CHAPTER 6

SPRITE GRAPHICS

- Introduction to Sprites
- Sprite Creation
- Additional Notes on Sprite
- Binary Arithmetic

INTRODUCTION TO SPRITES

In previous chapters dealing with graphics, we saw that graphic symbols could be used in PRINT statements to create animation and add chartlike appearances to our displays.

A way was also shown to POKE character codes in specific screen memory locations. This would then place the appropriate characters directly on the screen in the right spot.

Creating animation in both these cases requires a lot of work because objects must be created from existing graphic symbols. Moving the object requires a number of program statements to keep track of the object and move it to a new spot. And, because of the limitation of using graphic symbols, the shape and resolution of the object might not be as good as required.

Using sprites in animated sequences eliminates a lot of these problems. A sprite is a high-resolution programmable object that can be made into just about any shape—through BASIC commands. The object can be easily moved around the screen by simply telling the computer the position the sprite should be moved to. The computer takes care of the rest.

And sprites have much more power than just that. Their color can be changed; you can tell if one object collides with another; they can be made to go in front and behind another; and they can be easily expanded in size, just for starters.

The penalty for all this is minimal. However, using sprites requires knowing some more details about how the Commodore 64 operates and how numbers are handled within the computer. It's not as difficult as it sounds, though. Just follow the examples and you'll be making your own sprites do amazing things in no time.

SPRITE CREATION

Sprites are controlled by a separate picture-maker in the Commodore 64. This picture maker handles the video display. It does all the hard work of creating and keeping track of characters and graphics, creating colors, and moving around.

This display circuit has 46 different "ON/OFF" locations which act like internal memory locations. Each of these locations breaks down into a series of 8 blocks. And each block can either be "on" or "off". We'll get into more detail about this later. By POKEing the appropriate decimal value in the proper memory location you can control the formation and movement of your sprite creations.

In addition to accessing many of the picture making locations we will also be using some of the Commodore 64's main memory to store information (data) that defines the sprites. Finally, eight memory locations directly after the screen memory will be used to tell the computer exactly which memory area each sprite will get its data from.

As we go through some examples, the process will be very straightforward, and you'll get the hang of it.

So let's get on with creating some sprite graphics. A sprite object is 24 dots wide by 21 dots long. Up to eight sprites can be controlled at a time. Sprites are displayed in a special independent 320 dot wide by 200 dot high area. However, you can use your sprite with any mode, high-resolution, low-resolution, text etc.

Say you want to create a balloon and have it float around the sky. The balloon could be designed as in the 24 by 21 grid on page 70.

The next step is to convert the graphic design into data the computer can use. Get a piece of notebook or graph paper and set up a sample grid that is 21 spaces down and 24 spaces across. Across the top write 128,64,32,16,8,4,2,1, three times (as shown) for each of the 24 squares. Number down the left side of the grid 1-21 for each row. Write the word DATA at the end of each row. Now fill in the grid with any design or use the balloon that we have. It's easiest to outline the shape first and then go back and fill in the grid.

Now if you think of all the squares you filled in as "on" then substitute a 1 for each filled square. For the one's that aren't filled in, they're "off" so put a zero.

Starting on the first row, you need to convert the dots into three separate pieces of data the computer can read. Each set of 8 squares is equal to one piece of data called a byte in our balloon. Working from the left, the first 8 squares are blank, or 0, so the value for that series of numbers is 0.

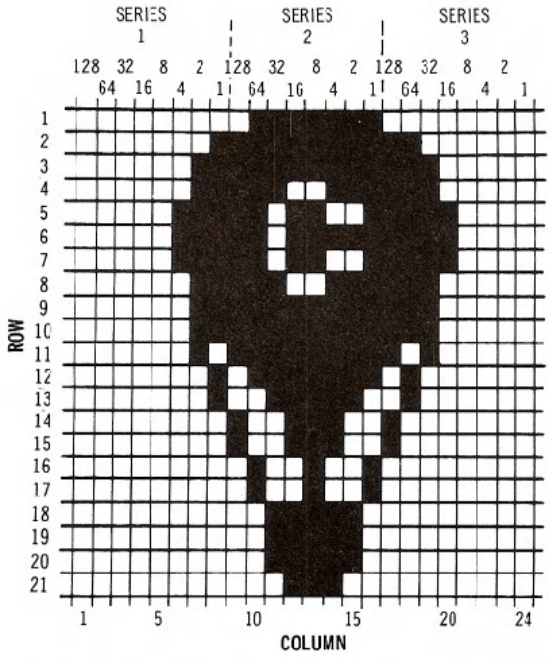
The middle series looks like this (again a 1 indicates a dot, 0 is a space):

