55
REM A PORT TO THE ATARI VCS

CODING THE CHARACTERS
BUILDING THE WALLS
COVERING THE SCREEN
BOUNDING THE MAZE

Alongside the general purpose home computers launched in 1977—the TRS-80, the Apple II, and the Commodore PET—was another computer, one that was hugely successful but that most people do not recognize as a computer. This was a videogame console, the Atari Video Computer System (VCS), which later came to be known as the Atari 2600. Unlike the other computers, the Atari VCS was built specifically to play videogames. It was also designed to be far less expensive: the VCS was priced at $199, while the original Apple II cost an astounding $1,298.

Due to its intended use, the requirement that the system sell for a low price, and the high costs of silicon components, the Atari VCS was designed in a very unusual way. Like the Apple II and the Commodore 64, the Atari VCS used a version of the inexpensive MOS Technology 6502 microprocessor. But in order to create moving images and sounds on an ordinary CRT television, engineers Joe Decuir and Jay Miner designed a custom graphics and sound chip called the Television Interface Adapter (TIA). The TIA supported five "high resolution" movable objects: two player sprites (movable objects that can be created once and then moved around freely), two missiles (one for each player), and a ball. These were exactly the right kind of movable graphics needed for the games first envisioned for the VCS—home versions of popular Atari arcade games including *Pong* and *Tank*. The TIA also enabled a low-resolution playfield and a changeable background color, along with a variety of methods to vary the appearance of each of these objects. To save money, the TIA was paired with a cheaper variant of the 6502 and 128 bytes of RAM, an incredibly modest amount of memory.

Unlike the Apple II and the PET, the Atari had no on-board ROM and no operating system, and only a fraction of the RAM of those other 1977 computers. As a result, Atari programmers had to write code that manipulated the TIA's registers not merely on a screen-by-screen basis, but on every single scanline of the television display. The result is one of history's most unusual methods of producing computer graphics (Montfort and Bogost 2009, 28–30). The launch titles for the Atari VCS used this system in a fairly straightforward way (see figure 55.1), while later titles exploited it to produce quite different effects.

While a number of remarkable games were designed for the Atari VCS over its lifetime, the constraints of the system make it a particularly difficult platform from the programmer's perspective. Consider the chal-

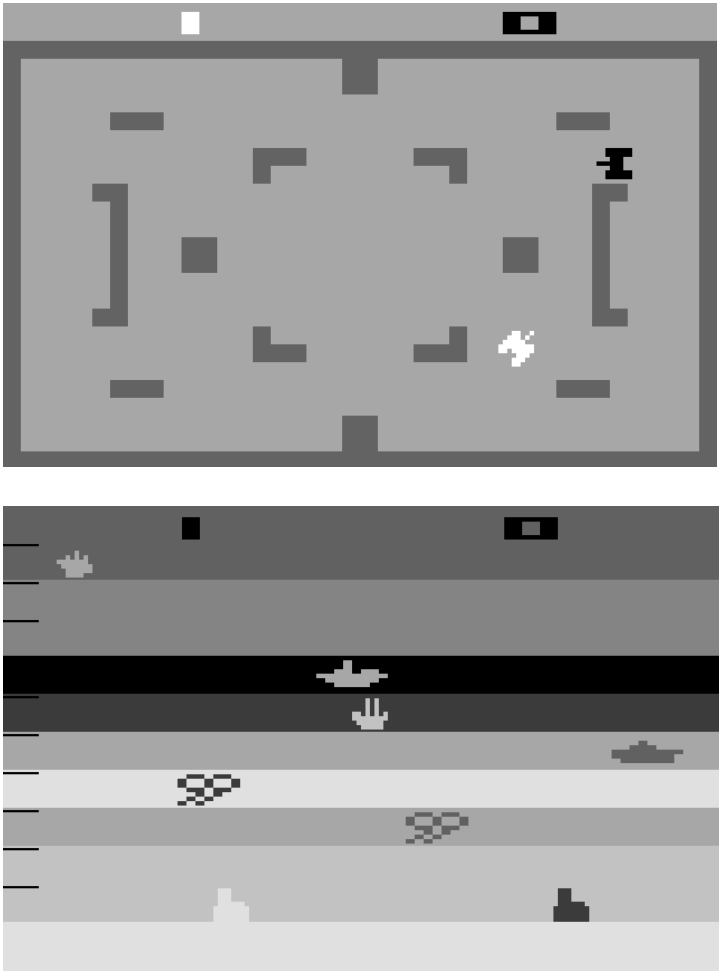


Figure 55.1
These screen captures from *Combat* (top) and *Air-Sea Battle* (below) show the visual quality of Atari VCS games.

lenges of porting `10 PRINT` to the Atari VCS:

1. The Atari does not have predefined character bitmaps, grids of pixels to represent each glyph, as the Commodore 64 does, making it necessary to create the patterns corresponding to the diagonal characters from scratch.

