SEMICONDUCTING – MAKING MUSIC AFTER THE TRANSISTOR[1]

Nicolas Collins

Department of Sound at the School of the Art Institute of Chicago
ncollins@saic.edu

INTRODUCTION

At some point in the late 1980s the composer Ron Kuivila proclaimed, "we have to make computer music that sounds like electronic music." This might appear a mere semantic distinction. At that time the average listener would dismiss any music produced with electronic technology – be it a Moog or Macintosh – as "boops and beeps." But Kuivila presciently drew attention to a looming fork in the musical road: "boops and beeps" were splitting into "boops" and "bits". Over the coming decades, as the computer evolved into an unimaginably powerful and versatile musical tool, this distinction would exert a subtle but significant influence on music.

Ron and I had met some 15 years earlier, in 1973 at Wesleyan University, where we both were undergraduates studying with Alvin Lucier. Under the guidance of mentors like David Tudor and David Behrman like a number of American composers of our generation, we began the 1970s building circuits, and finished out the decade programming pre-Apple microcomputers like the Kim 1. The music that emerged from our shambolic arrays of unreliable homemade circuits fit well into the experimental aesthetic that pervaded the times (the fact that we were bad engineers probably made our music better by the standards of our community), but we saw great potential in those crude early personal computers, and many of us welcomed the chance to hang up the soldering iron and start programming[2].

The Ataris, Amigas and Apples that we adopted in the course of the 1980s were vastly easier to program on than our first machines, but they still lacked the speed and processor power needed to generate complex sound directly. Most of the "Computer Music Composers" of the day hitched their machines to MIDI synthesizers, but even the vaunted Yamaha DX7 was no match for the irrational weirdness of a table strewn with Tudor's idiosyncratic circuits. The bottleneck lay in MIDI's crudely quantized data format, optimized for triggering equal-tempered notes, and ill suited for complex, continuous changes in sound textures. My personal solution was to flit back and forth between software and hardware as needed to find the right balance between control and splendor; Ron's was to become a much better programmer than I ever could be.

By 2011 MIDI no longer stands between code and sound: any laptop has the power to generate directly a reasonable simulation of almost any electronic sound – or at least play back a sample of it. But I'm not sure that Ron's goal has been met. I can still hear a difference

---

1 Presented at the "Technology and Aesthetics" Symposium - NOTAM, Oslo, May 2011.

2 Although this potential was clear to our small band of binary pioneers, the notion was so inconceivable to the early developers of personal computers that Apple trademarked its name with the specific limitation that its machines would never be used for musical applications, lest it infringe on the Beatles' semi- dormant company of the same name – a decision that would lead to extended litigation after the introduction of the iPod and iTunes. This despite the fact that the very first non-diagnostic software written and demonstrated at the Homebrew Computer Club in Menlo Park, CA in 1975 was a music program by Steve Dompier (http://www.convivialtools.net/index.php?title=Homebrew_Computer_Club) – an event attended by a young Steve Jobs.

between hardware and software. Why?

Most music today that employs any kind of electronic technology depends on a combination of hardware and software resources. Although crafted and/or recorded in code, digital music reaches our ears through a chain of chips, mechanical devices, speakers and earphones. "Circuit Benders", who open and modify electronic toys in pursuit of new sounds, often espouse a distinctly anti- computer aesthetic, but the vast majority of the toys they hack in fact consist of embedded microcontrollers playing back audio samples – one gizmo is distinguished from another not by its visible hardware but by the program hidden in ROM. Still, whereas a strict hardware/software dialectic can't hold water for very long, arrays of semiconductors and lines of code are imbued with various distinctive traits that combine to determine the essential "hardware-ness" or "software-ness" of any particular chunk of modern technology. Some of these traits are reflected directly in <u>sound</u>; others influence how we <u>compose</u> with certain technology, or how we interact with it in <u>performance</u>. Many of these traits can be ignored or repressed to some degree – just like a short person can devote himself to basketball – but they nonetheless affect the likelihood of one choosing a particular device for a specific application, and certainly exert an influence on the resulting music.

