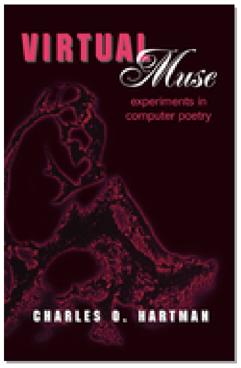
Virtual Muse

Charles O. Hartman

Published by Wesleyan University Press



>>>> / 1 /

START WITH COMPUTERS

When my son can be trusted not to put jelly in the keyboard, and when he can tell an O from a o, I'll give him a computer. I'll probably need to, just to keep his hands off mine. He can use it to play games; it may help him learn to read; and soon he'll be writing programs that draw shapes, calculate primes, graph complex functions, and perhaps compose fugues. None of this, by itself, will make him a really unusual person. It may cost me less than I hear some toys do—something between a car payment and a month's mortgage.

I'm hardly the first person to notice that this represents a striking change from the world that produced his father just a few decades back. Computers are in and of our lives in ways that thirty years ago weren't even envisioned by science fiction writers. The reality of computers then, despite their fascinating history of conceptual victories, was too depressing to inspire the leap of fantasy it would have taken to imagine where we are now.

As a high school student in a college town, I heard about an evening class in computer programming that I could attend. I was moderately happy with mathematics and decided to try it. (We used to think programming was a kind of mathematics, rather than just a product of mathematics.) This would have been about 1963. The computer language was FORTRAN IV. I have no idea what kind of computer ran our class exercise programs because I never saw it. To some juvenile computer jockeys today, this may sound like a paradox; let me explain.

These were the bad old days of "batch processing." You wrote your little program to print all the prime numbers between 1 and

100— I mean you *wrote* it out on a piece of paper. Then you went in search of a free and functioning keypunch machine. This was a gray object the size of a pygmy hippopotamus, solidly packed with gears and motors and sounding like a broken beehive or one of those automated looms that were its collateral ancestors. (The Jacquard loom, developed between 1725 and 1745, was the first machine controlled by punch cards.) There you typed your program onto "IBM cards," one line per card. Then, after doing your best to weed out all the erroneous cards, you carried your "deck" to a plexiglass window, where the clerk, a grad student training as a computer acolyte, took charge of it. He (women were very scarce around computers) noted your job account number and entered your program into an interminable queue. The next day—or the next—assuming that the computer was "up" and not, as so often, "down" (it lived in some bombproof basement in a mythified cool glass chamber where all its attendants wore white, and its health was the subject of widespread anxiety and considerable superstition), you picked up your deck of cards wrapped in a thin bundle of wide green-striped printout paper that listed in several columns of incomprehensible code all the errors the computer thought you had made in your pursuit of a project whose intention it had clearly not understood in the first place. Like Sisyphus, the fellow rolling a rock uphill in Hell, you were free to reiterate this process as many times as it took.

Oddly enough, I found the experience of computer programming attractive in many ways, though certainly not enough to pursue it as a career.

In college, at the end of the same decade, I took a distribution-requirement course about computers. The language was still FORTRAN, programs were still processed in batches, and I still didn't know much about the machine itself or how it arrived at my little printout of error messages. Batch processing continued to enforce that aweful religious separation between the computer and the profane programmer. While the attraction was still there, the frustration held steady too.

In this course we were encouraged to choose our own final projects. I was divided in soul and academic major between English and music, and I suppose I might have dithered between schemes for programs in those two areas. But in fact I don't remember contemplating even for a moment trying to deal with words. Computers didn't cope well with words at the time. The best programming language for the purpose, charmingly called SNOBOL, was a freak among its serious number-crunching relatives.

Instead, I wrote a program to harmonize chorales. My program would let me feed the computer the soprano melody of a Bach chorale, and the computer would write the alto, tenor, and bass voices to go with it. I worked out ways to encode all the rules for ensuring proper voice-leading, avoiding parallel fifths and octaves, encouraging contrary motion in the outer voices, and so on. There was even a random element, to introduce a bit of surprise into the harmonic progression. The input and output—the melody and the four-part harmonized chorale—took the form of lists of numbers that had to be translated back into notes by hand. Computers that could write (let alone read) musical notation were years and years away, even in advanced computer science labs.

