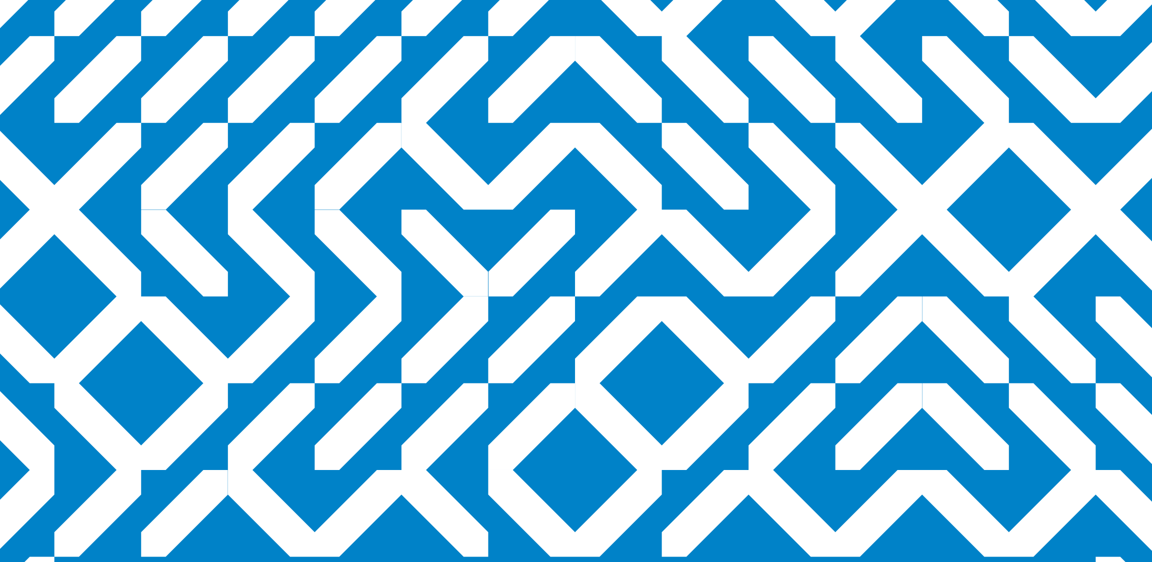
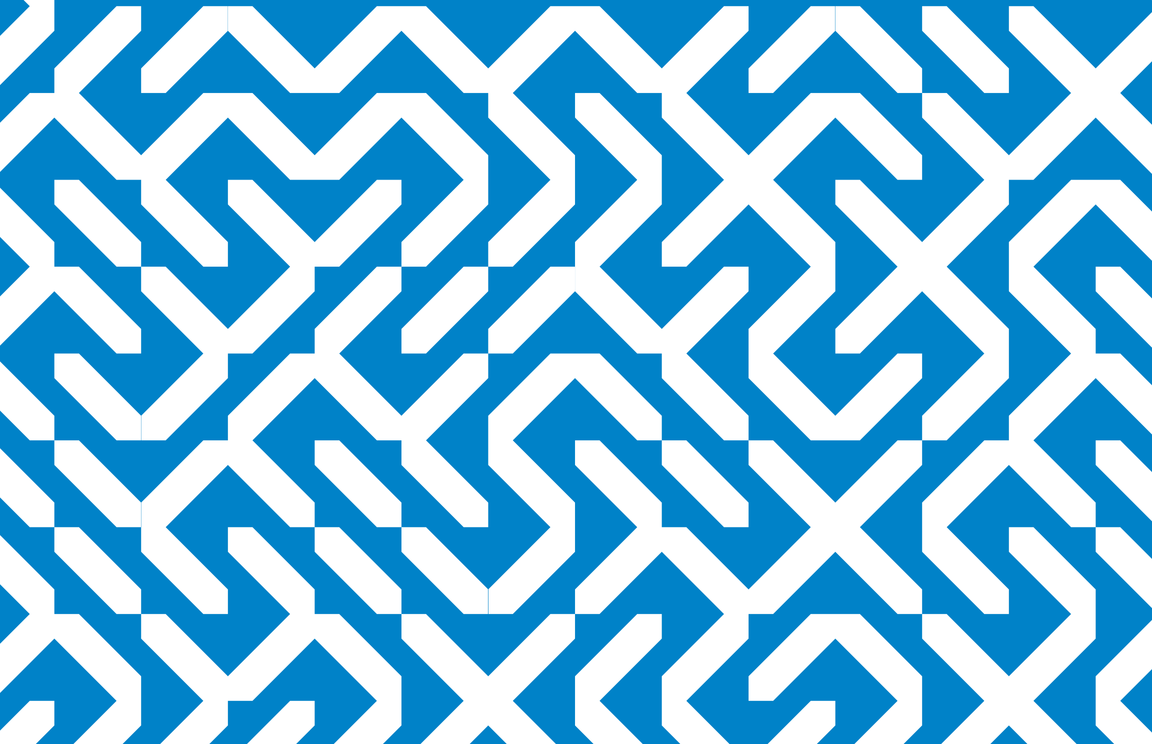
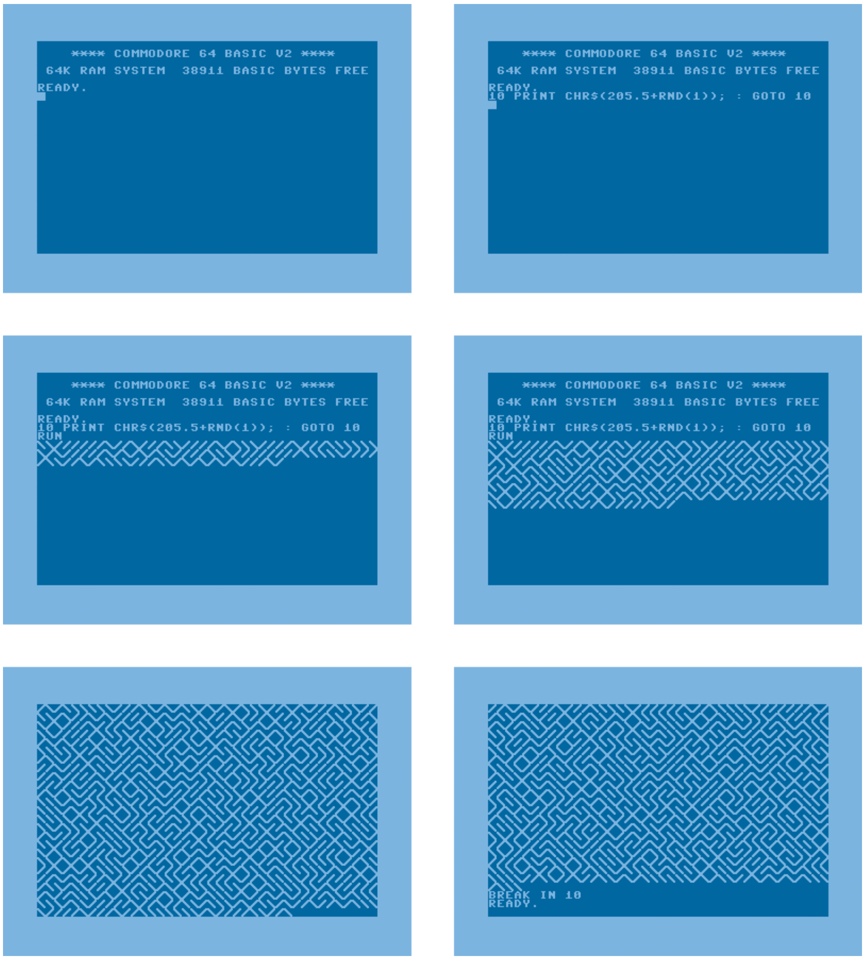