2. The TIA supports only two high-resolution sprites for on-screen display (the missiles and ball are mere dots, a pixel each). Somehow, the Atari has to be made to produce a large, changing pattern out of just these two 8-bit graphics registers.

3. The Atari has no concept of a row-and-column screen display like those found in minicomputer terminals and PCs. It was designed to play simple videogames, not to display text and numbers. As a result, the gridded layout that `10 PRINT` enjoys "for free," thanks to the Commodore 64's way of displaying text, must be laboriously simulated on the Atari VCS.

4. Once the previous hurdles are overcome, the Atari sports far less memory than the Commodore 64. The Commodore can hold all those display character references in memory because it has the room to do so, with 512 times as much storage as the Atari. Even if the Atari could be made to display enough high-resolution diagonal characters per line or per screen, the program would have to store references to those simulated characters so that each frame of the display would appear consistent with the preceding one.

Designing a port of `10 PRINT` for the Atari VCS is so quixotic that it might not seem to be worth even trying. Yet just as `10 PRINT` reveals much about BASIC and the Commodore 64, so too can a study of a seemingly impossible port on an incompatible platform reveal deeper levels to `10 PRINT`. Figure 55.2 shows is the closest approximation of `10 PRINT` that has been achieved on the Atari VCS, the output of a port written for this book.

CODING THE CHARACTERS

The matter of simulating PETSCII characters in the Atari's eight-bit graphics registers turns out to be the least troublesome challenge of the port. With the Commodore 64, graphical patterns that produce PETSCII characters are stored in ROM, and references in BASIC like `CHR$(205)` look up and

```
{198}  10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

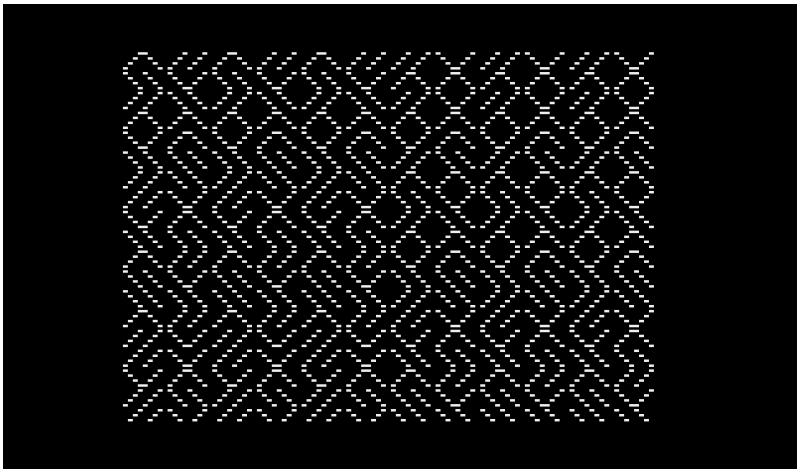
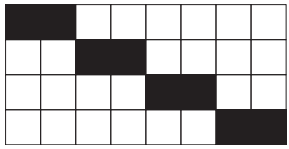


Figure 55.2
Screen capture from an Atari VCS port of `10 PRINT`.

retrieve the corresponding data for on-screen display, in a process all but invisible to the BASIC user. With the Atari, which has no ROM or built-in characters, it's necessary to "draw" the needed characters by defining a data table in the Atari's cartridge-based ROM. For example, the following data could be defined:

```
Diagonal
    .byte #%11000000
    .byte #%00110000
    .byte #%00001100
    .byte #%00000011
```

This binary data describes a left-leaning diagonal line, which would appear colored on screen wherever each bit of each byte of the bitmap is on:



This character looks satisfactory, but changes are necessary to eke out a credible rendition of `10 PRINT` on the Atari VCS. To understand why, it's important to consider the second and third challenges that were mentioned, the ones that are also the most troublesome.

The fact that TIA has only two 8-bit registers for displaying sprite graphics may come as a surprise to anyone who has played early Atari games, since many games appear to have more than two sprites on the screen at once. For example, *Air-Sea Battle*, one of the console's launch titles, depicts two antiaircraft guns at the bottom of the screen aimed up at seven rows of aeronautic enemies, each of which moves horizontally (figure 55.1). How is this possible?

The answer is strange but straightforward. It is typical to think of a computer display as a two-dimensional surface, like a painting or a photograph. Computers usually provide a block of video memory capable of storing enough information to create an entire screen's worth of display material. Typically the program resets this data during the brief moment before the 192 horizontal lines of a NTSC television screen are rescanned, a moment called the vertical blank. But the Atari has only 128 bytes of RAM total, making it impossible to set up a whole screen's worth of information at a time.

Instead, the Atari programmer sets up the display on a horizontal scanline-by-scanline basis, interfacing with the TIA to change its settings in the brief time between individual scanlines—a moment called horizontal blank. Once a particular line or group of lines is complete, the programmer can "reuse" the sprite registers later in the same screen, for a different purpose. The technique happens so fast, especially with the lingering glow of the television screen, that the reused sprites appear simultaneously, albeit with some flicker. This is exactly how the final `10 PRINT` port creates more than two "diagonal" graphics on the Atari's screen.

But games like *Air-Sea Battle* still only display one or two sprites on a single line—precisely because the TIA can display at most two player sprites. `10 PRINT` requires more than just two diagonals per row to look anything like a maze. The Commodore 64 screen can display forty columns of text; even half that number might be sufficient to give the sense of a maze, as evidenced by the VIC-20 version of `10 PRINT`, which runs on the VIC-20's twenty-two-column display and is discussed in the next chapter.

The two-sprite limitation leads to the third challenge that was stated

```
{200}  10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

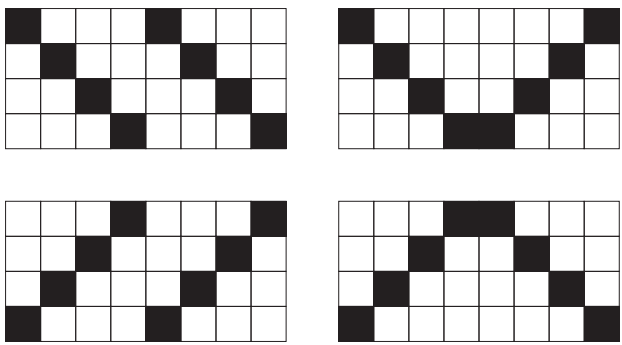
earlier: how to approximate the row-and-column display of the Commodore 64. Sprites may be reused on different horizontal sections of the television screen, which is helpful, but some way to display more than two columns worth of diagonals per row is needed. Three programming techniques, ranging from simple to complex, are required to produce an approximation of `10 PRINT`'s rows and columns of maze walls.

BUILDING THE WALLS

The simplest technique involves adjusting the sprite graphics to include two diagonals in eight bits of space rather than just one, each using one nybble (half-byte, or four bits). For example, this defines two left-leaning lines that are one pixel thick:

```
Diagonals
    .byte #%10001000
    .byte #%01000100
    .byte #%00100010
    .byte #%00010001
```

In working this way, there are four necessary permutations of two-line patterns to be encoded:



It's both easier and more efficient to store all four permutations as static data on the cartridge ROM than to try to construct them in RAM out of single diagonals, each one stores in half a byte—one-nybble diagonals.

This technique doubles the number of apparent diagonals per row, but with two sprites this still means only four diagonals—hardly a mazeworthy number. A second technique can be applied to triple that number, turning the individual diagonals into the walls of a maze.

The TIA provides a way to alter the appearance of each of the sprites automatically. These alterations include stretching the sprite to two times or four times its normal width, or doubling or tripling the sprite at different distances apart. In the VCS launch title *Combat*, many of the cartridge's plane game variants are accomplished simply by changing these settings for each player.

