Proceedings of the Electroacoustic Music Studies Network Conference
Electroacoustic Music Beyond Performance, Berlin, June 2014

www.ems-network.org

The Deadly Embrace Between Music Software and Its Users

Miller Puckette

University of California, San Diego

msp@ucsd.edu

Abstract

One of the many differences between software instruments and physical ones is that software instruments, since they deal in information instead of physical vibrations, can operate in ways that are indirect to the point of being mysterious. Software in the past half century has rapidly increased in complexity and decreased in stability. This raises problems for both designers and users of software intended for musical creation. Specific pieces of computer-mediated music can easily become impossible to perform within a decade of their creation. More broadly, musical practices can easily become embedded in specific software configurations and hard to study from vantage points other than the creator's chair. What one musician can learn from another one can be limited by incompatible differences between the software paradigms they favor. Although the software designer strives to make software as open and transparent as possible, this transparency is always limited by competing exigencies for efficiency, "power", and sometimes a commercially imposed need for secrecy. While music software can give the musician great power, its users should stay aware of the risks that can accompany its great and largely hidden complexity.

Introduction

It's an honor to have been invited to speak in front of such a distinguished gathering of musicians and scholars, perhaps all the more in light of my lack of formal training in either music or musicology. I speak not so much as one of you than as one of the objects you're looking at, a worker in electronic music, a laboratory animal under study. It's as if Pavlov's dog had acquired speech and were able to give Pavlov a dog's perspective on their experiment. If this happened I think Pavlov would have listened carefully to the dog, but of course he would not have listened as to a colleague, and would have decided for himself how the dog's propos should be interpreted, given the dog's lack of formal training in psychology.

The focus of this conference, the study of electronically mediated music, should properly rely on methodologies and rise to standards of scholarship that are carefully and cumulatively arrived at as this new field develops. It's a time of beginnings, in which the methodologies themselves are young and changing. The classical information-gathering techniques of musicology weren't designed for loudspeaker music, and people in this room are developing new ones right now.

One such technique could be to study the tools of the new musician's trade. Today these tools are primarily in the form of software. We have just lived through a period of transition from hardware to software tools, starting perhaps in 1957 with Max Mathew's MUSIC program and finishing (mostly) about ten years ago when tape recorders finally gave way to soundfiles.

To be more precise, it's not that we've quit using hardware. But a modern studio needs little more than microphones, loudspeakers, and a computer with an audio interface. These things are different from hardware-age hardware in that they are more transparent. The job of a microphone is to translate the air pressure at a point in space transparently to a voltage; and similar transparency is desired in the other conversion stages (interface and loudspeaker). The computer can also be described as transparent (ideally at least) in that it applies the musician's instructions (inputs to the computer), via whatever software is in use, as correctly and directly as possible to the sound or other musical object being manipulated on the computer.

My own role in this transition, as a developer of music software, has put me on one side of the continuing conversation between developers and musicians as the age of software-mediated music has set in. In these conversations both sides have to learn about each other's abilities and needs, and solutions have to be found that best navigate the tradeoffs that arise between competing requirements.

Here I'm mostly drawing on my own first-hand knowledge and experience developing real-time interactive music software, although I think some of my thoughts will apply to music software in general. I was there during the transition from hardware to software tools in live electronic music performance. At about the same time I was learning to code computer music applications (say 1980), Giuseppe Di Giugno was building a series of digital devices culminating in IRCAM's 4X, which was a milestone in the transition from analog to real-time digital computing. I started writing a program I called Max, for controlling the 4X in live musical performances, in 1985. About 1989-1992 I was part of Eric Lindemann's design team for the ISPW, IRCAM's successor to the 4X. The ISPW was the first practical real-time music number cruncher that could be programmed portably. In 1993-1994 I helped port the ISPW version of Max to a general-purpose desktop machine, completing its long journey from a 4X-specific tool to a purely software solution. This progression was approximately in sync with the overall movement of the field of computer music in that period, so my experiences might be representative of those of software developers during the transition from hardware to software.

