CHAPTER 5

ADVANCED COLOR AND GRAPHIC COMMANDS

- Color and Graphics
- PRINTing Colors
- Color CHR$ Codes
- PEEKs and POKEs
- Screen Graphics
- More Bouncing Balls

COLOR AND GRAPHICS

Up to now we've explored some of the sophisticated computing capabilities of the Commodore 64. But one of its most fascinating features is an outstanding ability to produce color and graphics.

You've seen a quick example of graphics in the "bouncing ball" and "maze" programs. But these only touched on the power you command. A number of new concepts will be introduced in this section to explain graphic and color programming and show how you can create your own games and advanced animation.

Because we've concentrated on the computing capabilities of the machine, all the displays we've generated so far were a single color (light blue text on a dark blue background, with a light blue border).

In this chapter we'll see how to add color to programs and control all those strange graphic symbols on the keyboard.

PRINTING COLORS

As you discovered if you tried the color alignment test in Chapter 1, you can change text colors by simply holding the **CTRL** key and one of the color keys. This works fine in the immediate mode, but what happens if you want to incorporate color changes in your programs?

When we showed the "bouncing ball" program, you saw how keyboard commands, like cursor movement, could be incorporated within PRINT statements. In a like way, you can also add text color changes to your programs.

You have a full range of 16 text colors to work with. Using the **CTRL** key and a number key, the following colors are available:

1	2	3	4	5	6	7	8
Black	White	Red	Cyan	Purple	Green	Blue	Yellow

If you hold down the **C=** key along with the appropriate number key, these additional eight colors can be used:

1	2	3	4	5	6	7	8
Orange	Brown	Lt. Red	Gray 1	Gray 2	Lt. Green	Lt. Blue	Gray 3

TYPE NEW, and experiment with the following. Hold down the **CTRL** key and at the same time hit the **1** key. Next, hit the **R** key without holding down the CTRL key. Now, while again depressing the CTRL key at the same time hit the 2 key. Release the CTRL key and hit the A key. Move through the numbers, alternating with the letters, and type out the word RAINBOW as follows:

```
10 PRINT" R A I N B O W"
         ↑ ↑ ↑ ↑ ↑ ↑ ↑
    CTRL 1 2 3 4 5 6 7
```

RUN
RAINBOW

Just as cursor controls show as graphic characters within the quote marks of print statements, color controls are also represented as graphic characters.

In the previous example, when you held down CTRL and typed 3 a "£" was displayed. CTRL 7 displayed a "←". Each color control will display its unique graphic code when used in this way. The table shows the graphic representations of each printable color control.

KEYBOARD	COLOR	DISPLAY	KEYBOARD	COLOR	DISPLAY
CTRL 1	BLACK	■	C= 1	ORANGE	♣
CTRL 2	WHITE	E	C= 2	BROWN	╱
CTRL 3	RED	£	C= 3	LT. RED	☒
CTRL 4	CYAN	◥	C= 4	GRAY 1	○
CTRL 5	PURPLE	▒	C= 5	GRAY 2	⁘
CTRL 6	GREEN	↑	C= 6	LT. GREEN	▌▌
CTRL 7	BLUE	←	C= 7	LT. BLUE	◆
CTRL 8	YELLOW	π	C= 8	GRAY 3	▚

Even though the PRINT statement may look a bit strange on the screen, when you RUN the program, only the text will be displayed. And it will automatically change colors according to the color controls you placed in the print statement.

Try a few examples of your own, mixing any number of colors within a single PRINT statement. Remember, too, you can use the second set of text colors by using the Commodore key and the number keys.

TIP:
You will notice after running a program with color or mode (reverse) changes, that the "READY" prompt and any additional text you type is the same as the last color or mode change. To get back to the normal display, remember to depress:

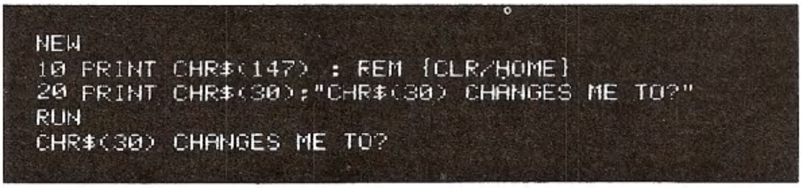
RUN/STOP and RESTORE

COLOR CHR$ CODES

Take a brief look at Appendix F, then turn back to this section.