Stretching and multiplying the sprites is accomplished by writing specific values into special registers on the TIA chip called the Number-Size registers. By setting both registers to "three copies, closely spaced," it is possible to get six total sprites to appear on a single line of the display. Given that each sprite contains two diagonals, that's already twelve total simulated PETSCII characters per row. But, two problems remain: positioning and repetition.

COVERING THE SCREEN

To make a computer game of the sort normally played on the Atari, a programmer might expect to be able to position a sprite on a Cartesian coordinate system at a particular (x, y) position. As described earlier, the Atari doesn't give the programmer access to such a two-dimensional memory space, meaning there's no particular location where a sprite might appear on the screen. That said, the Atari does have something like a vertical axis; the programmer can count horizontal scanlines and choose to start or continue a sprite on a particular one.

To position an object horizontally, the programmer must manually "reset" the position of the object in question by strobing a register on the TIA. When any value is written into these registers (named RESP0 and RESP1 for the two player sprites), the TIA sets the starting horizontal position of that object at that point in the scanline. To accomplish this strange task, the programmer has to count the number of microprocessor cycles that will have passed before the television's electron gun has reached the desired position on the screen. Called "racing the beam" by Atari pro-


```
10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

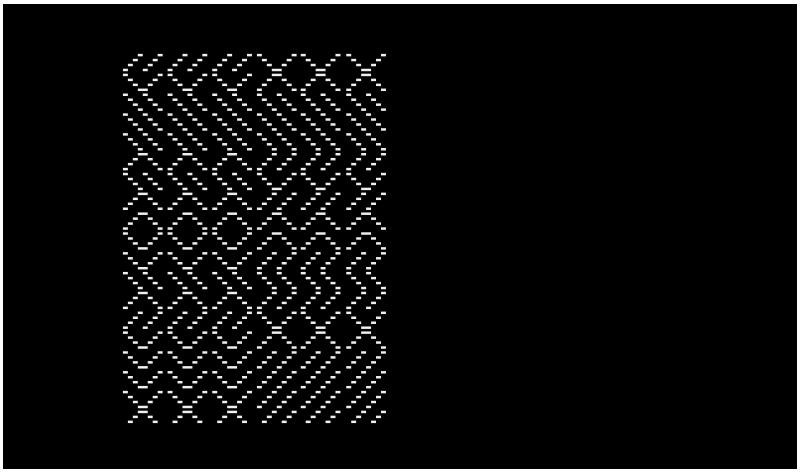


Figure 55.3
Identical copies of the diagonal pattern provide regularity rather than randomness.

grammers, this technique is relatively straightforward and can be used to position the two sprites next to one another, creating a sequence of six sets of two diagonals each.

The problem of repetition is more complex. When the TIA's number-size registers are set to triple a sprite, the result looks like three identical copies of the same pattern—whatever eight-bit value had been set in the sprite graphics register at the time the sprite was rendered to the screen. The resulting effect will be three identical copies of one diagonal pattern, followed by three identical copies of another diagonal pattern. This visual regularity (figure 55.3) is a serious problem, since the maze of `10 PRINT` is so strongly characterized by its apparent randomness. It's possible to overcome the visual repetition in the process of increasing the number of columns of sprites (and therefore diagonal lines) visible on a single row. Doing so involves taking advantage of an obscure behavior in the TIA.

When a sprite's number-size is set to double or triple, the TIA keeps an internal count of how many copies it has drawn. When the RESP0 or RESP1 is strobed, that value is reset. If that strobe occurs after the first copy is drawn but before the second has begun, the TIA's sprite counter is reset and it will start over, as if it hadn't yet drawn any copies of the sprites. By repeatedly strobing RESP0 and RESP1 in sequence, it is possible to pro-

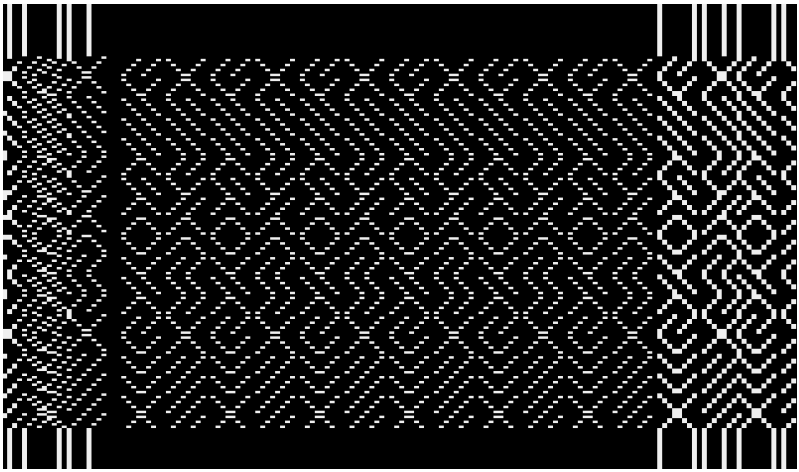


Figure 55.4
The Atari Television Interface Adapter wraps the characters around the screen. As this image shows, this is a problem for a `10 PRINT` port.

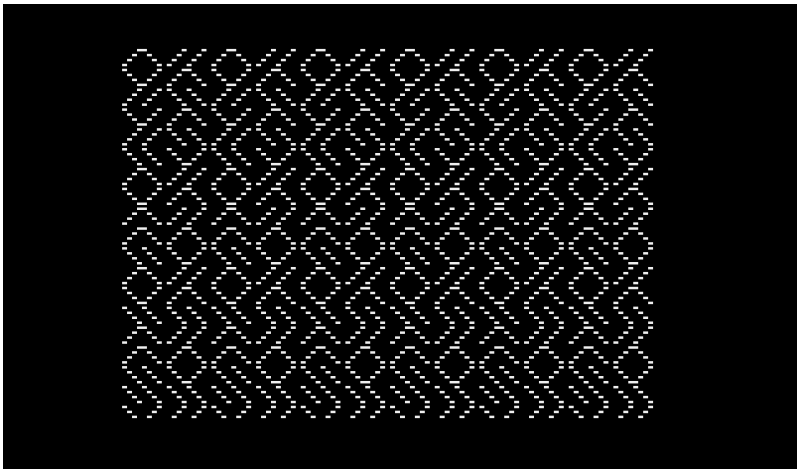


Figure 55.5
At this stage of the software development a convincing maze is generated, but the graphics are repeated and too regular in comparison to the original.