10 PRINT CHR$(205.5+RND(1)); : GOTO 10

NICK MONTFORT, PATSY BAUDOIN,
JOHN BELL, IAN BOGOST, JEREMY DOUGLASS,
MARK C. MARINO, MICHAEL MATEAS,
CASEY REAS, MARK SAMPLE, NOAH VAWTER



10 INTRODUCTION



```
    **** COMMODORE 64 BASIC V2 ****
 64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
```

```
    **** COMMODORE 64 BASIC V2 ****
 64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
10 PRINT CHR$(205.5+RND(1)); : GOTO 10
```

```
    **** COMMODORE 64 BASIC V2 ****
 64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
10 PRINT CHR$(205.5+RND(1)); : GOTO 10
RUN
```

```
    **** COMMODORE 64 BASIC V2 ****
 64K RAM SYSTEM  38911 BASIC BYTES FREE
READY.
10 PRINT CHR$(205.5+RND(1)); : GOTO 10
RUN
```

```
BREAK IN 10
READY.
```

Figure 10.1
From left to right and top to bottom, the `10 PRINT` program is typed into the Commodore 64 and is run. Output scrolls across the screen until it is stopped.

Computer programs process and display critical data, facilitate communication, monitor and report on sensor networks, and shoot down incoming missiles. But computer code is not merely functional. Code is a peculiar kind of text, written, maintained, and modified by programmers to make a machine operate. It is a text nonetheless, with many of the properties of more familiar documents. Code is not purely abstract and mathematical; it has significant social, political, and aesthetic dimensions. The way in which code connects to culture, affecting it and being influenced by it, can be traced by examining the specifics of programs by reading the code itself attentively.

Like a diary from the forgotten past, computer code is embedded with stories of a program's making, its purpose, its assumptions, and more. Every symbol within a program can help to illuminate these stories and open historical and critical lines of inquiry. Traditional wisdom might lead one to believe that learning to read code is a tedious, mathematical chore. Yet in the emerging methodologies of critical code studies, software studies, and platform studies, computer code is approached as a cultural text reflecting the history and social context of its creation. "Code . . . has been inscribed, programmed, written. It is conditioned and concretely historical," new media theorist Rita Raley notes (2006). The source code of contemporary software is a point of entry in these fields into much larger discussions about technology and culture. It is quite possible, however, that the code with the most potential to incite critical interest from programmers, students, and scholars is that from earlier eras.

This book returns to a moment, the early 1980s, by focusing on a single line of code, a BASIC program that reads simply:

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

One line of code, set to repeat endlessly, which will run until interrupted (figure 10.1).

Programs that function exactly like this one were printed in a variety of sources in the early days of home computing, initially in the 1982 *Commodore 64 User's Guide,* and later online, on the Web. (The published versions of the program are documented at the end of this book, in "Variants of 10 PRINT.") This well-known one-liner from the 1980s was recalled by one of the book's authors decades later, as discussed in "A Personal

Memory of 10 PRINT" in the BASIC chapter. This program is not presented here as valuable because of its extreme popularity or influence. Rather, it serves as an example of an important but neglected type of programming practice and a gateway into a deeper understanding of how computing works in society and what the writing, reading, and execution of computer code mean.

ONE LINE

This book is unusual in its focus on a single line of code, an extremely concise BASIC program that is simply called `10 PRINT` throughout. Studies of individual, unique works abound in the humanities. Roland Barthes's *S/Z,* Samuel Beckett's *Proust,* Rudolf Arnheim's *Genesis of a Painting: Picasso's Guernica,* Stuart Hall et al.'s *Doing Cultural Studies: The Story of the Sony Walkman,* and Michel Foucault's *Ceci n'est pas une pipe* all exemplify the sort of close readings that deepen our understanding of cultural production, cultural phenomena, and the Western cultural tradition. While such literary texts, paintings, and consumer electronics may seem significantly more complex than a one-line BASIC program, undertaking a close study of `10 PRINT` as a cultural artifact can be as fruitful as close readings of other telling cultural artifacts have been.

In many ways, this extremely intense consideration of a single line of code stands opposed to current trends in the digital humanities, which have been dominated by what has been variously called distant reading (Moretti 2007), cultural analytics (Manovich 2009), or culturomics (Michel et al. 2010). These endeavors consider massive amounts of text, images, or data—say, millions of books published in English since 1800 or a million Manga pages—and identify patterns and trends that would otherwise remain hidden. This book takes the opposite approach, operating as if under a centrifugal force, spiraling outward from a single line of text to explore seemingly disparate aspects of culture. Hence its approach is more along the lines of Brian Rotman's *Signifying Nothing* (1987), which documents the cultural importance of the symbol 0. Similarly, it turns out that in the few characters of `10 PRINT`, there is a great deal to discover regarding its texts, contexts, and cultural importance.

By analyzing this short program from multiple viewpoints, the book explains how to read code deeply and shows what benefits can come from such readings. And yet, this work seeks to avoid fetishizing code, an error that Wendy Chun warns about (2011, 51–54), by deeply considering context and the larger systems at play. Instead of discussing software merely as an abstract formulation, this book takes a variorum approach, focusing on a specific program that exists in different printed variants and executes on a particular platform. Focusing on a particular single-line program foregrounds aspects of computer programs that humanistic inquiry has overlooked. Specifically, this one-line program highlights that computer programs typically exist in different versions that serve as seeds for learning, modification, and extension. Consideration of `10 PRINT` offers new ways of thinking about how professional programmers, hobbyists, and humanists write and read code.

The book also considers how the program engages with the cultural imagination of the maze, provides a history of regular repetition and randomness in computing, tells the story of the BASIC programming language, and reflects on the specific design of the Commodore 64. The eponymous program is treated as a distinct cultural artifact, but it also serves as a grain of sand from which entire worlds become visible; as a Rosetta Stone that yields important access to the phenomenon of creative computing and the way computer programs exist in culture.

