CTRL + 3	
{CYN}	CTRL + 4	
{PUR}	CTRL + 5	
{GRN}	CTRL + 6	
{BLU}	CTRL + 7	
{YEL}	CTRL + 8	

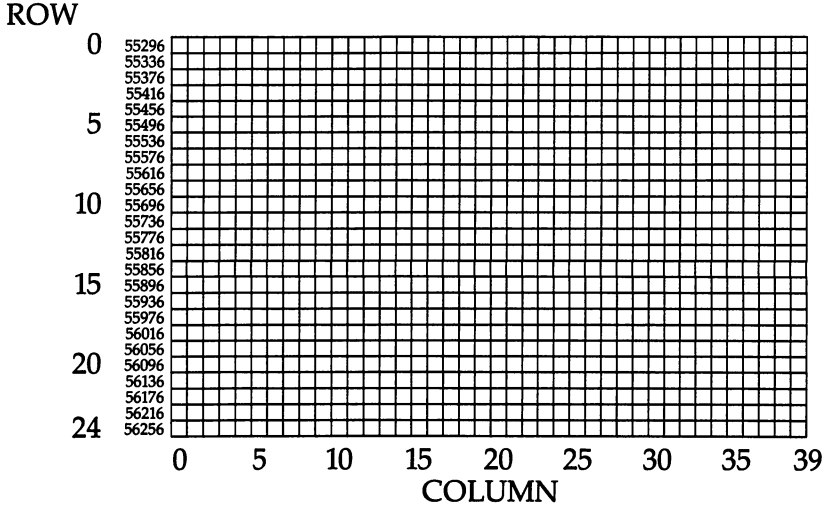
When You Read:	Press:	See:
{1}	COMMODORE + 1	
{2}	COMMODORE + 2	
{3}	COMMODORE + 3	
{4}	COMMODORE + 4	
{5}	COMMODORE + 5	
{6}	COMMODORE + 6	
{7}	COMMODORE + 7	
{8}	COMMODORE + 8	
{F1}	f1	
{F2}	f2	
{F3}	f3	
{F4}	f4	
{F5}	f5	
{F6}	f6	
{F7}	f7	
{F8}	f8	
£	£	

Screen Location Table



Appendix D

Screen Color Memory Table



Color Values Table

Value To POKE For Each Color

Color	Low nybble color value	High nybble color value	Select multicolor color value
Black	0	0	8
White	1	16	9
Red	2	32	10
Cyan	3	48	11
Purple	4	64	12
Green	5	80	13
Blue	6	96	14
Yellow	7	112	15
Orange	8	128	—
Brown	9	144	—
Light Red	10	160	—
Dark Gray	11	176	—
Medium Gray	12	192	—
Light Green	13	208	—
Light Blue	14	224	—
Light Gray	15	240	—

Where to POKE Color Values For Each Mode

Mode*	Bit or bit-pair	Location	Color value
Regular text	0	53281	Low nybble
	1	Color memory	Low nybble
Multicolor text	00	53281	Low nybble
	01	53282	Low nybble
	10	53283	Low nybble
	11	Color memory	Select multicolor
Extended color text#	00	53281	Low nybble
	01	53282	Low nybble
	10	53283	Low nybble
	11	53284	Low nybble
Bitmapped	0	Screen memory	Low nybble±
	1	Screen memory	High nybble±
Multicolor bitmapped	00	53281	Low nybble
	01	Screen memory	High nybble±
	10	Screen memory	Low nybble±
	11	Color memory	Low nybble±

* For all modes, the screen border color is controlled by POKEing location 53280 with the low nybble color value.

\# In extended color mode, bits 6 and 7 of each byte of screen memory serve as the bit-pair controlling background color. Because only bits 0-5 are available for character selection, only characters with screen codes 0-63 can be used in this mode.

± In the bitmapped modes, the high and low nybble color values are ORed together and POKEd into the *same location* in screen memory to control the colors of the corresponding *cell* in the bit map. For example, to control the colors of cell 0 of the bit map, OR the high and low nybble values and POKE the result into location 0 of screen memory.