You may have noticed in looking over the list of CHR$ codes in Appendix F that each color (as well as most other keyboard controls, such as cursor movement) has a unique code. These codes can be printed directly to obtain the same results as typing CTRL and the appropriate key within the PRINT statement.

For example, try this:



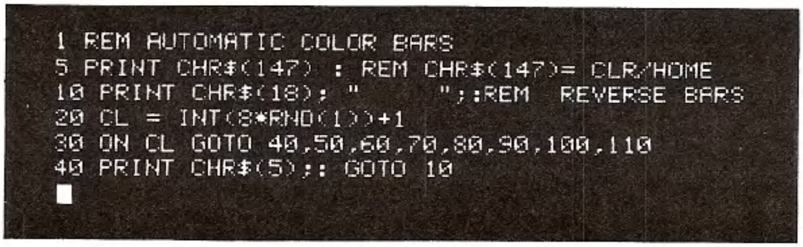
```
NEW
10 PRINT CHR$(147) : REM {CLR/HOME}
20 PRINT CHR$(30);"CHR$(30) CHANGES ME TO?"
RUN
CHR$(30) CHANGES ME TO?
```

The text should now be green. In many cases, using the CHR$ function will be much easier, especially if you want to experiment with changing colors. The following program is a different way to get a rainbow of colors. Since there are a number of lines that are similar (40-110) use the editing keys to save a lot of typing. See the notes after the listing to refresh your memory on the editing procedures.

```
NEW

1 REM AUTOMATIC COLOR BARS
5 PRINT CHR$(147) : REM CHR$(147)= CLR/HOME
10 PRINT CHR$(18); "       " ;:REM REVERSE BAR
20 CL = INT(8*RND(1))+1
30 ON CL GOTO 40,50,60,70,80,90,100,110
40 PRINT CHR$(5);: GOTO 10
50 PRINT CHR$(28);: GOTO 10
60 PRINT CHR$(30);: GOTO 10
70 PRINT CHR$(31);: GOTO 10
80 PRINT CHR$(144);: GOTO 10
90 PRINT CHR$(156);: GOTO 10
100 PRINT CHR$(158);: GOTO 10
110 PRINT CHR$(159);: GOTO 10
```

Type lines 5 through 40 normally. Your display should look like this:

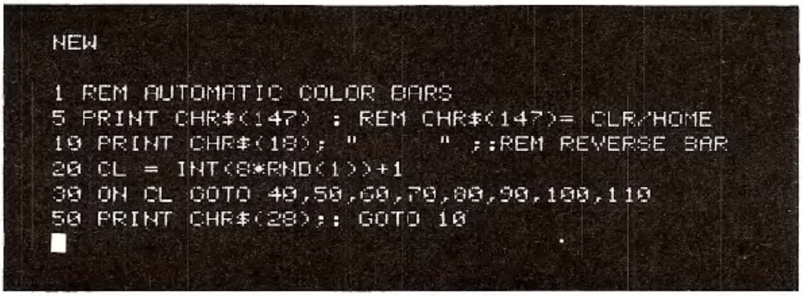


```
1 REM AUTOMATIC COLOR BARS
5 PRINT CHR$(147) : REM CHR$(147)= CLR/HOME
10 PRINT CHR$(18); "        ";:REM  REVERSE BARS
20 CL = INT(8*RND(1))+1
30 ON CL GOTO 40,50,60,70,80,90,100,110
40 PRINT CHR$(5);: GOTO 10
█
```

EDITING NOTES

Use the CRSR-UP key to position the cursor on line 40. Then type 5 over the 4 of 40. Next, use the CRSR-RIGHT key to move over to the 5 in the CHR$ parentheses. Hit SHIFT INST/DEL to open up a space and type '28'. Now just hit RETURN with the cursor anywhere on the line.

The display should now look like this:



```
NEW

1 REM AUTOMATIC COLOR BARS
5 PRINT CHR$(147) : REM CHR$(147)= CLR/HOME
10 PRINT CHR$(18); "        " ;:REM REVERSE BAR
20 CL = INT(8*RND(1))+1
30 ON CL GOTO 40,50,60,70,80,90,100,110
50 PRINT CHR$(28);: GOTO 10
█
```

Don't worry. Line 40 is still there. LIST the program and see. Using the same procedure, continue to modify the last line with a new line number and CHR$ code until all the remaining lines have been entered. See, we told you the editing keys would come in handy. As a final check, list the entire program to make sure all the lines were entered properly before you RUN it.