Consequences of the digitization of electronic music production

Computers as musical tools have major advantages over the sound-generating and recording equipment of the hardware era. Analog gear (oscillators, for example) can only have limited accuracy, but computer oscillators can be made as accurate as desired. Further, operations may be designed to be deterministic, so that a same set of instructions, tried on different days, would have the same result; this is in contrast with analog equipment with which it can be impossible to recreate something one has done the day before.

A third advantage is the permanence of digital media. Not only sounds (both working materials and finished pieces), but also the scores, patches, and the software programs themselves, can be archived one day and retrieved in identical form twenty or a hundred years later.

Compared with hardware, software has at least these two serious disadvantages: first, you can't put your hands directly on the materials you're working with (all actions are mediated through a mouse-and-keyboard interface that is better suited to banking transactions than to making musical gestures). Second, computer documents quickly become obsolete as the supporting software evolves. Of these two, the first is of limited import since it only hinders us at the time of creation; but the second issue, obsolescence, strikes silently when the creator isn't looking. It's a potentially fatal syndrome with no early symptoms.

The problem of document obsolescence is well known to computer programmers and a good programmer and/or a savvy computer user will work hard to understand its causes and to prevent or ameliorate it wherever possible. In my opinion, the problem arises mostly because a document never truly depends only on one piece of software but on a network of interdependent modules: one or more software environments, connected by various protocols, supported by libraries and an operating system often augmented by third-party extensions and drivers. And it isn't even that simple, because other projects realized on the same computer, even stuff as innocuous as electronic mail, also depends on some of the same software modules as your music project. Your web browser might auto-update itself some day, in the process updating some shared library or configuration file, and behold: the patchwork of software that was enabling your music project (while remaining politely out of sight) is suddenly broken in some way.

Armies of programmers and software engineers are put to work updating software to keep it from falling out of compatibility with evolving computer systems. The evolution that pushes software into obsolescence today mostly lies in other software, which itself is evolving for the same reason. It's a chain reaction, and it keeps many programmers gainfully employed.

There is, of course, another source of pressure for software to evolve: the constantly changing demands of users. An added feature might then require a more recent version of some software library, and another chain reaction of updates has begun.

Documents, programs, and software environments

The discussion above tacitly assumes that computer data is neatly separated into software programs and documents, in which the programs are used to create, edit, process, and/or archive the documents. On more careful examination, this assumption turns out to be oversimplified. A software program is itself a document, and its source code is a collection of other documents. Another program, a compiler, builds programs from sources. Documents used in the electronic mediation of music are not necessarily passive heaps of data: they can themselves function as programs (a Pd patch, for example). One characterization of the difference between fixed-media music and live electronic music could be that the former is mediated by static documents and the latter by documents that function as computer programs that realize the music.

Software today seems mostly to consist of environments (such as Pd or Sibelius or Microsoft Excel) and plug-ins (such as Freeverb). The environments are stand-alone programs that are controlled interactively by the user, usually in order to edit documents. The plug-ins are usually smaller, simpler programs that are loaded into some environment to fill a specific task. An environment might "host" several or many plug-ins simultaneously. One view of my original Max program is that it was an early plug-in-based architecture, in which Max proper was an interconnection mechanism and scheduler for the plug-ins (the objects making up a patch).

A working Max or Pd document, one for instance that realizes a piece of live electronic music, depends not only on Max or Pd in itself, but also on all those plug-ins (called "externs") that aren't supplied internally as part of Pd.

Incidentally, if someone today wanted to write software of use in making real-time electronic music, should they try to make an environment or a plug-in? My provisional answer to that is that the time for inventing real-time computer music generating environments might be at an end. There is a sort of ecological niche for real-time computer music generating environments, and at the moment at least it looks quite full. In particular, on the experimental end of the spectrum (which is the end that particularly interests me) it appears that the only reasonable thing to propose as a working environment is some kind of programming environment (ether graphical or textual) – but Max/MSP and Pd and Supercollider seem to cover everyone's needs at least adequately. Even if a much better environment came along it's hard now to imagine it supplanting one of those widely-used environments, in the same way as it seems unlikely we'll ever see the qwerty-style keyboard replaced. I think much the same thing is true in most existing sectors of the computer music software world. Young programmers might be wisest to try their hand at plug-ins instead.