CORE CONTRIBUTIONS

The subject of this book—a one-line program for a thirty-year-old microcomputer—may strike some as unusual and esoteric at best, indulgent and perverse at worst. But this treatment of `10 PRINT` was undertaken to offer lessons for the study of digital media more broadly. If they prove persuasive, these arguments will have implications for the interpretation of software of all kinds.

First, to understand code in a critical, humanistic way, the practice of scholarship should include programming: modifications, variations, elaborations, and ports of the original program, for instance. The programs written for this book sketch the range of possibilities for maze generators within Commodore 64 BASIC and across platforms. By writing them, the `10 PRINT` program is illuminated, but so, too, are some of the main plat-

CRITICAL CODE STUDIES, SOFTWARE STUDIES, PLATFORM STUDIES

Critical Code Studies (CCS) is the application of critical theory and hermeneutics to the interpretation of computer source code, as defined by one of this book's authors (Marino 2006). During an online, collaborative conference, another of this book's authors challenged the 2010 Critical Code Studies Working Group to apply these methodologies to the one-line program that is this book's focus (Montfort 2010). Until then, a number of exemplary readings had taken up software and other encoded objects possessing considerably more code, clear social implications (for example, a knowledge base about terrorists), and more free space for writing of human significance in the form of comments or variable names. Members of the working group had demonstrated they could interpret a large program, a substantial body of code, but could they usefully interpret a very spare program such as this one? What followed, with some false starts, was a great deal of productive discussion, an article in *Emerging Language Practices* (Marino 2010), and eventually this book, with those who replied in the Critical Code Studies Working Group thread being invited to work together as coauthors.

CCS is a set of methodologies for the exegesis of code. Working together with platform studies, software studies, and media archaeology and forensics, critical code studies uses the source code as a means of entering into discussion about the technological object in its fullest context. CCS considers authorship, design process,

forms of home computing, as well as the many distinctions between Commodore 64 BASIC and contemporary programming environments.

Second, there is a fundamental relationship between the formal workings of code and the cultural implications and reception of that code. The program considered in this book is an aesthetic object that invites its authors to learn about computation and to play with possibilities: the importance of considering specific code in many situations. For instance, in order to fully understand the way that redlining (financial discrimination against residents of certain areas) functions, it might be necessary to consider the specific code of a bank's system to approve mortgages, not simply the appearance of neighborhoods or the mortgage readiness of particular populations.

This book explores the essentials of how a computer interprets code function, funding, circulation of the code, programming languages and paradigms, and coding conventions. It involves reading code closely and with sustained and rigorous attention, but is not limited to the sort of close reading that is detached from historical, biographical, and social conditions. CCS invites code-based interpretation that invokes and elucidates contexts.

This book also employs other approaches to the interpretation of technical objects and culture, notably software studies and platform studies. While software studies can include the consideration and reading of code, it generally emphasizes the investigation of processes, focusing on function, form, and cultural context at a higher level of abstraction than any particular code. Platform studies conversely focuses on the lower computational levels, the platforms (hardware system, operating system, virtual machines) on which code runs. Taking the design of platforms into account helps to elucidate how concepts of computing are embodied in particular platforms, and how this specificity influences creative production across all code and software for a particular system. This book examines one line of code as a means of discussing issues of software and platform.

In addition to being approaches, software studies and platform studies also refer to two book series from MIT Press. This book is part of the Software Studies series.

and how particular platforms relate to the code written on them. It is not a general introduction to programming, but instead focuses on the connection of code to material, historical, and cultural factors in light of the particular way this code causes its computer to operate.

Third, code is ultimately understandable. Programs cause a computer to operate in a particular way, and there is some reason for this operation that is grounded in the design and material reality of the computer, the programming language, and the particular program. This reason can be found. The way code works is not a divine mystery or an imponderable. Code is not like losing your keys and never knowing if they're under the couch or have been swept out to sea through a storm sewer. The working of code is knowable. It definitely *can* be understood with adequate time and effort. Any line of code from any program can be as thoroughly explicated as the eponymous line of this book.

Finally, code is a cultural resource, not trivial and only instrumental, but bound up in social change, aesthetic projects, and the relationship of people to computers. Instead of being dismissed as cryptic and irrelevant to human concerns such as art and user experience, code should be valued as text with machine and human meanings, something produced and operating within culture.

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

The pattern produced by this program is represented on the endpapers of this book. When the program runs, the characters appear one at a time, left to right and then top to bottom, and the image scrolls up by two lines each time the screen is filled. It takes about fifteen seconds for the maze to fill the screen when the program is first run; it takes a bit more than a second for each two-line jump to happen as the maze scrolls upward.

Before going through different perspectives on this program, it is useful to consider not only the output but also the specifics of the code—what exactly it is, a single token at a time. This will be a way to begin to look at how much lies behind this one short line.

10

The only line number is this program is 10, which is the most conventional starting line number in BASIC. Most of the programs in the *Commodore 64 User's Guide* start with line 10, a choice that was typical in other books and magazines, not only ones for this system. Numbering lines in increments of 10, rather than simply as 1, 2, 3, . . . , allows for additional lines to be inserted more easily if the need arises during program development: the lines after the insertion point will not have to be renumbered, and references to them (in `GOTO` and `GOSUB` commands) will not have to be changed.

The standard version of BASIC for the Commodore 64, BASIC version 2 by Microsoft, invited this sort of line numbering practice. Some extensions to this BASIC later provided a `RENUMBER` or `RENUM` command that would automatically redo the line numbering as 10, 20, 30, and so on.

This convenience had a downside: if the line numbers were spaced out in a meaningful way so that part of the work was done beginning at 100, another segment beginning at 200, and so on, that thoughtful segmentation would be obliterated. In any case, `RENUMBER` was *not* provided with the version of BASIC that shipped on the Commodore 64.

One variant of this program, which was published in the Commodore-specific magazine *RUN*, uses 8 as its line number. This makes this variant of the program more concise in its textual representation, although it does not change its function and saves only one byte of memory—for each line of BASIC stored in RAM, two bytes are allocated for the line number, whether it is 1 or the maximum value allowed, 63999. The only savings in memory comes from `GOTO 10` being shortened to `GOTO 8`. Any single digit including 1 and even 0 could have been used instead. Line number variation in the *RUN* variants attests to its arbitrariness for function, demonstrating that 10 was a line-numbering convention, but was not required. That 8 was both arbitrary and a specific departure from convention may then suggest specific grist for interpretation. For a one-line program that loops forever, it is perhaps appealing to number that line 8, the endlessly looping shape of an infinity symbol turned upon its side. However, whether the program is numbered 8 or 10, the use of a number greater than 0 always signals that `10 PRINT` (or `8 PRINT`) is, like Barthes's "work," "a fragment of substance," partial with potential for more to be inserted and with the potential to be extended (Barthes 1977, 142).