Here is a short explanation of what's going on.

You've probably figured out most of the color bar program by now except for some strange new statement in line 30. But let's quickly see what the whole program actually does. Line 5 prints the CHR$ code for CLR/HOME.

Line 10 turns reverse type on and prints 5 spaces, which turn out to be a bar, since they're reversed. The first time through the program the bar will be light blue, the normal text color.

Line 20 uses our workhorse, the random function to select a random color between 1 and 8.

Line 30 contains a variation of the IF . . . THEN statement which is called ON . . . GOTO. ON . . . GOTO allows the program to choose from a list of line numbers to go to. If the variable (in this case CL) has a value of 1, the first line number is the one chosen (here 40). If the value is 2, the second number in the list is used, etc.

Lines 40-110 just convert our random key colors to the appropriate CHR$ code for that color and return the program to line 10 to PRINT a section of the bar in that color. Then the whole process starts over again.

See if you can figure out how to produce 16 random numbers, expand ON . . . GOTO to handle them, and add the remaining CHR$ codes to display the remaining 8 colors.

PEEKS AND POKES

No, we're not talking about jabbing the computer, but we will be able to "look around" inside the machine and "stick" things in there.

Just as variables could be thought of as a representation of "boxes" within the machine where you placed your information, you can also think of some specially defined "boxes" within the computer that represent specific memory locations.

The Commodore 64 looks at these memory locations to see what the screen's background and border color should be, what characters are to be displayed on the screen—and where—and a host of other tasks.

By placing, "POKEing," a different value into the proper memory location, we can change colors, define and move objects, and even create music.

These memory locations could be represented like this:

53280	53281	53282	53283
X	Y		
BORDER COLOR	BACKGROUND COLOR		

On page 60 we showed just four locations, two of which control the screen and background colors. Try typing this:

POKE 53281,7 RETURN

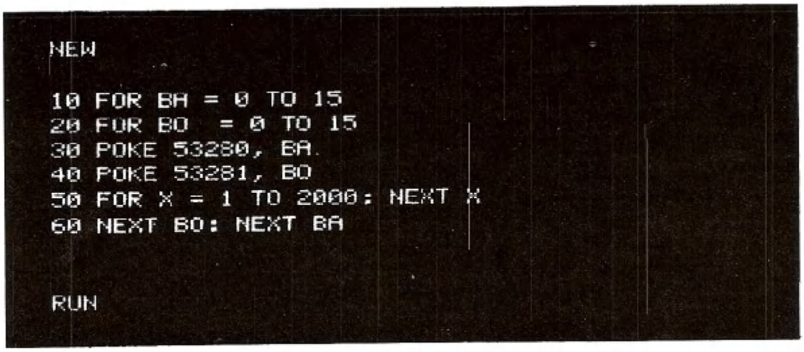
The background color of the screen will change to yellow because we placed the value '7'—for yellow—in the location that controls the background color of the screen.

Try POKEing different values into the background color location, and see what results you get. You can POKE any value between 0 and 255, but only 0 through 15 will work.

The actual values to POKE for each color are:

0	BLACK	8	ORANGE
1	WHITE	9	BROWN
2	RED	10	Light RED
3	CYAN	11	GRAY 1
4	PURPLE	12	GRAY 2
5	GREEN	13	Light GREEN
6	BLUE	14	Light BLUE
7	YELLOW	15	GRAY 3

Can you think of a way to display the various background and border combinations? The following may be of some help:



```
NEW

10 FOR BA = 0 TO 15
20 FOR BO  = 0 TO 15
30 POKE 53280, BA
40 POKE 53281, BO
50 FOR X = 1 TO 2000: NEXT X
60 NEXT BO: NEXT BA


RUN
```

Two simple loops were set up to POKE various values to change the background and border colors. The DELAY loop in line 50 just slows things down a bit.

For the curious, try:

? PEEK (53280) AND 15

You should get a value of 15. This is the last value BORDER was given and makes sense because both the background and border colors are GRAY (value 15) after the program is run.