The life cycle of software programs

Although it's an oversimplification, I'll assume that the overall software collection living on a musician's machine can be separated into independently developed programs. Each software program is written and maintained autonomously, perhaps by a single developer, or perhaps by a large team whose membership might change over time. The program might be written to a more or less pre-determined and written set of specifications, or alternatively it might be developed in a more experimental way, allowing the adoption of whatever new ideas come up in the middle of the process.

Some programs are designed to do generically defined tasks and/or to be clones of other programs. These programs can often be replaced in the future with other ones created to do the same task better. Examples might include web browsers or C compilers. Both of these are often replaced by other, compatible ones in the lifetime of a musician.

Pd and Max are examples of the opposite: their file formats are unique to them, and their functionalities need obey no standards. Furthermore, because both were written in the experimental mode described above, an exact description of what either program does (which one would need to write and verify new programs to replace them) is hard or impossible to come by. The only complete description of the functionality of Pd is its own source code.

For this reason, the longevity of Max and Pd is an important question: patches written in either environment will only remain usable as long as that environment can be kept in working order. I would rate longevity as one of the three most important criteria in my development of Pd (the others are that new versions of Pd should run older patches compatibly, and that Pd should never stop working unexpectedly during a live performance).

The prospects for longevity of a program may depend in part on conditions that are beyond the developer's control. In most private enterprises the code belongs to the developer's employer, and may be bought and sold, so that the developer may be left in a position of not being able to continue to maintain the software. Even if a private company retains copyright, it may simply stop updating the program. In this case nobody else is likely to be able to develop the program further. And even if the company continues to develop the program it might be changed incompatibly for commercial reasons.

In my experience, the life cycle of a piece of software, if it isn't terminated or corrupted for any of the above reasons, is as follows. It starts out small and relatively error-free. As development continues, features are added and errors corrected; each of these usually increases the size and complexity of the code, so the program grows larger and harder to maintain. Sometimes the developer decides to restructure the code to make it simpler and increase the reuse of code (this is called "refactoring" in coders' jargon). Although most coders claim that this tends to reduce the size and complexity of a program, in my experience it has always had the opposite effect, making the program yet larger and more complicated as the developers see opportunities to increase generality and power. At some point, the program becomes so large and complex that it becomes difficult to maintain: the software errors multiply, fixing them becomes more complicated and difficult, and increasingly, errors are introduced in the process of fixing other ones. Finally the program becomes so unreliable that its users start to defect, and at some point, with the user base shrinking and the developer becoming interested in other things, the program ceases to be worthwhile to maintain and it is left to die.

The faster the developer writes code, the faster the program will reach the point of unmaintainability. Developers of commercial programs are often under great pressure to add all sorts of features to make programs more salable; those who do not go to extraordinary lengths to resist this pressure will thereby hasten the program's demise.

The most recent program I maintained before Pd was Max/FTS at IRCAM, whose code base grew from about 1987 to 1994, at which point it had about 100,000 lines of code. Although development of Max/FTS was stopped for other reasons (IRCAM decided to re-implement the whole thing from scratch and assigned the task to other software developers), it is only fair to say that at 100,000 lines of code, Max/FTS was approaching the limit of maintainability. If we count all the C code and headers and the TCL/TK GUI code that makes up Pd, it now totals 78115 lines. Seeing this, I have drastically slowed down the growth of Pd in the last five years or so. I'm aiming for a usable lifetime of 50 years, of which Pd has now seen about 1/3. It looks like the 50-year life span might actually work out, but only if I avoid adding any but the most carefully justified and planned extensions now.