Why are typed line numbers required at all in a BASIC program? Programs written today in C, Perl, Python, Ruby, and other languages don't use line numbers as a language construct: they aren't necessary in BASIC either, as demonstrated by QBasic and Visual Basic, which don't make use of them. If one wants a program to branch to a particular statement, the language can simply allow a label to be attached to the target line instead of a line number. Where line numbers particularly helped was in the act of editing a program, particularly when using a line editor or without access to a scrolling full-screen editor. The Commodore 64 does allow limited screen editing when programming in BASIC: the arrow keys can be used to move the cursor to any visible line, that line can be edited, and the new version of the line can be saved by pressing RETURN. This is a better editing capability than comes standard on the Apple II, but there is still no scrollback (no ability to go back past the current beginning of the screen) in BASIC on the

Commodore 64. Line numbers provide a convenient way to get back to an earlier part of the program and to list a particular line or range of lines. Typing a line number by itself will delete the corresponding line, if one exists in memory. The interactive editing abilities that were based on line numbers were well represented even in very early versions of BASIC, including the first version of the BASIC that ran on the Dartmouth Time-Sharing System. Line numbers thus represent not just an organizational scheme, but also an interactive affordance developed in a particular context.

{SPACE}

The space between the line number 10 and the keyword `PRINT` is actually optional, as are all of the spaces in this program. The variant line `10PRINT CHR$(205.5+RND(1));:GOTO10` will function exactly as the standard `10 PRINT` with spaces does. The spaces are of course helpful to the person trying to type in this line of code correctly: they make it more legible and more understandable.

Even in this exceedingly short program, which has no variables (and thus no variable names) and no comments, the presence of these optional spaces indicates some concern for the people who will deal with this code, rather than merely the machine that will process it. Spaces acknowledge that the code is both something to be automatically translated to machine instructions and something to be read, understood, and potentially modified and built upon by human programmers. The same acknowledgment is seen in the way that the keywords are presented in their canonical form. Instead of `PRINT` the short form `?` could be used instead, and there are Commodore-specific two-character abbreviations that allow the other keywords to be entered quickly (e.g., `GOTO` can typed as G followed by SHIFT-O.) Still, for clarity, the longer (but easier-to-read) version of these keywords is shown in this program, as it is in printed variants.

PRINT

The statement `PRINT` causes its argument to be displayed on the screen. The argument to `PRINT` can take a variety of forms, but here it is a string that is in many ways like the famous string “HELLO WORLD.” In `PRINT “HELLO WORLD”` the output of the statement is simply the string literal, the text between double quotes. The string in the maze-generating program is generated by a function, and the output of each PRINT execution consists of only a single character, but it is nevertheless a string.

Today the PRINT command is well known, as are many similarly named print commands in many other programming languages. It is easy to overlook that, as it is used here, PRINT does not literally "print" anything in the way the word normally is used to indicate reproduction by marking a medium, as with paper and ink—instead, it displays. To send output to a printer, PRINT must be followed by # and the appropriate device number, then a comma, and then the argument that is to be printed. By default, without a device number, the output goes to the screen—in the case of the Commodore 64, a television or composite video monitor.

When BASIC was first developed in 1964 at Dartmouth College, however, the physical interface was different. Remarkably, the language was designed for college students to use in interactive sessions, so that they would not have to submit batch jobs on punch cards as was common at the time. However, the users and programmers at Dartmouth worked not at screens but at print terminals, initially Teletypes. A PRINT command that executed successfully did actually cause something to be printed. Although BASIC was less than twenty years old when a version of it was made for the Commodore 64, that version nevertheless has a residue of history, leftover terms from before a change in the standard output technology. Video displays replaced scrolls of paper with printed output, but the keyword PRINT remained.

CHR$

This function takes a numeric code and returns the corresponding character, which may be a digit, a letter, a punctuation mark, a space, or a "character graphic," a nontypographical tile typically displayed alongside others to create an image. The standard numerical representation of characters in the 1980s, still in wide use today, is ASCII (the American Standard Code for Information Interchange), a seven-bit code that represents 128 characters. On the Commodore 64 and previous Commodore computers, this representation was extended, as it often was in different ways on different systems. In extensions to ASCII, the other 128 numbers that can be represented in eight bits are used for character graphics and other symbols.

The Commodore 64's character set, which had been used previously on the Commodore PET, was nicknamed PETSCII.

The complement to CHR$ is the function ASC which takes a quoted character and returns the corresponding numeric value. A user who is curious about the numeric value of a particular character, such as the capital letter A, can type PRINT ASC("A") and see the result, 65. A program can also use ASC to convert a character to a numeric representation, perform arithmetic on the number that results, and then convert the new number back to a character using CHR$. In lowercase mode, this can be used to shift character between uppercase and lowercase, or this sort of manipulation might be used to implement a substitution cipher.

Character graphics exist as special tiles that are more graphical than typographical, more like elements of a mosaic than like pieces of type to be composed on a press. That is, they are mainly intended to be assembled into larger graphical images rather than "typeset" or placed alongside letters, digits, and punctuation. But these special tiles do exist in a typographical framework: a textual system, built on top of a bitmapped graphic display, is reused for graphical purposes. This type of abstraction may not be a smooth, clean way of accomplishing new capabilities, but it represents a rather typical way in which a system, adapted for a new, particular purpose, can be retrofitted to do something else.

(

CHR$ and RND are both functions, so the keyword is followed in both cases by an argument in parentheses. CHR$ ends with the dollar sign to indicate that it is a string function (it takes a numeric argument and returns a string), while RND does not, since it is an arithmetic function (it takes a numeric argument and returns a number). The parentheses here also make clear the order of arithmetic operations. For instance, RND(1-2) is the same as RND(-1), while RND(1)-2 is two subtracted from the whatever value is returned by RND(1).

205.5

All math in Commodore BASIC is done on floating point numbers (numbers with decimal places). When an integer result is needed (as it is in the case of CHR$), the conversion is done by BASIC automatically. If this value, 205.5, were to be converted into an integer directly, it would be truncated (rounded down) to become 205. If more than .5 and less than 1 is added to 205.5, the integer result will be 206.

This means the character printed will either be the one corresponding to 205 or the one corresponding to 206: ╲ or ╱. A quirk of the Commodore 64 character set is that these two characters, and a run of several character graphics, have two numeric representations. Characters 109 and 110 duplicate 205 and 206, meaning that 109.5 could replace 205.5 in this program and the identical output would be produced.