128	64	32	16	8	4	2	1
0	1	1	1	1	1	1	1
↑	↑	↑	↑	↑	↑	↑	↑

0 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 127

The third series on the first row also contains blanks, so it, too, equals zero. Thus, the data for the first line is:

DATA 0, 127, 0



The series that make up row two are calculated like this:

Series 1:

0	0	0	0	0	0	0	1

1 = 1

Series 2:

1	1	1	1	1	1	1	1

128 + 64 + 32 + 16 + 8 + 4 + 2 + 1 = 255

Series 3:

1	1	0	0	0	0	0	0

128 + 64 = 192

For row 2, the data would be:

DATA 1,255,192

In the same way, the three series that make up each remaining row would be converted into their decimal value. Take the time to do the remainder of the conversion in this example.

Now that you have the data for your object, how can it be put to use? Type in the following program and see what happens.

```
1 REM UP, UP, AND AWAY!
5 PRINT "{CLR/HOME}"
10 V=53248  : REM START OF  DISPLAY CHIP
11 POKE V+21,4 : REM ENABLE SPRITE 2
12 POKE 2042,13 : REM SPRITE 2 DATA FROM 13TH BLK
20 FOR N = 0 TO 62: READ Q : POKE 832+N,Q: NEXT
30 FOR X = 0 TO 200          <-- GETS ITS INFO. FROM DATA*
40 POKE V+4,X: REM UPDATE X COORDINATES
50 POKE V+5,X: REM UPDATE Y COORDINATES
60 NEXT X
70 GOTO 30          <-- INFO. READ IN FROM Q*
200 DATA 0,127,0,1,255,192,3,255,224,3,231,224
210 DATA 7,217,240,7,223,240,7,217,240,3,231,224
220 DATA 3,255,224,3,255,224,2,255,160,1,127,64
230 DATA 1,62,64,0,156,128,0,156,128,0,73,0,0,73,0
240 DATA 0,62,0,0,62,0,0,62,0,0,28,0
```

*FOR MORE DETAIL ON READ & DATA SEE CHAPTER 8.

If you typed everything correctly, your balloon is smoothly flying across the sky (page 72).

In order to understand what happened, first you need to know what picture making locations control the functions you need. These locations, called registers,could be illustrated in this manner:

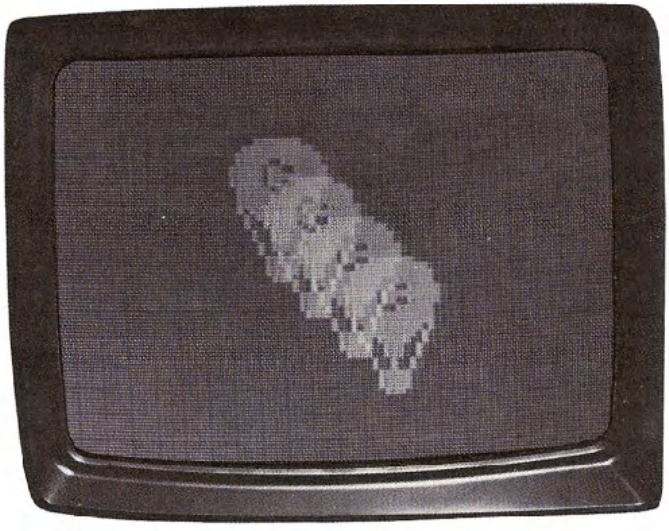
Register(s)	*Description*
0	X coordinate of sprite 0
1	Y coordinate of sprite 0
2 – 15	Paired like 0 and 1 for sprites 1-7
16	Most Significant Bit—X Coordinate
21	Sprite appear: 1=**appear** 0=**disappear**
29	Expand sprite in "X" Direction
23	Expand sprite in "Y" Direction
39 – 46	Sprite 0 – 7 color

In addition to this information you need to know from which 64 byte section sprites will get their data (1 byte is not used).