Hiding complexity

A mantra of software design is to hide complexity. If our desire is to make a program as complex as possible, we will be more successful if this complexity is kept out of sight. The process of hiding complex code behind simpler interfaces is called "encapsulation" in coders' jargon. The idea is that a program may be made more complex if the complexity is well divided up into smaller portions, all hidden away to keep those portions from mixing and making a larger tangle.

Arguably, though, an even more powerful strategy for managing complexity is simply to avoid it altogether, that is, try to think of simple things a program could do instead of doing complicated ones. This is because, if to be able to do complicated stuff we have to hide the complexity, then the user probably won't understand what the program is really doing. In many fields it's actually good to keep the user from fretting about the complex details that the software is taking care of. But perhaps not so much in live electronic music production. Why would you not want to know how your instrument works? If it's doing stuff you don't understand or even know about, are you really in control of the music you are making?

As a program becomes more and more complex over its life cycle, it is desirable that specific projects using the software not have to always take advantage of the full complexity of the program. In the ecosystem of computer music, there might be a huge range of possible ways to use a given program, each of which only exercises some reasonably-sized part of it. Patching environments are good for this reason: you can get away with only using the Pd objects you need for a particular piece of music; a reasonably scoped artwork in general won't use every single thing in the tool shed.

Alas, it seems that, in practice, composers and other artists always exceed every limit that is imposed by the software and hardware, in terms not only of computing power but also of complexity. The tendency is to keep adding stuff until something simply ceases to work (in an amusing microcosm of the software development life cycle described above). The software developer is always urging the artist to reduce complexity; the artist is always pushing to the limit.

Complexity is not the same thing as expressive power. One wants one's software to have the greatest expressive power possible, in the simplest possible way. It is hard to figure out how to endow a program with expressive power without excessive complexity. I hope Pd is a good example of this, but I can't explain to others very effectively what they should do to give their own programs similar qualities.

Communication between artists and developers

It is possible to imagine a software developer writing a program in isolation from the loci in which the program is used, relying on artists' statements about what they need. However, much more can be learned much faster if the software developer becomes personally involved in at least some projects in which artists use the software. (There might seem to be a chicken-and-egg problem here, since the software presumably doesn't exist during the time period in which it is being developed; but this can always be finessed in practice.) It is not only more effective to work this way, but also more fun. It's also salutary for the software developer to feel the fear and stress of opening night---never more so than when there is a live performance at stake.

When a software developer and an artist discuss working together on a project, neither one of the two is likely to understand more than a small slice of the other's situation and concerns. Most artwork takes a great deal more work than just the technical realization that the developer sees. And software designers have to deal with many technical considerations (including some that I've described here) that often aren't clear to the artist.

The artist can often describe specific things he or she would like to do. Can we just write software that does that? Usually not, because, first, we're hoping this isn't the only artist that will use the work; so we should aim to make whatever we do enable the specific aims of the artistic project, but also the specific aims of any other artist wishing to do something different with some similar technique. The artist might ask for a piano sound and the developer respond with a sampler. The developer somehow filled the artist's needs by making something different from what was asked for.

This is an almost universal quality of objects I've designed for Max and Pd. Nobody asked for objects to do arithmetic, for example; it was just clear to me that that was going to be needed in productions using Max, starting from the very first one. The process by which a software developer translates the artist's spoken and unspoken needs into software solutions is one more thing I don't know how to teach.

Pd's design choices

Pd, and many of its plug-ins, are written in the C programming language. In the same way as a Pd patch is a program written in Pd, Pd is a program written in C. Why, then, can't you just skip a step of indirection and realize your computer music piece directly in c? Possibly because Pd takes care of some things so that the user won't have to (for instance, by supplying a sinusoidal oscillator). If you wrote your piece in C you might have to write an oscillator, but in Pd you simply invoke the pre-supplied one. Pd supplies many such plug-ins, called "objects".