```
(204)  10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

duce a tight, interleaved grid of the sprites. By performing this trick over and over again, it's possible to easily produce a grid twelve sprites across.

This technique has the additional benefit of reducing the appearance of repetition, as two different sprite patterns can be interleaved. While a repeated pattern is still visible, it's not as obvious, and there are additional techniques available to further reduce the repetition.

The obstacle at this point, however, is that the screen has been set up to render twelve columns of alternating sets of sprites, each capable of displaying one of the four patterns of diagonals. But those twelve columns don't fill the whole screen. Centering them in the middle of the screen to mimic the borders of the Commodore 64 display creates a new problem: by the time the final sprite reset strobes have taken place, the maze "characters" are so far to the right side of the screen that they begin to overlap and wrap around on the borders (figure 55.4). This happens because the TIA automatically wraps sprites around the sides of the screen, a valuable technique for single-screen games like *Asteroids* but one that is of little use for a visual pattern partially defined by its borders.

BOUNDING THE MAZE

Luckily, low-resolution playfield graphics can hide the characters wrapping around the screen. Setting another bit on a TIA register will place the playfield in front of the sprites rather than behind it. This almost, but not quite, solves the problem. Timing the reset strobes just right leaves the twelve columns of sprites off center, so a small area of messy sprite junk is left at the right side of the pattern. The solution is the ball. Even though the name "ball" suggests a rounded image, to the TIA the "ball" is simply a square object of a single pixel that can be turned on or off. Turned on and positioned correctly, the ball will cover the offending sprite residue.

With all that work done, the fourth challenge remains: storing the diagonal pattern variation in what remains of the 128 bytes of RAM and loading the right data for each row of simulated PETSCII characters. Surprisingly, this is the least troubling task of all, although it does require more work than would be necessary on the Commodore 64. First it's necessary to write a random number-generation routine, since that function isn't provided in hardware on the machine. The next step is to write a routine that

will run the random number routine and use it to choose sets of diagonal bitmap data to use in each row of the visible display. This could be a lot of data, but it's not necessary to store the bitmaps themselves, just the sixteen-bit addresses of the ROM locations where they can be found. As it turns out, the program only requires eleven bytes of RAM to run everything else, leaving enough room in RAM to store twenty-nine rows worth of bitmap data pointers for each of the two sprites.