This data is handled by 8 locations directly after screen memory:

2040	41	42	43	44	45	46	2047
↑	↑	↑	↑	↑	↑	↑	↑
SPRITE 0	1	2	3	4	5	6	7

Now let's outline the exact procedure to get things moving and finally write a program.



ACTUAL SCREEN PHOTO

There are only a few things necessary to actually create and move an object.

1. Make the proper sprite(s) appear on the screen by POKEing into location 21 a 1 for the bit which turns on the sprite.
2. Set sprite pointer (locations 2040-7) to where sprite data should be read from.
3. POKE actual data into memory.
4. Through a loop, update X and Y coordinates to move sprite around.
5. You can, optionally, expand the object, change colors, or perform a variety of special functions. Using location 29 to expand your sprite in the "X" direction and location 23 in the "Y" direction.

There are only a few items in the program that might not be familiar from the discussion so far.

In line 10;

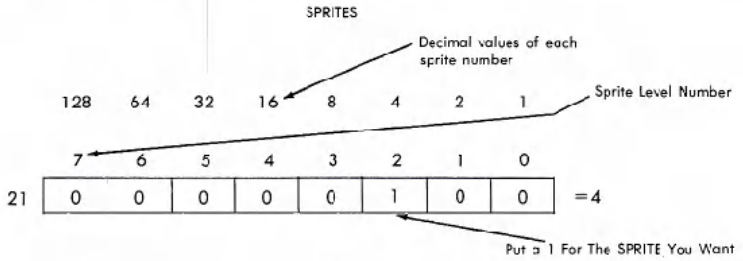
V=53248

sets V to the starting memory location of the video chip. In this way we just increase V by the memory number to get the actual memory location. The register numbers are the ones given on the sprite register map.

In line 11,

POKE V+21,4

makes sprite 2 appear by placing a 4 in what is called the sprite enable register (21) to turn on sprite 2. Think of it like this:



Each sprite level is represented in section 21 of the sprite memory and 4 happens to be sprite level 2. If you were using level 3 you would put a 1 in sprite 3 which has a value of 8. In fact if you used both sprites 2 and 3 you would put a 1 in both 4 and 8. You would then add the numbers together just like you did with the DATA on your graph paper. So, turning on sprites 2 and 3 would be represented as V+21,12.

In line 12;

POKE 2042,13

instructs the computer to get the data for sprite 2 (location 2042) from the 13th area of memory. You know from making your sprite that it takes up 63 sections of memory. You may not have realized it, but those numbers you put across the top of your grid equal what is known as 3 bytes of the computer. In other words each collection of the following numbers, 128,64,32,16,8,4,2,1 equals 1 byte of computer memory. Therefore with the 21 rows of your grid times the 3 bytes of each row, each sprite takes up 63 bytes of memory.

1 WHOLE SPRITE

```
20 FOR N = 0 to 62: READ Q: POKE 832+N,Q: NEXT
```

This line handles the actual sprite creation. The 63 bytes of data that represent the sprite you created are READ in through the loop and POKEd into the 13th block of memory. This starts at location 832.

```
30 FOR X = 0 TO 200
40 POKE V+4,X
50 POKE V+5,X
```

SPRITE 2's X COORDINATE

SPRITE 2's Y COORDINATE

If you remember from school the X coordinate represents an objects horizontal movement across the screen and the Y coordinate represents the sprite's vertical movement across the screen. Therefore as the values

of X change in line 30 from 0 to 200 (one number at a time) the sprite moves across the screen DOWN and TO THE RIGHT one space for each number. The numbers are READ by the computer fast enough to make the movement appear to be continuous, instead of 1 step at a time. If you need more details take a look at the register map in Appendix O.

When you get into moving multiple objects, it would be impossible for one memory section to update the locations of all eight objects. Therefore each sprite has its own set of 2 memory sections to make it move on the screen.

Line 70 starts the cycle over again, after one pass on the screen. The remainder of the program is the data for the balloon. Sure looks different on the screen, doesn't it?

Now, try adding the following line:

```
25 POKE V+23,4 : POKE V+29,4: REM EXPAND
```

and RUN the program again. The balloon has expanded to twice the original size! What we did was simple. By POKEing 4 (again to indicate sprite 2) into memory sections 23 and 29, sprite 2 was expanded in the X and Y direction.