+

This symbol indicates addition, of course. It is less obvious that this is the addition of two floating point numbers with a floating point result; Commodore 64 BASIC always treats numbers as floating point values when it does arithmetic. The first number to be added is 205.5; the second is whatever value that `RND` returns, a value that will be between 0 and 1. On the one hand, because all arithmetic is done in floating point, figuring out a simple 2 + 2 involves more number crunching and takes longer than it would if integer arithmetic was used. On the other hand, the universal use of floating point math means that an easy-to-apply, one-size-fits-all mathematical operation is provided for the programmer by BASIC. Whether the programmer wishes to add temperatures, prices, tomato soup cans, or anything else, "+" will work.

The mathematical symbol "+" originated, like "&," as an abbreviation for "and." As is still conventional on today's computers, the Commodore 64 has a special "plus" or addition key but does not have any way to type a multiplication sign or a division sign. While they appear in some eight-bit codes that extend ASCII and in Unicode, the multiplication and division signs are absent from ASCII and from PETSCII. Instead, the asterisk (*) and the slash, virgule, or solidus (/) are used. Given the computer's development as a machine for the manipulation of numbers, it is curious that typographical symbols have to be borrowed from their textual uses ("*" indicating a footnote, "/" a line break or a juxtaposition of terms) and pressed into service as mathematical symbols. But this has to do with the history of computer input devices, which in early days included teletypewriters, devices that were not originally made for mathematical communication.

RND

This function returns a (more or less) random number, one which is between 0 and 1. The number returned is, more precisely, pseudorandom. While the sequence of numbers generated has no easily discernible pattern and is hard for a person to predict, it is actually the same sequence each time. This is not entirely a failing; the consistent quality of this "random" output allows other programs to be tested time and time again by a programmer and for their output to be compared for consistency.

It is convenient that the number is always between 0 and 1; this allows it to easily be multiplied by another value and scaled to a different range. If one wishes to pick between two options at random, however, one can also simply test the random value to see if it is greater than 0.5. Or, as is done in this program, one can add 205.5 and convert to an integer so that 205 is produced with probability 0.5 and 206 with probability 0.5.

More can be said about randomness, and much more is said in the chapter on the topic.

1

When RND is given any positive value (such as this 1) as an argument, it produces a number using the current seed. This means that when `RND(1)` is invoked immediately after startup, or before any other invocation of RND, it will always produce the same result: 0.185564016. The next invocation will also be the same, no matter which Commodore 64 is used or at what time, and the next will be the same, too. Since the sequence is deterministic, the pattern produced by the `10 PRINT` program, when run before any other invocation of RND, is a complex-looking one that is always the same.

;

Using a semicolon after a string in a `PRINT` statement causes the next string to be printed immediately after the previous one, without a newline or any spaces between them. Other options include the use of a comma, which moves to the next tab stop (10 spaces), or the use of no symbol at all, which causes a new line to be printed and advances printing to the next line. Although this use of the semicolon for output formatting was not original to BASIC, the semicolon was introduced very early on at Dartmouth, in version 2, a minor update that had only one other change. The semicolon here is enough to show that not only short computer programs like this one, but also the languages in which they are written, change over time.

:

The colon separates two BASIC statements that could have been placed on different lines. In a program like this on the original Dartmouth version of BASIC, each statement would have to be on its own line, since, to keep programs clear and uncluttered, only a single statement per line is allowed. The colon was introduced by Microsoft, the leading developer of microcomputer BASIC interpreters, as one of several moves to allow more code to be packed onto home computers.

GOTO

This is an unconditional branch to the line indicated—the program's only line, line 10. The `GOTO` keyword and line number function here to return control to an earlier point, causing the first statement to be executed endlessly, or at least until the program is interrupted, either by a user pressing the STOP key or by shutting off the power.

`GOTO`, although not original to BASIC, came to be very strongly associated with BASIC. A denunciation of `GOTO` is possibly the most-discussed document in the history of programming languages; this letter (discussed in the "Regularity" chapter) plays an important part in the move from unstructured high-level languages such as BASIC to structured languages such as ALGOL, Pascal, Ada, and today's object-oriented programming languages, which incorporate the control structures and principles of these languages.

RUN

Once a BASIC program is entered into the Commodore 64, it is set into motion, executed, by the `RUN` command. Until `RUN` is typed, the program lies dormant, full of potential but inert. RUN is therefore an essential token yet is not itself part of the program. RUN is what is needed to actualize the program.

In a similar fashion, describing the purpose of each of the twelve tokens in 10 PRINT does address the underlying complexity of the program. A token-by-token explanation is like a clumsy translation from BASIC into English, naively hewing to a literal interpretation of every single character. Translation can happen this way, of course, but it glosses over nuance, ambiguity, and most important, the cultural, computational, and historical depth hidden within this one line of code. Plumbing those depths is precisely the goal of the rest of this book. The rest of this book is the RUN to the introduction here. So, as the Commodore 64 says . . .

READY.

PLAN OF THE BOOK

The more general discussions in this book are organized in five chapters and a conclusion. Preceding each of the five chapters and before the conclusion are six "Remarks." These are more specific discussions of particular computer programs directly related to 10 PRINT; they are programs that the authors have found or (in the spirit of early Commodore 64 BASIC programmers, who were encouraged to modify, port, and elaborate code and who often did so) ones that the authors have developed to shed light on how 10 PRINT works. These remarks are indicated with "REM" to refer to the BASIC statement of that name, one that allows programmers to use a line of a program to write a remark or comment, such as 55 REM START OF MAIN LOOP.

The first chapter, Mazes, offers the cultural context for reading a maze pattern in 1982. The chapter plumbs cultural and scientific associations with the maze and some of the history of mazes in computing as well. Regularity, the second chapter, considers the aspects of 10 PRINT that repeat in space, in time, and in the program's flow of control. The aesthetic and computational nature of repetition is discussed as well as the interplay between regularity and randomness. The third chapter, Randomness, offers a look at cultural uses and understandings of randomness and chance, as they are generated in games, by artists, and in simulations. It aims to show that behind a simple, commonly used capability of the computer lie numerous historical associations and uses, from the playful to the extraordinarily violent. BASIC, the fourth chapter, explains the origins of BASIC and describes how this language came to home computing. The ways in which short BASIC programs were circulated is also discussed. The fifth chapter, The Commodore 64, delves into the computer's history, exploring the machine on which `10 PRINT` runs. The most relevant technical topics, including the PETSCII character set, the VIC-II video chip, and the KERNAL (the Commodore 64's operating system, stored in 8K of ROM) are also discussed. This chapter situates `10 PRINT` in the context of its platform and that platform's rich cultural contexts.

The remarks reflect on a series of slight variations in the original BASIC program, all of which are also in Commodore 64 BASIC; on ports of `10 PRINT` to different languages and computers; on several ports and elaborations of `10 PRINT` on the Processing platform; on a collection of one-liners, including some Commodore 64 BASIC one-liners found in early 1980s print sources; on an Atari VCS port of the program; and on some greatly elaborated versions of the program in Commodore 64 BASIC. The last remark includes elaborations that generate stable full-screen mazes, allow a user to navigate a symbol around those mazes, and test those generated mazes for solubility.