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

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	[	129	ORANGE
92	£	133	f1
93	]	134	f3
94	↑	135	f5
95	←	136	f7
96		137	f2
97		138	f4
98		139	f6
99		140	f8
100		141	SHIFTED RETURN
101		142	UPPERCASE
102		144	BLACK
103		145	CURSOR UP
104		146	REVERSE VIDEO OFF
105		147	CLEAR SCREEN
106		148	INSERT
107		149	BROWN
108		150	LIGHT RED
109		151	GRAY 1
110		152	GRAY 2
111		153	LIGHT GREEN
112		154	LIGHT BLUE
113		155	GRAY 3
114		156	PURPLE
115		157	CURSOR LEFT
116		158	YELLOW
117		159	CYAN
118		160	SPACE
119		161	

ASCII	CHARACTER	ASCII	CHARACTER
162		200	
163		201	
164		202	
165		203	
166		204	
167		205	
168		206	
169		207	
170		208	
171		209	
172		210	
173		211	
174		212	
175		213	
176		214	
177		215	
178		216	
179		217	
180		218	
181		219	
182		220	
183		221	
184		222	
185		223	
186		224	SPACE
187		225	
188		226	
189		227	
190		228	
191		229	
192		230	
193		231	
194		232	
195		233	
196		234	
197		235	
198		236	
199		237	

ASCII	CHARACTER
238	
239	
240	
241	
242	
243	
244	
245	
246	
247	
248	
249	
250	
251	
252	
253	
254	
255	

0-4, 6, 7, 10-12, 15, 16, 21-27, 128, 130-132, and 143 are not used.

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	=	=

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
62	>	>
63	?	?
64		
65		A
66		B
67		C
68		D
69		E
70		F
71		G
72		H
73		I
74		J
75		K
76		L
77		M
78		N
79		O
80		P
81		Q
82		R
83		S
84		T
85		U
86		V
87		W
88		X
89		Y
90		Z
91		
92		
93		
94		
95		
96	-space-	
97		
98		

POKE	Uppercase and Full Graphics Set	Lower- and Uppercase
99		
100		
101		
102		
103		
104		
105		
106		
107		
108		
109		
110		
111		
112		
113		
114		
115		
116		
117		
118		
119		
120		
121		
122		
123		
124		
125		
126		
127		

128 – 255 reverse video of 0-127

Commodore 64 Key Codes

Key	Keycode	Key	Keycode
A	10	6	19
B	28	7	24
C	20	8	27
D	18	9	32
E	14	0	35
F	21	+	40
G	26	-	43
H	29	£	48
I	33	CLR/HOME	51
J	34	INST/DEL	0
K	37	←	57
L	42	@	46
M	36	*	49
N	39	↑	54
O	38	:	45
P	41	;	50
Q	62	=	53
R	17	RETURN	1
S	13	,	47
T	22	.	44
U	30	/	55
V	31	CRSR↑↓	7
W	9	CRSR⇄	2
X	23	f1	4
Y	25	f3	5
Z	12	f5	6
1	56	f7	3
2	59	SPACE	60
3	8	RUN/STOP	63
4	11	NO KEY	
5	16	PRESSED	64

The keycode is the number found at location 197 for the current key being pressed. Try this one-line program:

```
10 PRINT PEEK (197): GOTO 10
```

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

Reference Books of Interest

Commodore 64 Programmer's Reference Guide. Indianapolis: Commodore Business Machines, Inc., and Howard W. Sams and Co., Inc., 1982

COMPUTE!'s First Book of Commodore 64 Sound and Graphics. Greensboro, N.C.: COMPUTE! Publications, Inc., 1983

COMPUTE!'s First Book of Commodore 64 Games. Greensboro, N.C.: COMPUTE! Publications, Inc., 1983

COMPUTE!'s First Book of Commodore 64. Greensboro, N.C.: COMPUTE! Publications, Inc., 1983

Heilborn, John. *COMPUTE!'s Reference Guide to Commodore 64 Graphics.* Greensboro, N.C.: COMPUTE! Publications, Inc., 1983

Leemon, Sheldon. *Mapping the Commodore 64.* Greensboro, N.C.: COMPUTE! Publications, Inc., 1984

Mansfield, Richard. *Machine Language for Beginners.* Greensboro, N.C.: COMPUTE! Publications, Inc., 1983

The Automatic Proofreader

"The Automatic Proofreader" will help you type in program listings in this book without typing mistakes. It is a short error-checking program that hides itself in memory. When activated, it lets you know immediately after typing a line from a program listing if you have made a mistake. Please read these instructions carefully before typing any programs.