It's important to note that the sprite will start in the upper left-hand corner of the object. When expanding an object in either direction, the starting point remains the same.

For some added excitement, make the following changes:

```
11 POKE V+21,12
12 POKE 2042,13 : POKE 2043,13
30 FOR X = 1 to 190
45 POKE V+6,X
55 POKE V+7,190-X
```

A second sprite (number 3) has been turned on by POKEing 12 into the memory location that makes the sprite appear (V+21). The 12 turns sprites 3 and 2 on (00001100 = 12).

The added lines 45 and 55 move sprite 3 around by POKEing values into sprite 3's X and Y coordinate locations (V+6 and V+7).

Want to fill the sky with even more action? Try making these additions:

28 IS REALLY 4 (SPRITE 2) + 8 (SPRITE 3) + 16 (SPRITE 4)

```
11 POKE V+21,28
12 POKE 2042,13:POKE 2043,13:POKE 2044,13
25 POKE V+23,12 : POKE V+29,12
48 POKE V+8,X
58 POKE V+9,100
```

In line 11 this time, another sprite (4) was made to appear by POKEing 28 into the appropriate "on" location of the sprite memory section. Now sprites 2-4 are on (00011100 = 28).

Line 12 indicates that sprite 4 will get its data from the same memory area (13th 63 section area) as the other sprites by POKEing 2044,13.

In line 25, sprites 2 and 3 are expanded by POKEing 12 (Sprites 2 and 3 on) into the X and Y direction expanded memory locations (V+23 and V+29).

Line 48 moves sprite 3 along the X axis. Line 58 positions sprite 3 halfway down the screen, at location 100. Because this value does not change, like it did before with X=0 to 200, sprite 3 just moves horizontally.

ADDITIONAL NOTES ON SPRITES

Now that you've experimented with sprites, a few more words are in order. First, you can change a sprite's color to any of the standard 16 color codes (0-15) that were used to change character color. These can be found in Chapter 5 or in appendix G.

For example, to change sprite 1 to light green, type: POKE V+40,13 (be sure to set V=53248).

You may have noticed in using the example sprite programs that the object never moved to the right-hand edge of the screen. This was because the screen is 320 dots wide and the X direction register can only hold a value up to 255. How then can you get an object to move across the entire screen?

There is a location on the memory map that has not been mentioned yet. Location 16 (of the map) controls something called the most significant bit (MSB) of the sprite's X direction location. In effect, this allows you to move the sprite to a horizontal spot between 256 and 320.

The MSB of X register works like this: after the sprite has been moved to X location 255, place a value into memory location 16 representing the sprite you want to move. For example, to get 2 to move to horizontal locations 256-320, POKE the value for sprite 2 which is (4) into memory location 16:

```
POKE V+16,4.
```

Now start from 0 again in the usual X direction register for sprite 2 (which is in location 4 of the map). Since you are only moving another 64 spaces, X locations would only range between 0 and 63 this time.

This whole concept is best illustrated with a version of the original sprite 1 program:

```
10 V= 53248: POKE V+21,4 : POKE 2042,13
20 FOR N = 0 TO 62 : READ Q : POKE 832+N,Q : NEXT
25 POKE V+5, 100
30 FOR X = 0 TO 255
40 POKE V+4,X
50 NEXT
60 POKE V+16,4
70 FOR X = 0 TO 63
80 POKE V+4, X
90 NEXT
100 POKE V+16,0
110 GOTO 30
```

Line 60 sets the most significant bit for sprite 2. Line 70 starts moving the standard X direction location, moving sprite 2 the rest of the way across the screen.

Line 100 is important because it "turns off" the MSB so that the sprite can start moving from the left edge of the screen again.

To define multiple sprites, you may need additional blocks for the sprite data. You can use some of BASIC's RAM by moving BASIC. Before typing or loading your program type:

```
POKE44,16:POKE16*256,0:NEW
```

Now, you can use blocks 32 through 41 (locations 2048 through 4095) to store sprite data.

BINARY ARITHMETIC

It is beyond the scope of this introductory manual to go into details of how the computer handles numbers. We will, however, provide you with a good base for understanding the process and get you started on sophisticated animation.