But frequently it happens that a tool isn't so easy to define universally. A band-pass filter, for example, can be designed in many different ways depending on the user's needs (do center frequency and Q need to change quickly? How steeply should the frequency response drop away from the pass band? And so on.) There is a trade-off: the more an environment offers to take low-level tasks out of the care of the user, the more the program has invisibly specified.

One can mitigate this difficulty simply by supplying an enormous number of choices for objects for which there's no one clear canonical design. Unfortunately, this would greatly increase the size of the application and therefore reduce its likely useful lifetime. Alternatively, in a plug-in architecture one can simply leave the void for others to fill. But then, Pd patches that use objects (plug-ins) from outside Pd will depend not only on the continued, compatible availability of Pd but also on the objects in the same way, so the fragility of the patch (its inability to withstand the ravages of the upgrade cycle) will be worsened. The design of the operations (such as oscillators or filters) in Pd strives for transparency but inevitably affects what the user will get out of a Pd patch.

A deeper and more central issue is raised in the way Pd interconnects and schedules the objects in a patch. This was the outcome of a long process of careful design, aiming always at transparency (a term especially hard to define in this context), but also striving for ease of use and expressive power. This is not the right moment to go into all the considerations that came into play, but rather just to note that the user, if he or she adopts Pd, is adopting all this as part of the bargain.

In keeping with the generating metaphor of Pd (a workbench for designing electronic musical instruments for live music performance), objects in Pd make their calculations largely as reactions to inputs, as a piano reacts when you depress a key, for instance. The meaning of control connections (one object's output triggering another's calculations by sending it data) is not at all a given in a graphical programming language; connections could alternatively have represented order of computation, or dependency, or a constraint relation, or something else again. The particular way it's done in Pd is chosen to make it easy to build patches whose behavior is reactive.

An easily made critique of Pd (and of Max whose behavior in this respect is essentially the same) is that its heavy use has given rise to pieces of live electronic music that suffer from a stereotypical reactivity to gestures from live performers. This is particularly acute in live music incorporating solo classical instrument players whose instrumental sounds are augmented by Pd (and this is indeed the historically original Max configuration from IRCAM in 1988). The prevalence of over-reactive and over-obviously reactive pieces of live electronic music in today's repertory can be partly blamed, perhaps, on the fact that Max's and Pd's designs make it so easy to code up that sort of knee-jerk behavior.

The deadly embrace

Musicians can't do much today without software, and so they are dependent on software developers. Software developers in turn are dependent on "users" (the musicians) to make artistic creations with their software; without that, the work of software development is pointless. The software developer strives to impose as few stylistic restrictions as possible on the musician. Yet every new generation of software that comes along reveals possibilities that were somehow not made possible, or at least not encouraged, by the previous generation. Soon we will learn that, no matter how general and powerful we believe today's software to be, it was in fact steeped in tacit assumptions about music making that restrict the field of musical possibility.

We software developers can watch the interactions between artists and our software creations in order to try to see where the software is helping, and where it is hindering, artistic creation. We should seek to empower the artists to the extent that we can, and doing this requires long and serious thought about the implications of what we are making and doing. Do Pd's design choices, for example, get in the way of some or another thing the musician is trying to do, and if so, is there a way to ameliorate it?

There is also a more subtle, and perhaps more fundamental, aim: to make it so that the software doesn't impose one or another stylistic bias on the musician. Such a bias might be easy to spot (a built-in set of available time signatures or musical scales, for instance), or might be so ingrained as to be almost invisible (for example, Max's and Pd's orientation toward reactivity that seems to privilege some approaches to real-time performance over others). Ideally, it should not be the case that the choice of software used to realize a piece of music makes a perceptible stamp on the music, in either overt or more subtle ways. And yet this reasonable-sounding goal seems always to recede as we try to approach it.

I think that musicologists might find some excellent research questions here. How, exactly, are the implications of software design choices insidiously affecting the practice of music composition and performance today? If software developers like me knew more about this, we could use the knowledge to inform our designs. The thing that limits us is not so much the time spent writing the code, as it is the limited understanding we have about what is needed.