One line of code gives rise here to an assemblage of readings by ten authors, offering a hint of what the future could hold—should personal computers once again invite novice programmers to `RUN`.

15 REM VARIATIONS IN BASIC

Even small changes to the `10 PRINT` code can have a significant impact on the visual output and the pattern produced. The output of `10 PRINT` has a unique visual appeal that can be understood in terms of design (a diagonal vs. an orthogonal composition, for instance), and in terms of how it plays against the contextual expectations of the historical period when it emerged (all-text BASIC programs on the one hand and graphical software, particularly videogames, on the other).

To understand more about this, it's possible not only to read the program the way one might go over a poem or other literary text, but also to modify the program and see what happens, as the *Commodore 64 User's Guide* and *RUN* magazine explicitly invite programmers to do. Writing code can be a method of reading it more closely, as was recognized decades ago. The text accompanying the first two printed variants suggested modifying the distribution of characters (in *Commodore 64 User's Guide*) and adding code to cause random color changes (in the magazine *RUN*). This section shows the results of doing the first of these, explores what happens if other PETSCII characters are chosen for display, and finally gives a one-line variation that uses `POKE` to directly write to screen memory.

As tweaking the program will show, `10 PRINT` is a kind of optimal solution that is uniquely elegant in its design space, that of the Commodore 64 BASIC one-line maze generator. Any similar attempt is both less concise (it requires more code) and less expressive (it resembles a maze less or produces a less interesting visual pattern). In fact, the concision of the code and the expressiveness of the image are tightly related. They arise out of a unique set of constraints and interactions, particularly the interaction between the desire to constrain the program code to a single line and the sequence of adjacent characters in the PETSCII table.

EMULATING THE COMMODORE 64

The Commodore 64 was an extremely popular computer; many millions of units were sold and many remain in working condition. It is still possible to cheaply acquire a Commodore 64, hook it to a television, and operate it as users of the 1980s did. When one's goal is to provide a classroom of students with access to the platform, however, or when one wishes to be able to play with and program for the Commodore 64 in many different locations on one's own contemporary notebook computer, there is a more practical alternative to finding, setting up, and starting up the classic taupe unit.

This alternative is a Commodore 64 emulator, a software version of the computer that runs on contemporary hardware and functions in the way the original Commodore 64 did. In 1983, a Commodore 64 could be purchased for $600. Today, for those who already have Internet-connected computers, it costs nothing to download and use an emulator. Emulators have been disparaged as inadequate attempts to mimic computers; while they do not capture the material aspects of older computers, they need not be considered as poor substitutes. Instead, an emulator can be usefully conceptualized as an edition of a computer.

When developers produce a program, such as the free software emulator VICE, that operates like a Commodore 64, it can be considered as a software edition of the Commodore 64. It isn't an official or authorized edition—only being a product of Commodore would allow for that. (There are official, authorized emulators for some systems, but VICE and many of the most frequently used emulators are not official.) An emulator like this is an attempt—more or less successful—to produce a system that functions like a Commodore 64. The development of an emulator typically takes a great deal of effort and can be extremely effective, as it is in the case of VICE. Thinking of this as an edition of the system seems to be a useful way to frame emulation, as it allows users to compare editions and usefully understand differences and similarities. Some emulators (like some editions) may be better for teaching, for casual reading or play, or for research and study. Instead of dismissing the emulator as useless because it isn't the original hardware, it makes more sense to consider how it works and what it affords, to look at what sort of edition it is.

The BASIC programs printed in this chapter can be run on a Commodore 64 emulator. The reader is encouraged to download an emulator, run the programs, and imagine how various differences between emulation and the original hardware influence the experience. For instance, the modern PC keyboard does not have the Commodore 64 graphics characters printed on the keys, and mapping the Commodore 64 keys to a modern keyboard layout is not straightforward. Graphically, a composite video monitor or television display attached to a Commodore 64 do not function exactly like a modern LED flat panel; the pixels drawn by an emulator are

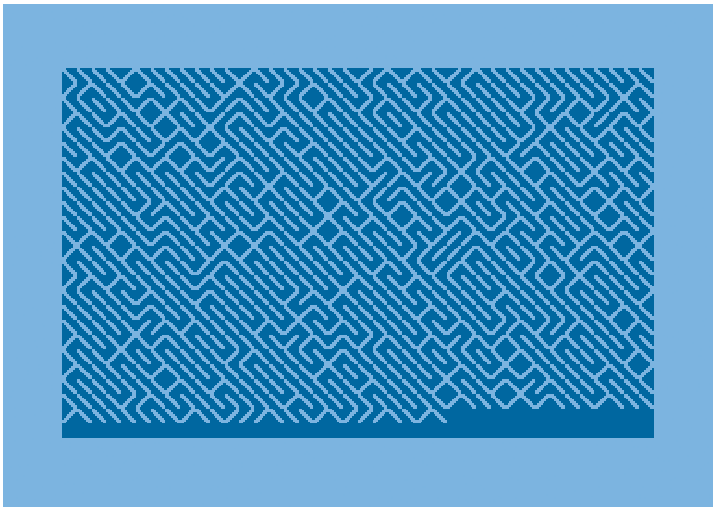


Figure 15.1

```
10 PRINT CHR$(205.25+RND(1)); : GOTO 10
```

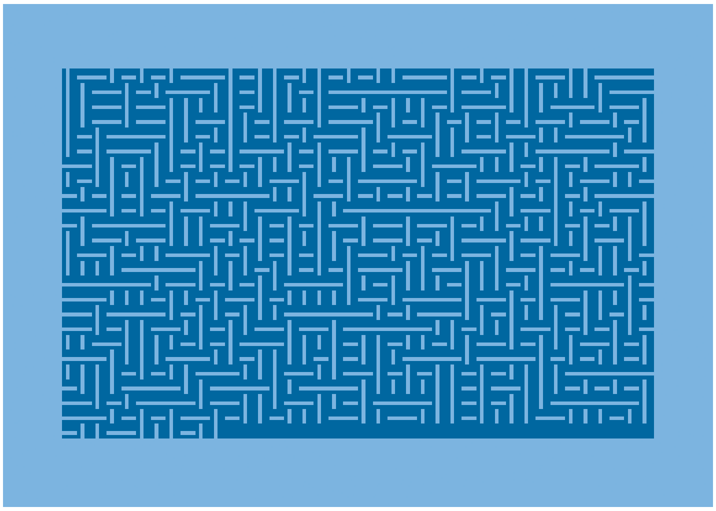


Figure 15.2

```
10 PRINT CHR$(198.5+RND(1)); : GOTO 10
```

overly crisp when compared to those seen on an early display. An emulator lets the user to save the current state of memory, registers, and so on more easily than BASIC programs can be saved to and loaded from disk on the hardware Commodore 64.