I want to talk a little about distinctive differences between hardware and software tools as applied to music composition and performance. My observations are based on several decades of personal experience: in my own activity as a composer and performer, and in the music of my mentors and peers, as observed and discussed with them since my student days. I am not an authority in this area in any academic sense. I confess that I have done precious little hard research, and freely acknowledge that my perspective comes from an extreme fringe of musical culture.

Before starting I'd like to qualify some of the terms I'll be using. When I speak of "hardware" I mean not only electronic circuitry, but mechanical and electromechanical devices as well – from traditional acoustic instruments to electric guitars. By "software", however, I'm limiting myself pretty much to computer code as we know it today, whether running on a personal computer or embedded in a dedicated microcontroller or DSP. I use the words "infinite" and "random" not in their scientific sense, but rather as one might in casual conversation, to mean "a hell of a lot" (the former) and "really unpredictable" (the latter).

THE TRAITS

Here then are what I see as the most significant features distinguishing software from hardware:

- Traditional acoustic instruments are <u>three-dimensional</u> objects, radiating sound in every direction, filling the volume of architectural space like syrup spreading over a waffle. Electronic circuits are much flatter, essentially <u>two-dimensional</u>. Software is linear, every program a <u>one-dimensional</u> string of code. In an outtake from his 1976 interview with Robert Ashley for Ashley's *Music With Roots in the Aether*, Alvin Lucier justified his lack of interest in the hardware of electronic music with the statement, "sound is three-dimensional, but circuits are flat." At the time Lucier was deeply engaged with sound's behavior in acoustic space, and he regarded the flatness of circuitry as a fundamental weakness in the work of composers in the thrall of homemade circuitry. A circuit may never be able to embody the topographic richness of standing waves in a room, but at least a two-dimensional array of electronic components on a fiberglass board allows for simultaneous, parallel activity of multiple strands of electron flow, and its resulting sounds. In software all action is sequential, all sounds queue up through a linear pipeline. With sufficient processor speed and the right programming environment one can create the impression of simultaneity, but this is an illusion -- much like a Bach flute

sonata weaving a monophonic line of melody into contrapuntal chords. This limitation in software was a major factor motivating Kuivila to develop, with David Anderson, the programming language "Formula", whose strength lay in its accurate control of the timing and synchronization of parallel musical events – getting linear code a little closer to planar. Given the ludicrous speed of modern computers this distinction might seem academic -- modern software does an excellent job of simulating simultaneity -- but writing code nonetheless nudges the programmer in the direction of <u>sequential thinking</u>, and that this can affect the overall character of work produced in software.

- Hardware occupies the <u>physical world</u> and is appropriately constrained in its behavior by various natural and universal mechanical and electrical laws and limits. Software is <u>ethereal</u> – its constraints are artificial, different for every language, the result of linguistic design rather than pre-existing physical laws. When selecting a potentiometer for inclusion in a circuit one has a finite number of options in terms of maximum resistance, curve of resistive change (i.e., linear or logarithmic), number of degrees of rotation, length of its slider, etc. When implementing a potentiometer in software all these parameters are infinitely variable. Hardware has edges; software is a *tabula rasa* wrapped around a torus.

- As a result of its physicality, hardware – especially mechanical devices – often displays <u>non-linear adjacencies</u> similar to state-changes in the natural world (think of the transition of water to ice or vapor). Pick a note on a guitar and then slowly raise your fretting finger until the smooth decay is abruptly choked off by a burst of enharmonic buzzing as the string clatters against the fret. In the physical domain of the guitar these two sounds – the familiar plucked string and its noisy dying skronk – are immediately adjacent to one another, separated by the slightest movement of a finger. Either sound can be simulated in software, but each requires a wholly different block of code – no single variable in the Karplus-Strong "plucked string algorithm" can be nudged by a single bit to produce a similar death rattle; this kind of adjacency must be programmed at a higher level, and does not typically exist in the natural order of a programming language. Generally speaking, adjacency in software rld of hardware abounds with this kind of abrupt transition. A break-point in a hardware instrument – fret buzz on a guitar, the unpredictable squeal of a the STEIM Cracklebox – can be painstakingly avoided or joyously exploited, but is always lurking in the background, a risk, an essential property of the instrument.