There is an unexpected consequence to this randomization approach. The Atari's random number generator has to be seeded somehow. It could be given a fixed seed, but in order to ensure that different seeds are chosen (resulting in different mazes), the program starts with a blank screen and increments a counter each frame. The user starts the program by depressing the console's RESET switch, at which time the frame counter is put to use as a random number seed. Every subsequent flick of the Reset switch will reset the seed and the diagonal graphics pointer data, resulting in a different maze. The result looks a great deal like the output of `10 PRINT`—it's clearly identifiable as some sort of port of the program (figure 55.5). It's even possible to make the rows scroll to mimic the Commodore 64's screen buffer, using a byte of RAM to store a memory offset location for the rows of bitmap data pointers.

But notice the horizontal symmetry of the upper part of the maze—the six diamonds spaced evenly across the top. This symmetry gives lie to the supposed randomness of the maze. It occurs because the same sprite data is used across the entire line of each row of the pattern. Recycling sprite data is necessary because the sprite reset strobing technique occurs so rapidly that it's impossible to alter the sprite graphics in between them. There's yet one more programming trick invented by Atari 2600 game designers that proves helpful here: flicker. Flicker is a common technique used on the Atari to give the player the impression that more objects appear on screen than are technically possible. It's a simple solution: when more than two objects need to seem to appear on a single scanline, draw some of them on one frame and the rest on another frame. The television screen is refreshed at 60Hz, so the result appears as a light flickering effect, like a ghost image. The result can be distracting or even disorienting, particularly when (as is not the case here) the objects are also moving.

The apparent regularity of the VCS port of `10 PRINT` can be reduced by deploying the flicker technique. On odd frames, render the first six col-

```
{206}  10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

umns with one set of diagonal patterns; on even frames, render the second six with another set of patterns. To do this, it's necessary to duplicate the loop that renders the screen and send the program to the correct one. Even this seemingly simple task proves difficult, since "turning off" half the pattern is not as easy as it sounds. It requires loading the processor's accumulator with the value zero and setting the two sprite graphics registers to that value at exactly the right time, before the TIA starts to render the next one. The result is convincing, even if it still doesn't look as random as the Commodore 64 original.

The technique used here is only one possible way to reproduce `10 PRINT`; other methods might allow for a more random display. For example, a common technique used in Atari games was a fairly complex routine for a six-digit score. By taking advantage of a setting called vertical delay, it's possible to push one sprite graphics value into the other by writing to the opposite register. This technique can produce six unique, closely spaced, high-resolution graphics. By combining this technique with the screen flickering approach discussed earlier, it might be possible to get a maze without any apparent repetition; but the careful cycle timing required to generate these patterns in exactly the correct place on the screen would also disrupt the evenness of the resulting maze. Violating the expected grid layout even slightly might make the "maze" look less mazelike.

The difficulty of creating the `10 PRINT` pattern on the Atari VCS is a reminder that computers with similar components from similar eras were designed to do very different things. `10 PRINT` depends on the Commodore 64's ability to render text in a line and screen buffer. Even though such abilities are fundamental to computers of the 1970s and 1980s, the Atari VCS was not designed with that usage in mind. The BASIC code `10 PRINT CHR$(205.5+RND(1)); : GOTO 10` is defined with text of 38 bytes; as is described in the next chapter, an assembly version of the program can be accomplished in less space. But the simplest version of the program on the Atari VCS requires 360 bytes, largely because the program has to perform "from scratch" so many functions that in the Commodore 64 are part of the ROM.

The very idea of creating a program like `10 PRINT` depends on aspects of the platform and the platform's facility for such a program—the presence of BASIC and `RND` in ROM, the existence of PETSCII, the cultural context of shared programming techniques, and of course the ability to

program the computer in the first place, something owners of an Atari 2600 did not truly have. Reimplementing the program on the Atari VCS, a platform both contemporaneous with the Commodore 64 and highly incompatible with the program that is this book's subject, helps to show all of the things the Commodore 64 programmer takes for granted. If the Commodore 64 programmer had to go to these lengths to produce the output of `10 PRINT`—from writing a random number generation routine to coercing a line-buffered display with two high-resolution objects to produce a two-dimensional grid of graphics—it's possible the program would never have been written.

```
(208)  10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```