UNBALANCED

The *Commodore 64 User's Guide* encourages users to modify its version of `10 PRINT` in this way: “If you'd like to experiment with this program, try changing 205.5 by adding or subtracting a couple tenths from it. This will give either character a greater chance of being selected” (1982, 53).

Figure 15.1 shows the effect of changing the ".5" to ".25." As one diagonal predominates, the perceived architecture of the maze tends to long corridors along that direction. More extreme variations, such as going to or beyond 0.95 or below 0.05, present what looks like a regular diagonal pattern with a very few lines going the other way, as if they were occasional defects.

WEAVE

There are no other adjacent characters in the PETSCII data set that, when substituted for the diagonal ╲ and ╱, will result in the construction of a traditional orthogonal maze, one that is aligned to the vertical and horizontal axes of the screen. Using vertical and horizontal bars, for example, results in a disconnected weave (figure 15.2), while solid and empty squares result in a pattern similar to rough static.

Though the result certainly does not suggest a maze as strongly, this "Weave" version of the program is not without visual interest. The output imparts a three-dimensional impression, as if someone had woven bands of material over and under one another.

CORNERS

The Commodore 64 PETSCII character set includes corner characters, such as 204 and 207, which correspond to lower-left and upper-right corner pieces. Randomly selecting either 204 or 207, as is done in this program, produces an image similar to a honeycomb. Diagonal mazes are particularly efficient ones to produce on a Cartesian grid. If a diagonal line is used, four characters can meet at the corners, whereas only two meet along an edge when tiles touch left-to-right or top-to-bottom. This pattern (see figure 15.3) does not offer as many meeting points, but has some of its own interesting visual properties.

CORNERS AND DIAGONALS

A simplification of the program above involves dropping the `INT` function, so that the program chooses at random between other characters in addition to 204 and 207, the two corners; this "Corners and Diagonals" version can also choose the two characters in between. These characters are, of course, 205 and 206, which are the ╲ and ╱ characters that are invoked by `10 PRINT`. The result (see figure 15.4) does not have the clear structure of the `10 PRINT` maze and its pathways run for shorter stretches, appearing to be blocked more frequently. Nevertheless, the pattern that is produced is somewhat compelling in its confusion of elements.

FOUR WALLS

A reasonably intuitive method of constructing a maze-grid is to fill in one edge of each square on a sheet of graph paper. That is, when considering any specific square, fill in the top, right, bottom, or left to form a "wall," then move to the next square and repeat. The four characters in this program correspond to a top-wall, bottom-wall, left-wall or right-wall. Such characters exist in PETSCII in both "thick" and "thin" variants; the ones used in figure 15.5 are the thick ones. Such a process is unfortunately less elegant, as these characters are not (in either variety) placed adjacent to one another in the PETSCII character set—for instance, the ones used here