By entering AND 15 you eliminate all other values except 1–15, because of the way color codes are stored in the computer. Normally you would expect to find the same value that was last POKEd in the location. In general, PEEK lets us examine a specific location and see what value is presently there. Can you think of a one line addition to the program that will display the value of BACK and BORDER as the program runs? How about this:

```
25 PRINT CHR$(147); "BORDER = ";PEEK (53280) AND 15, "BACK-
   GROUND = "; PEEK (53281) AND 15
```

SCREEN GRAPHICS

In all the printing of information that you've done so far, the computer normally handled information in a sequential fashion: one character is printed after the next, starting from the current cursor position (except where you asked for a new line, or used the ',' in PRINT formatting).

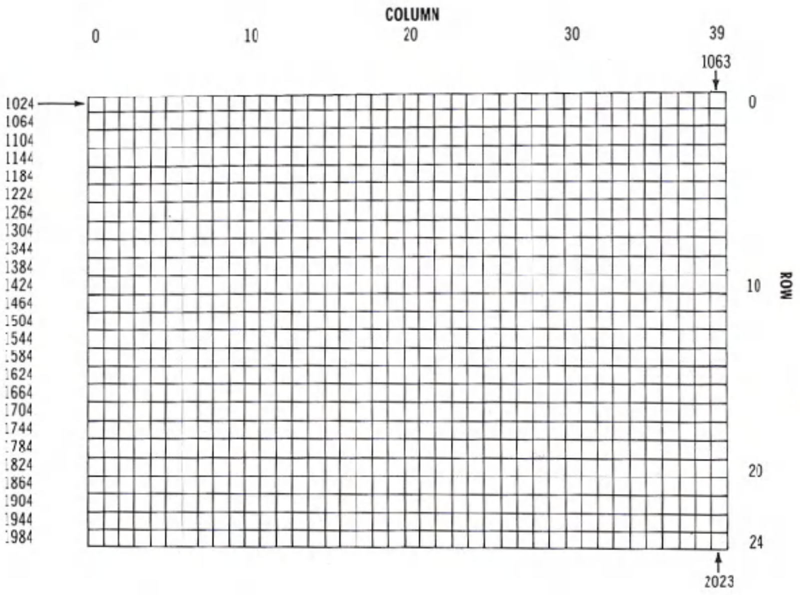
To PRINT data in a particular spot you can start from a known place on the screen and PRINT the proper number of cursor controls to format the display. But this takes program steps and is time consuming.

But just as there are certain spots in the Commodore 64's memory to control color, there are also locations that you can use to directly control each location on the screen.

SCREEN MEMORY MAP

Since the computer's screen is capable of holding 1000 characters (40 columns by 25 lines) there are 1000 memory locations set aside to handle what is placed on the screen. The layout of the screen could be thought of as a grid, with each square representing a memory location.

And since each location in memory can contain a number from 0 to 255, there are 256 possible values for each memory location. These values represent the different characters the Commodore 64 can display (see Appendix E). By POKEing the value for a character in the appropriate screen memory location, that character will be displayed in the proper position.



Screen memory in the Commodore 64 normally begins at memory location 1024, and ends at location 2023. Location 1024 is the upper left corner of the screen. Location 1025 is the position of the next character to the right of that, and so on down the row. Location 1063 is thė right-most position of the first row. The next location following the last character on a row is the first character on the next row down.

Now, let's say that you're controlling a ball bouncing on the screen. The ball is in the middle of the screen, column 20, row 12. The formula for calculation of the memory location on the screen is:

$$\textbf{POINT} = \textbf{1Ø24} + \overset{\text{COLUMN}}{\textbf{X}} + \textbf{4Ø}*\underset{\text{ROW}}{\textbf{Y}}$$

where X is the column and Y is the row.

Therefore, the memory location of the ball is:

$$\textbf{1Ø24} + \overset{\text{COLUMN}}{\textbf{2Ø}} + \underset{\text{ROW (40* 12)}}{\textbf{48Ø}} \text{ or } \textbf{1524}$$

Clear the screen with SHIFT and CLR/HOME and type:

```
POKE 1524,81
POKE 55796,1
```