- Software is <u>Boolean</u> through and through: it either works correctly or fails catastrophically; and when corrupted code crashes the result is usually silence. Hardware performs along on a continuum that stretches from the optimum behavior intended by its designers to irreversible, smoky failure; circuitry – especially analog circuitry – usually produces sound even as it veers toward catastrophic breakdown. Overdriving an amplifier to produce guitar distortion; feeding back between a microphone and a speaker to play a room's resonant frequencies; "starving" the power supply voltage in an electronic toy to produce erratic behavior – these "misuses" of circuitry generate sonic artifacts that can be analyzed and modeled in software, but the quasi-destructive processes themselves (saturation, feedback, under-voltage) are very difficult to transfer intact from the domain of hardware to that of software while preserving functionality in the code. (This comparison echoes the familiar "digital vs. analog" distinction, but I prefer to focus on the difference between software and hardware because even digital hardware can be made to sing outside its "normal" mode of operation.)

- Software is <u>deterministic</u>, while all hardware is <u>indeterminate</u> to some degree. Once debugged, code runs the same every time. Hardware is notoriously unrepeatable: consider recreating a patch on an analog synthesizer, restoring mixdown settings on a pre-automation mixer, or tuning a guitar. The British computer scientist John Bowers

once observed that he had never managed write a "random" computer program that would run, but that he could make "random" component substitutions and connections in a circuit with a high certainty of a sonic outcome (a classic technique of Circuit Bending).

• Hardware is <u>unique</u>, software is a <u>multiple</u>. Hardware is constrained in its "thinginess" by number: whether hand-crafted or mass-produced, each iteration of a hardware device requires a measurable investment of time and materials. Software's lack of physical constraints gives it tremendous powers of duplication and dissemination. Lines of code can be cloned with a simple cmd-C/cmd-V: building 76 oscillators into a software instrument takes barely more time than one, and no more resources beyond the computer platform and development software needed for the first. There is no distinction in software between an "original" and a "copy": MP3 audio files, PDFs of scores, and runtime versions of music programs can be downloaded and shared thousands of times without any deterioration or loss of the matrix – any copy is as good as the master. If a piano is a typical example of traditional musical hardware, "pre-digital software" would lie somewhere between a printed score (easily and accurately reproduced and distributed, but at a quantifiable – if modest – unit cost) and the folk song (freely shared by oral tradition, but more likely to be transformed in its transmission.) Way too many words have already been written on the significance of this trait of software – of its impact on the character and profitability of publishing as it was understood before the advent of the World Wide Web; I will simply point out that if all information wants to be free, that freedom has been attained by software, but is still beyond the reach of hardware.

• Software accepts infinite undos, is eminently <u>tweakable</u>. But once the solder cools, hardware <u>resists change</u>. I have long maintained that the young circuit-building composers of the 1970s switched to programming by the end of that decade because, for all the headaches induced by writing lines of machine language on tiny keypads, it was still easier to debug code than de-solder chips. Software invites endless updates, where hardware begs you to close the box and never open it again. Software is good for <u>composing</u> and <u>editing</u>, for keeping things in a state of flux; hardware is good for making reasonably stable, <u>playable instruments</u> that you can return to with a sense of familiarity (even if they have to be tuned). The natural outcome of software's malleability has been the extension of the programming process from the "behind the scenes", pre-concert preparation of a composition to a part of the actual performance -- as witnessed in the rise of "live coding" culture (as practiced by devotees of SuperCollider and Chuck, for example). Live circuit building has been a fringe activity at best: Tudor finishing circuits in the pit while Cunningham danced overhead; Live Objects soldering PICs on top of an overhead projector; live coding vs. live circuit building in annual competition between the younger Nick Collins (UK) and me for the *Nic(k) Collins Cup*[3].

• On the other hand, once a program is burned into ROM and its source code is no longer accessible, software flips into an <u>inviolable</u> state. At which point re-soldering, for all it unpleasantness, remains the only option for effecting <u>change</u>. Circuit Benders hack digital toys not by disassembling and re-assembling the code (typically sealed under a malevolent beauty-mark of black epoxy) but by messing about with traces and components on the circuit board. A hardware hack is always lurking as a last resort, like a shim bar when you lock your keys in the car.