But, before you get too involved we have to define a few terms:

BIT—This is the smallest amount of information a computer can store.

Think of a BIT as a switch that is either "on" or "off". When a BIT is "on" it has a value of 1; when a BIT is "off" it has a value of 0.

After BIT, the next level is BYTE.

BYTE—This is defined as a series of BITS. Since a BYTE is made up of 8 BITS, you can actually have a total of 256 different combinations of BITS. In other words, you can have all BITS "off" so your BYTE will look like this:

128	64	32	16	8	4	2	1
0	0	0	0	0	0	0	0

and its value will be 0. All BITS "on" is:

128	64	32	16	8	4	2	1
1	1	1	1	1	1	1	1

which is 128+64+32+16+8+2+1=255.

The next step up is called a REGISTER.

REGISTER—Defined as a block of BYTES strung together. But, in this case each REGISTER is really only 1 BYTE long. A series of REGISTERS makes up a REGISTER MAP. REGISTER MAPS are charts like the one you looked at to make your BALLOON SPRITE. Each REGISTER controls a different function, like turning on the SPRITE is really called the ENABLE REGISTER. Making the SPRITE longer is the EXPAND X REGISTER, while making the SPRITE wider is the EXPAND Y REGISTER. Keep in mind that a REGISTER is a BYTE that performs a specific task.

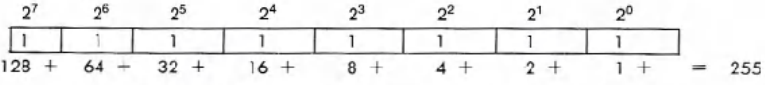
Now let's move on to the rest of BINARY ARITHMETIC.

BINARY TO DECIMAL CONVERSION

Decimal Value								
128	64	32	16	8	4	2	1	
0	0	0	0	0	0	0	1	2↑0
0	0	0	0	0	0	1	0	2↑1
0	0	0	0	0	1	0	0	2↑2
0	0	0	0	1	0	0	0	2↑3
0	0	0	1	0	0	0	0	2↑4
0	0	1	0	0	0	0	0	2↑5
0	1	0	0	0	0	0	0	2↑6
1	0	0	0	0	0	0	0	2↑7

Using combinations of all eight bits, you can obtain any decimal value from 0 to 255. Do you start to see why when we POKEd character or color values into memory locations the values had to be in the 0-255 range? Each memory location can hold a byte of information.

Any possible combination of eight 0's and 1's will convert to a unique decimal value between 0-255. If all places contain a 1 then the value of the byte equals 255. All zeros equal a byte value of zero; "00000011" equals 3, and so on. This will be the basis for creating data that represents sprites and manipulating them. As just one example, if this byte grouping represented part of a sprite (0 is a space, 1 is a colored area):



Then we would POKE 255 into the appropriate memory location to represent that part of the object.

TIP:

To save you the trouble of converting binary numbers into decimal values—we'll need to do that a lot—the following program will do the work for you. It's a good idea to enter and save the program for future use.

```
5 REM BINARY TO DECIMAL CONVERTER
10 INPUT "ENTER 8-BIT BINARY NUMBER :";A$
12 IF LEN (A$) <> 8 THEN PRINT "8 BITS PLEASE...":
   GOTO 10
15 TL = 0 : C = 0
20 FOR X = 8 to 1 STEP -1 : C = C + 1
30 TL = TL + VAL(MID$(A$,C,1))*2↑(X-1)
40 NEXT X
50 PRINT A$;" BINARY ";" = ";TL;" DECIMAL"
60 GOTO 10
```

This program takes your binary number, which was entered as a string, and looks at each character of the string, from left to right (the MID$ function). The variable C indicates what character to work on as the program goes through the loop.

The VAL function, in line 30, returns the actual value of the character. Since we are dealing with numeric characters, the value is the same as the character. For example, if the first character of A$ is 1 then the value would also be 1.

The final part of line 30 multiplies the value of the current character by the proper power of 2. Since the first value is in the 2^7 place, in the example, TL would first equal 1 times 128 or 128. If the bit is 0 then the value for that place would also be zero.

This process is repeated for all eight characters as TL keeps track of the running total decimal value of the binary number.