I got a C+ on that project, which taught me, or should have taught me, a complex lesson. Though my program was correctly written, it wouldn't run within the tiny area of computer memory that the college had allotted for class projects. (I never knew the details of the inadequacy. Understanding them might have been beyond me anyway. Intimate knowledge of an operating system was very advanced stuff at the time.) My section teacher's evaluation, which I preserved in the file with my class notes in the vague hope of going back to the project someday, has enough exemplary historical interest to be quoted in full:

> While not being able to judge the musical quality or insight which you have demonstrated, I would say that as a computer project you have made a strategic error:
>
> Rather than aim for something which is realizeable, you have written a very complex and lengthy program, and as a result have not been able to run it and show that any of it works.

> It would have been better to start with a small, working program and build on that.
>
> Evidently you have mastered FORTRAN and I have no quarrel with your program as written.

In short, to borrow Howard Nemerov's line about some poets, I was a fine programmer—on paper.

One version of the lesson might have been that what counted in computer science was results, not beautiful fancies. Another version was that I was temperamentally unsuited to computer science. The idea of ironing out logistical details (especially in the nightmare hurry-and-wait world of batch processing) was not what attracted me to computers. The task of most computer professionals was, and still is, a management task; the talent of the best is a talent for getting things done. Design and innovation occupy a tiny percentage of the person-hours consumed by any major program. If what I wanted to do was to dream up new possibilities, I should stick to dreamy things like music and poetry, where talk is cheap and nearly sufficient.

Yet the truest form of the lesson might not have been so chastening after all. It would have sounded like pure arrogance at the time, even if I had had the temerity to formulate it: *This is no way to program computers.*

A decade or two later I could sit at my own computer, at my own desk, fiddling until my program worked. The real beginning of this approach came with the BASIC language, developed at Dartmouth specifically to teach people how to program. BASIC is an *interpreted* language. That means that you can type a statement on a keyboard, and the computer will respond directly by doing what your statement has told it to do. This is obviously good for learning. You can see your mistakes right away and correct them. But it requires that you have a computer at your immediate disposal. A batch-processed interpreted language would be an absurdity, like a car pulled by a team of goats or a game of Monopoly played by mail.

Historically, the first solution to the frozen bureaucracy surrounding computers was "time-sharing." This is a way of sharing the resources of a big computer among several people, all working at separate terminals that the central computer services in a round-robin. But time-sharing can slow down the computer's responses, and speed is the main virtue of a computer to begin with. Time-sharing just moves the bureaucracy inside the machine. The individual user's contact with the computer is still mediated by elaborate protocols (passwords, protected file systems, and so on). Ultimately, the way to implement a system of direct interaction between computers and the people who program and use them is to give every person her or his own computer.

That was the accomplishment of the late seventies and early eighties: the microcomputer revolution. Now everyone who can afford a good color TV *could* afford a small, usable computer. (This doesn't mean that everyone who's in a pinch will make that choice; but in 1995, sales of computers beat sales of TVs for the first time.) With the advent of the affordable desktop micro, the computer became exactly what I had thought it was until I got my final grade in the college course: a dreaming machine.

Interpreted languages like BASIC have some built-in limitations. They're often contrasted with *compiled* languages like FORTRAN and Pascal and C. In these, a special program called a compiler translates the code you write (which looks at least *somewhat* like a human language) into a code the machine can read. In the eighties very fast, cheap, powerful compilers came along and made this distinction less important. The feedback for the experimenter is pretty immediate either way. But in any case, all these developments have aimed in the same direction: toward placing as much computing power as possible in the hands of individual programmers, not reserving it for regimented initiates.

It's true that good computer programs that do complex and important tasks aren't usually written this way. They're written in many small pieces by large teams of highly trained, intelligent, cooperative people after a long period of planning. They aren't so much written as *conducted*, like military campaigns, and their success depends on the same virtues—plenty of resources (money, equipment, people), extreme care for details, and just the right bit of audacity.

But all over the world a growing number of people are *playing* with computers. And the most interesting play isn't like card or board games but like Erector sets.

If I were writing that chorale harmonization program now and put all my best fiddling into it, and it still wouldn't work, I would know I was up against one of three things: (a) a bit of fundamental ignorance about computers; (b) a bit of fundamental ignorance about the harmonization of chorales; or (c) a fundamental incompatibility between the computer and this project.

The first two problems would be interesting and informative. If I had the time to continue, I'd sit down and learn enough to rectify them. The third problem might be still more interesting. It would mean that I'd encountered something computers simply can't do. This third possibility brings me back to questions I was trying to explore in that college course.