COLOR

LOCATION

COLOR MEMORY MAP

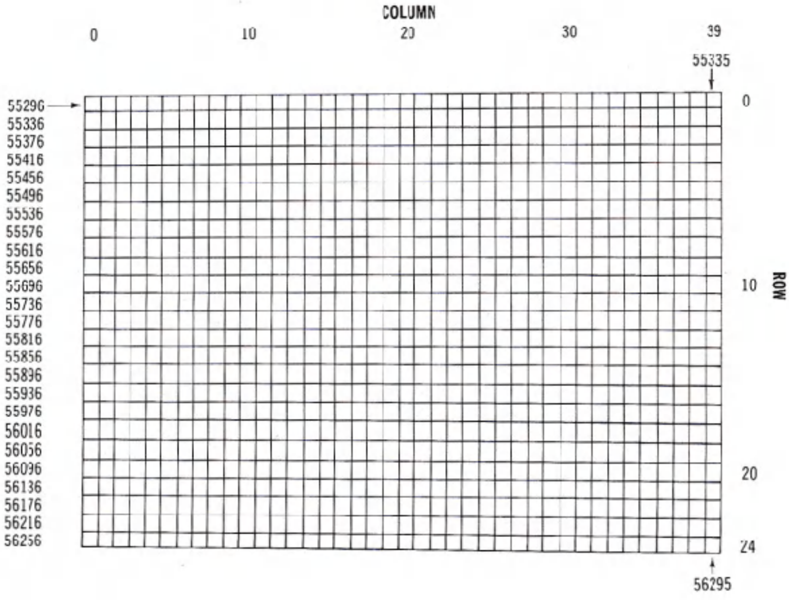
A ball appears in the middle of the screen! You have placed a character directly into screen memory without using the PRINT statement. The ball that appeared was white. However there is a way to change the color of an object on the screen by altering another range of memory. Type:

```
POKE 55796,2
```

LOCATION

COLOR

The ball's color changes to red. For every spot on the Commodore 64's screen there are two memory locations, one for the character code, and the other for the color code. The color memory map begins at location 55296 (top left-hand corner), and continues on for 1000 locations. The



same color codes, from 0-15, that we used to change border and background colors can be used here to directly change character colors.

The formula we used for calculating screen memory locations can be modified to give the locations to POKE color codes. The new formula is:

COLOR PRINT = 55296 − X + 4Ø*Y

MORE BOUNCING BALLS

Here's a revised bouncing ball program that prints directly on the screen with POKEs, rather than using cursor controls within PRINT statements. As you will see after running the program, it is much more flexible than the earlier program, and will lead up to programming much more sophisticated animation.

```
NEW

10 PRINT "{CLR/HOME}"
20 POKE 53280,7 : POKE 53281,13
30 X = 1 : Y = 1
40 DX = 1 : DY = 1
50 POKE 1024 + X + 40*Y,81
60 FOR T = 1 TO 10 : NEXT
70 POKE 1024 + X + 40*Y,32
80 X = X + DX
90 IF X <= 0 OR X >= 39 THEN DX = -DX
100 Y = Y + DY
110 IF Y <= 0 OR Y >= 24 THEN DY = -DY
120 GOTO 50
```

Line 10 clears the screen, and line 20 sets the background to light green with a yellow border.

The X and Y variables in line 30 keep track of the current row and column position of the ball. The DX and DY variables in line 40 are the horizontal and vertical direction of the ball's movement. When a **+1** is added to the X value, the ball is moved to the right; when **−1** is added, the ball moves to the left. A **+1** added to Y moves the ball down a row; a **−1** added to Y moves the ball up a row.

Line 50 puts the ball on the screen at the current cursor position. Line 60 is the familiar delay loop, leaving the ball on the screen just long enough to see it.

Line 70 erases the ball by putting a space (code 32) where the ball was on the screen.

Line 80 adds the direction factor to X. Line 90 tests to see if the ball has reached one of the side walls, reversing the direction if there's a bounce. Lines 100 and 110 do the same thing for the top and bottom walls.

Line 120 sends the program back to display and moves the ball again.

By changing the code in line 50 from 81 to another character code, you can change the ball to any other character. If you change DX or DY to 0 the ball will bounce straight instead of diagonally.

We can also add a little more intelligence. So far the only thing you checked for is the X and Y values getting out of bounds for the screen. Add the following lines to the program.

```
21 FOR L = 1 TO 10
25 POKE 1024 + INT(RND(1)*1000), 166 ← SCREEN CODE
27 NEXT L
85 IF PEEK(1024 + X + 40*Y) = 166 THEN DX = -DX:
   GOTO 80
105 IF PEEK(1024 + X + 40*Y) = 166 THEN DY = -DY:
    GOTO 100
```

Lines 21 to 27 put 10 blocks on the screen in random positions. Lines 85 and 105 check (PEEK) to see if the ball is about to bounce into a block, and changes the ball's direction if so.