• Thanks to memory, software can work with <u>time</u>. As several writers have pointed out, the transition from analog circuitry to programmable microcomputers gave composers a new tool that combined characteristics of instrument, score and performer: memory allows software to play back prerecorded sounds (an instrument), script a sequence of

---

3   See http://www.nicolascollins.com/collinscup.htm

events in time (a score), and make decisions built on past experience (a performer.) Before user-programmable computers, electronic circuitry was used primarily in an instrumental capacity – to produce sounds immediately[4].　It took software-driven microcomputers to fuse this trio of resources into a new paradigm for music creation.

- Given the sheer speed of modern personal computers and software's quasi-infinite power of duplication (see above), software has a distinct edge over hardware in the density of musical texture it can produce: a circuit is to code as a solo flute is to the full orchestra.　But at extremes of its behavior hardware can exhibit a degree of complexity still beyond the power of software to simulate effectively: the burst of breath at the attack of a note on that flute or the initial tug of rosined bow hair on the string of a violin; the unstable squeal of wet fingers on a radio's circuit board; the supply voltage collapsing in a cheap electronic keyboard. Hardware still does a better job of giving voice to the irrational, the chaotic, the unstable.

- Software is imbued with an ineffable sense of now – it is the technology of the present, and we are forever downloading and updating to keep it current.　Hardware is yesterday, the tools that were supplanted by software.　Vinyl records, patchcord synthesizers, and tape recorders have been replaced by MP3s, samplers, and ProTools.　In the ears and minds of most users this is an improvement – software does the job "better" than its hardware antecedents.　Before its replacement by a superior device, qualities of a tool that don't serve its main purpose can be seen as weaknesses, defects, or failures: ticks and pops, oscillators drifting out of tune, tape saturation and distortion. But when a technology is no longer relied upon for its original purpose, these same qualities can become interesting in and of themselves. The return to "outmoded" hardware is not always a question of nostalgia, but often an indication that the scales have dropped off our ears – like being pleasantly surprised by the beauty of a co-worker encountered outside the office.

CONCLUSION

I came of age as a musician during the golden age of the "composer-performer": the Sonic Arts Union, David Tudor, Terry Riley, LaMonte Young, Pauline Oliveros, Steve Reich, Philip Glass.　Sometimes this dual role was a matter of simple expediency (established orchestras and ensembles wouldn't touch the music of these young mavericks at the time), but more often it was a question of direct, personal control that led to this flowering of composer-led ensembles that resembled rock bands more than orchestras.　Fifty years on, the computer – with its above-mentioned power to fuse three principle components of music production – has emerged as the natural tool for this mode of working.

But another factor propelling composers to the performance stage was the spirit of improvisation.　The generation of artists listed above may have been trained in a rigorous classical tradition, but by the late 1960s it was no longer possible to ignore the musical world beyond the gates of academe or the banks of the river of European high-art music. What was then known as "World Music" was reaching our ears through a trickle of records and concerts. Progressive Jazz was in full flower.　Pop was inescapable.　And composers of my age – the next generation -- had no need to reject an older tradition to strike out in a new direction: Indian music, Mile Davis, the Beatles, John Cage, Charles Ives and Monteverdi were all laid out in

---

4　A handful of artist-engineers designed circuits that embodied some degree of performer-like decision-making: Gordon Mumma's "Cybersonic Consoles" (1960s-70s), which as far as I can figure out were some kind of analog computers; my own multi-player instruments built from CMOS logic chips in emulation of Christian Wolff's "co-ordination" notation (1978).　The final stages of development of David Behrman's "Homemade Synthesizer" included a primitive sequencer that varied pre-scored chord sequences in response to pitches played by a cellist ("Cello With Melody Driven Electronics", 1975) presaging Behrman's subsequent interactive work with computers.　And digital delays begat a whole school of post-Riley canonical performance.

front of us like a buffet, and we could heap our plates with whatever pleased us, regardless of how odd the juxtapositions might have tasted to anyone else. Improvisation was an essential property of many of these dishes, and we sought technology that expanded the horizons of improvisation and performance just as we experimented with new techniques for composition.

It is in the area of performance where I feel that hardware – with its tactile, sometimes unruly properties I described above -- still holds the upper hand. I regard this as a testimony not to any failure of software to make good on a perceived promise to make everything better in our lives, but rather as a pragmatic affirmation of the sometimes messy but inarguably fascinating irrationality of human beings.