I think now that I undertook my computer chorale project because I was fascinated by a certain kind of boundary. The year before, I had completed the rigorous introductory course for music majors. We had learned to harmonize chorales, even on sight (the final exam was terrifying). This is a peculiar skill and by no means a simple one, but its relation to musical composition was not very clear.

"Musical composition" was the name of that sphere of activity in which J. S. Bach excelled. Along the way toward that lofty sphere the ardent student spent a year learning the rules of harmony and voice-leading. It was the idea of a *gradus ad Parnassum*, an apprenticeship climb up the steep hill of the Muses. To anyone who had practiced and practiced playing an instrument, this concept of progressing from the technical elements to the real music seemed quite plausible. Yet there was also an obvious gap between exercises and music. It's not unlike the gap between mindless neurons and the mind they make up.

I do *not* mean that Bach's music was inexpressibly sublime and the rules were mere mechanical drudgery. I had learned enough in high school from my composer friend Charles Wolterink to see that the beauty of Bach's music was inseparable from the complex beauty of the tonal system he used (and helped perfect). The grandeur of many passages in his music depended on a sense of inevitability in the way the system worked out its own consequences. It's often compared to the grand beauty of mathematics, though Newtonian physics might supply a clearer analogy. The musical "rules" were the map or skeleton of that same system. They weren't petty restrictions, like parking laws, but possibilities of motion, like the rules of chess. Furthermore, the rules themselves, when you set them to work on a bit of musical material, could often produce some beautiful effects, as if automatically.

Bach, the master, usually abided by the same harmonic rules my teachers made me obey. So when it occurred to me that a computer could do what I had spent a year learning to do, that wasn't a reason to feel foolish or under mechanical attack. Instead, I became interested in formulating one part of musical knowledge by automating it. In part, it was a way of confirming what I knew and bringing it all into consciousness.

How was it that anyone could set out to compose impossibly beautiful music? First, learn the rules of harmony? Fine. Then counterpoint, then orchestration? Fine. If it could be done, it could be done in stages. But if it could be done in stages, and if these could be broken down and specified exactly enough, how far through the stages could a computer be taught to follow?

Sometimes Bach broke the rules. My favorite example was a passage near the end of the fifth fugue from the second book of the *Well-Tempered Clavier*—two measures in which the harmony suddenly goes completely berserk, unpredictable, barely analyzable. That wild moment couldn't be programmed, I was sure. But the beauty of Bach's music didn't depend on those rule-breaking moments. Rather, it had to do with some balance between rules and rulelessness. (It wasn't random, either. The boundaries among the random, the arbitrary, and the unpredictable will concern us later on.)

Though I didn't know it at the time, I was investigating something analogous to the question of "computability." Some problems (in mathematics, for instance) are computable, and others are not. To some questions, the answers can be determined in a finite amount of time; to others, not. (Marvin Minsky has written a hard but readable book on the theory of Finite Automata that treats this distinction in detail.) Researchers in artificial intelligence (AI, for short) are constantly wrestling with the question of whether aspects of human behavior ("intelligence" for short?) are computable or not. In a minor way, without much rigorous philosophical comprehension, I was puzzling over a similar kind of question. For me it was a matter of what was knowable about the process of musical composition; trying to compute it was a way of finding that out.

My main purpose in writing the chorale program wasn't to produce chorales, to save myself the labor of harmonizing them. The computer was a way of exploring the nature of what I knew about music. Later, when I began to use the computer in poetry, it would participate in a similar process of thought. Yet it would also turn out to *change* the poetry I would write, with or without a computer.

In the end I studied English; music became my avocation, not my profession. I wrote poems and published them. I studied poetry and wrote critical articles and books. I became a teacher of literature and of the writing of poetry. For a while, when I briefly tired of teaching, I worked as a technical writer for computer companies. Writing computer manuals is (at its most interesting, which is rare) an act of cultural translation: What your reader needs is to be initiated into a new way of thinking; your job is to lead the way across that border.

Poets tend to read widely, and as compared with other teachers we assign reading that sends students indiscriminately to all floors of the campus library. Literary critics who are interested in music or science or computers tend to wander away from traditionally defined regions of scholarly labor. Technical writers are amphibians between still different worlds. So much of what I did, during the years after college and graduate school, kept me thinking about the boundaries between different systems of knowledge, different specializations of mental activity, different ways of making things work.

My recent programming experiments have grown out of all this boundary crossing. Before I describe the experiments themselves, I need to skip across the biggest gap in the neighborhood. Let me try to describe some of what poetry looks like, at least to some poets, these days.