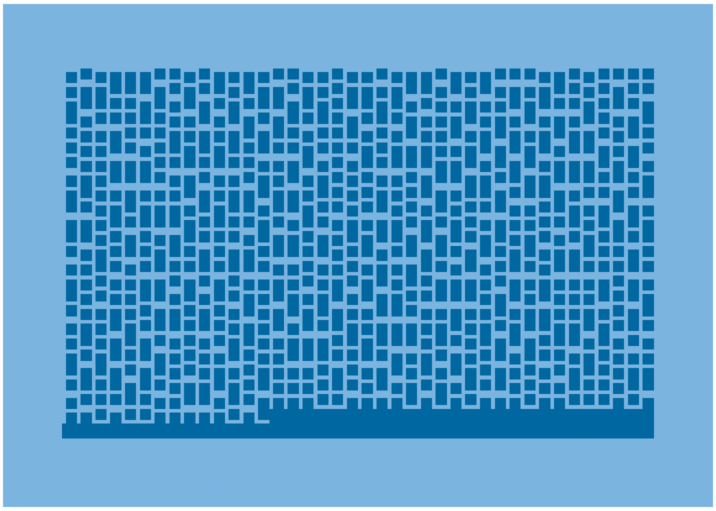


Figure 15.3

```
10 PRINT CHR$(204+(INT(RND(1)+.5)*3)); : GOTO 10
```

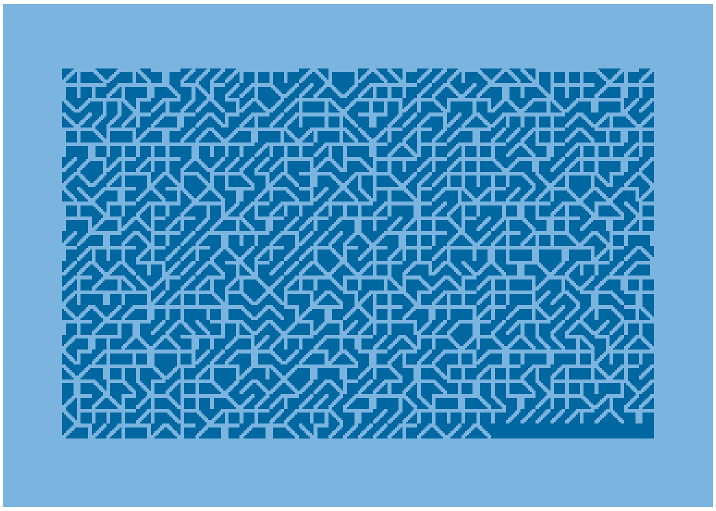


Figure 15.4

```
10 PRINT CHR$(204+(RND(1)+.5)*3); : GOTO 10
```

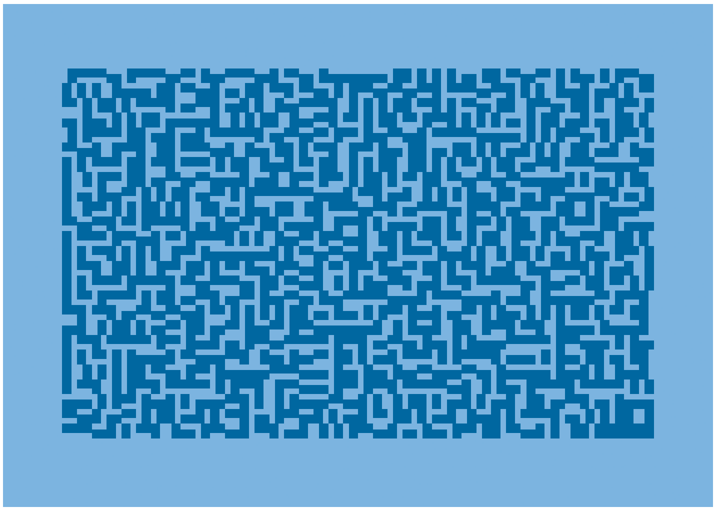


Figure 15.5

```
10 PRINT CHR$(181+(INT(RND(1)+.5)*3)+(INT(RND(1)+.5))); : GOTO 10
```

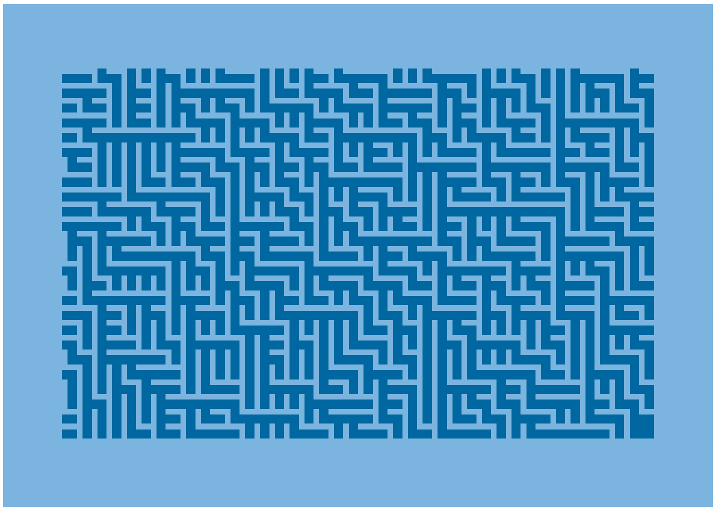


Figure 15.6

```
10 PRINT CHR$(181+(INT(RND(1)+.5)*3)); : GOTO 10
```

are 181, 182, 184, and 185—and so cannot be addressed with a single base value plus an offset, as was done in the previous program.

The image that emerges is indeed mazelike, but this image, like the underlying code, lacks simplicity and elegance. Since top and bottom and left and right lines can be printed up against each other, a variation in the thickness of the walls appears—a noticable but potentially distracting implication of messiness and texture.

TWO WALLS

The selection of characters 181 and 184, a thick left line and thick top line (figure 15.6), provides the best approximation of the classic orthogonal maze that is seen in arcade, console, and computer games. Producing it is still less elegant than selecting between 205 and 206 as PETSCII values. The characters used are not adjacent, so some trick, such as this one involving the use of `INT`, must be used to select one of the two at random. The resulting output is less visually interesting. It is a maze, but is both less formally dynamic (being aligned to the screen) and less contextually unexpected (being typical of familiar game mazes).

POKE

A similar maze pattern can be drawn by directly placing characters in video memory using the `POKE` command, which writes directly to memory—screen memory, in this case, which is mapped to the decimal addresses 1024–2024 (see figure 15.7). The `1024+RND(1)*1000` selects a random number in this range as the first argument to `POKE`, pointing that command at some specific location on the screen. The `77.5+RND(1)` selects ╱ or ╲. It should seem odd that after using 205.5 (and thus the values 205 and 206) to refer to these two characters, this program refers to them using the values 77 and 78. It is, indeed, odd. This difference is due to the PETSCII codes for characters not corresponding to their *screen codes*—each character has a different address for `PRINT`ing and for `POKE`ing into screen memory. This rather esoteric feature of the Commodore 64 is discussed in the final chapter of this book, The Commodore 64.

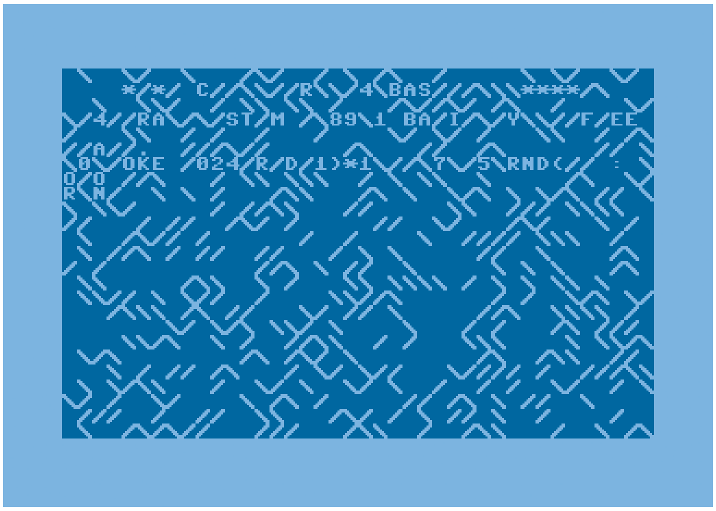


Figure 15.7

`10 POKE 1024+RND(1)*1000,77.5+RND(1) : GOTO 10`

This “POKE” program works by randomly selecting one of the one thousand positions on the screen, randomly selecting the screen code for ╱ or ╲, and placing that code in that memory location. Then, of course, it uses `GOTO 10` to loop back to the beginning and do everything again. While the steady-state output is a full screen of characters changing one at time, the program overwrites the existing contents of the screen slowly, filling in the maze pattern at random.

RANDOM SOUNDS

Finally, consider this considerably more complex program, an audio analogue of 10 PRINT. It plays a sequence of tones chosen from a distribution of two, both of which have the same timbre that approximates that of a piano. The selection is done using the same pseudorandom pattern that 10 PRINT uses, thanks to the invocation of RND(1) in line 30:

```
10 S=54272 : POKE S+24,15 : POKE S+5,190 : POKE S+6,248
20 A(0)=17 : A(1)=37 : A(2)=21 : A(3)=76
30 Q=INT(.5+RND(1)) : POKE S+1,A(Q*2) : POKE S,A(Q*2+1)
40 POKE S+4,17 : FOR T=1 TO 75 : NEXT
50 POKE S+4,16 : FOR T=1 TO 150 : NEXT
60 GOTO 30
```

In Commodore 64 BASIC, one can point into a table of PETSCII characters by simply using 205 and 206 as indices. But there is no similar built-in way to index into a table of notes. After setting up the sound chip in line 10, this program builds such a table using the array A in line 20. Furthermore, the sound chip requires two POKE commands—the ones on line 30—to change the note frequency. Although this is because the chip is extremely accurate in its pitch control, it does make for longer and more involved programs.

This book does not cover arrays (which are not part of the canonical 10 PRINT) in any detail; it would move the discussion quite far afield to explain exactly what is happening in each invocation of POKE in this program. Suffice it to say that POKE is being used to set the sound chip's registers, causing the Commodore 64 to emit musical sounds in a strightforward way—the standard way one would produce music in BASIC. The invocations of POKE are not simply storing values in memory for later use, nor are they placing values in screen memory, as in the previous example—yet all of this is necessary to move from a randomized generator of block graphics to a randomized generator of tones. This program shows how much easier it is for Commodore 64 BASIC to work on graphic, rather than musical, elements.