Preparing the Proofreader

1. Using the listing below, type in the Proofreader. Watch out for typing an l instead of a 1, or an O instead of a 0, extra commas, etc.

2. SAVE it on tape or disk at least twice *before running it for the first time.* If you mistype the Proofreader, it may cause a system crash when you first run it. By SAVEing a copy beforehand, you can reLOAD it and hunt for your error. Also, you'll want a backup copy of the Proofreader. Future COMPUTE! Books as well as *COMPUTE!'s Gazette* will use the Proofreader.

3. RUN the Proofreader. It will be POKEd into a relatively safe area of memory, the cassette buffer.

4. Type RUN to activate the Proofreader. If you ever need to reactivate it, just enter the command SYS 886 and press RETURN.

Using The Proofreader

Most of the listings in this book have a *checksum number* appended to the end of each line, for example ":rem123". The only exceptions are some of the one-line listings. *Don't enter this statement when typing in a program.* It is just for your information. The rem makes the number harmless if someone does type it in. It will, however, use up memory if you enter it, and it will confuse the Proofreader, even if you entered the rest of the line correctly.

When you type a line from a program listing and press RETURN, the Proofreader displays a number at the top of your screen. *This checksum number must match the checksum number in the printed listing.* If it doesn't, it means you typed the line differently than the way it is listed. Immediately recheck your typing. Remember, don't type the rem statement with the checksum number; it is pubished only so you can check it against the number which appears on your screen.

The Proofreader is not picky with spaces. It will not notice extra spaces or missing ones. This is for your convenience, since spacing is generally not important. But occasionally proper spacing *is* important, so be extra careful with spaces, since the Proofreader will catch practically everything else that can go wrong.

There's another thing to watch out for: if you enter the line by using abbreviations for commands, the checksum will not match up. But there is a way to make the Proofreader check it. After entering the line, LIST it. This eliminates the abbreviations. Then move the cursor up to the line and press RETURN. It should now match the checksum. You can check whole groups of lines this way.

When you're done with the Proofreader, disable it by pressing RUN/STOP-RESTORE (hold down the RUN/STOP key and press RESTORE). If you need it again, enter SYS 886. It will then be ready once again to act as your personal typing aid. However, sometimes the Proofreader can be wiped out of memory. In this case, you'll have to reLOAD the Proofreader from tape or disk.

Automatic Proofreader

```
100 PRINT"{CLR}PLEASE WAIT...":FORI=886TO1018:READ
    A:CK=CK+A:POKEI,A:NEXT
110 IF CK<>17539 THEN PRINT"{DOWN}YOU MADE AN ERRO
    R":PRINT"IN DATA STATEMENTS.":END
120 SYS886:PRINT"{CLR}{2 DOWN}PROOFREADER ACTIVATE
    D.":NEW
886 DATA 173,036,003,201,150,208
892 DATA 001,096,141,151,003,173
898 DATA 037,003,141,152,003,169
904 DATA 150,141,036,003,169,003
910 DATA 141,037,003,169,000,133
916 DATA 254,096,032,087,241,133
922 DATA 251,134,252,132,253,008
928 DATA 201,013,240,017,201,032
934 DATA 240,005,024,101,254,133
940 DATA 254,165,251,166,252,164
946 DATA 253,040,096,169,013,032
952 DATA 210,255,165,214,141,251
958 DATA 003,206,251,003,169,000
964 DATA 133,216,169,019,032,210
970 DATA 255,169,018,032,210,255
976 DATA 169,058,032,210,255,166
982 DATA 254,169,000,133,254,172
```

```
988 DATA 151,003,192,087,208,006
994 DATA 032,205,189,076,235,003
1000 DATA 032,205,221,169,032,032
1006 DATA 210,255,032,210,255,173
1012 DATA 251,003,133,214,076,173
1018 DATA 003
```

Index

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.

36-1

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 challenging programming tasks. Whether you are an advanced programmer, or just starting out, you'll find many valuable, useful techniques and suggestions here. This book — a guide for anyone who has thought of designing and writing a game program on the Commodore 64 — shows you how.

Here are just a few of the topics covered:

- Custom character sets
- Sprite creation and movement
- Animated graphics
- Title screens, instructions, and farewells
- Sound
- Joystick and keyboard controls
- Machine language movement routines
- How to develop an idea
- Tips on getting real speed in BASIC programs

ISBN 0-942386-36-1