A case study in software for artists: Max/MSP and Pd

Miller Puckette
Department of Music
University of California, San Diego
msp@ucsd.edu

Reprinted from David-Olivier Lartigaud, Ed., *Art++*, Editions Hyx (2009).

July 5, 2016

There's no guarantee that someone who has long worked in any given field will be the best person to describe the workings of that field itself. So the following writings about software and art, written by one who makes the software, should be regarded as coming from a special point of view. It's an experienced one, to be sure, but it's nothing like an unfiltered one.

The following thoughts are drawn from my experiences of the last twenty-seven years, trying to write software for artists wishing to realize interactive works of art. At the beginning I focussed more narrowly on live performed music using computers; but the scope of the project has gradually widened to include other media (video and graphics; networking; robotics) and other modes of presentation (installations; studio production). That said, my own personal interest is in new musical possibilities.

Although it is possible for software to project itself artistically, and/or for art to address software or the overall computing experience itself, I will take a more conventional attitude in which software, and the computer, are regarded as tools for art making, to be seen in the artist's studio but not directly by the viewing public. Still, the design of software for art-making has a crucial impact on what the artist will eventually be able to create, and an appreciation for what artists are making, or want to make, should ideally be at the root of the software's design.

Software and the artist

Software artists make software, of course, but at one level or another any software artist must use tools. These might be the toggle switches on the panel of a PDP8 or the newest HCI gadget; in any case there is always some interface or other and, therefore, always the need for someone to think about the design of interfaces that could be of use to an artist. How can this be done in such a way as to make it easy for the artist to do things, the best of which probably weren't at all in the mind of the tool designer?

To measure a software designer's success at enabling art-making would be impossible, so there's no way the software designer can determine systematically whether one design would be better than another one. Whatever guidance can be found will probably be vague and sometimes contradictory. Instead of metrics and specifics, the best that can be offered is intuitions and generalities.

There is an old Romantic ideal of the technophobic artist who deals in feelings and can operate only in isolation from the practicalities of materials and techniques. Artists working in electronic media can't afford to think this way, because wielding electronic media in an original, artist-driven way requires a full and deep understanding of the medium itself. In the specific case of arts using computers, the large, easy-to-use packages for editing sounds and images have become so nearly ubiquitous that the computer landscape is now overfilled with their products: expertly edited images and sounds. Artists who wish to explore a different path from the well-tended ones must take a more active stance in the way they use their tools.

Tools for electronic art-making can be arranged in a spectrum reaching from finished applications to development environments. Any real software tool will lie somewhere in between these extremes. It's easy to suppose that in more creative, exploratory art forms, the artist will have to lean to the "development environment" end of the spectrum. It's also easy to make the assumption that more sophistication is required on the part of the user at that end.

But the distinction is hard to define unambiguously, and at bottom might even be an illusion. Photo editors, at the "finished application" extreme, often have some capacity for scripting; development environments, on the other hand, are no more than special sorts of applications or suites of applications.

One possible way to describe the difference is that applications "do" things but programming environments wait for the user to do them. A programming environment gives the user an empty screen to project his or her imagination onto. At the other extreme, an "end user" application presents all the choices its creator was able to implement as pretty GUI widgets. Such an application looks much the same after strenuous use as it did at the beginning.

The idea that more effort or mastery is required at the programming environment end of the spectrum, on closer inspection, appears to be simply wrong. Whatever the orientation of a software package, over its development it always tends toward greater and greater complexity, the only bound being the cognitive limits of the user. The user, in turn, attains a level of skill at using the software that is limited only by the time available.

Here is an alternative way of thinking about software and its "reach", that is, the ensemble of its possibilities as a tool. Suppose we are comparing two software packages, A and B. At bottom, each one can be thought of as taking a stream of ones and zeros (coming from the user) and translating them into a finished product such as a soundfile. Supposing that A and B are both sufficiently general, any output could be generated by either A or B. However, if we limit the size of the input to something specific, a megabyte for example, there are only 2-to-the-millionth-power outputs that can be reached (fewer, actually, since it's likely that many possible strings of input will result in exactly the same outputs).

If we assume, therefore, that the software designer and the user conspire together to make the user's reach as large as possible, then in some sense the overall size of the set of things the user can do is invariant, regardless of the basic approach taken in the design of the software.

It might be that certain strings of input are more accessible to the user than others; for example, if the translation is carried out via a programming language, there might be tricks that the user would have to invest time in mastering, thus costing more per bit. In principle at least, we could correct for such variations by replacing the overall length of the binary input supplied by the user with another measure that reported either the time spent preparing the input, or perhaps even the Shannon entropy of the bit stream; in any event, conceptually at least, we still get some set of reachable outputs, and its size for the software package A will be comparable to that for B.

Viewed in this way, to design a good tool for making computer art is to maximize the variety of landscape covered by a set of limited overall size. Unfortunately, this leaves in the air the meaning of the word "variety", and it's hard to see how to define it. We know it when we see it; but then, some new idea may open up possibilities we hadn't thought of before and make what previously looked like a widely varied landscape later appear to occupy a small niche within the new possibilities.

Yet another possible measure of the usefulness of a software tool would be simply to evaluate whether or not the software is fun to use. This, like the "variety" measure above, is one that computer science sheds no light on. As an intuitive guess, if something is fun, that means the person having the fun is getting his or her mind exercised somehow. Things that are fun to do are more likely to lead an artist in unfamiliar and potentially productive directions.

Fun is measured by the fun-haver and not by the software designer, of course. Moreover, the fact that Grand Theft Auto is fun to some people does not in itself make GTA a useful tool for artists. So in some way, we need stuff that's fun for creators as opposed to fun for consumers. Somehow the software designer must guess what artists will think is fun.

Whether the software designer wishes it or not, software unfailingly projects his or her mentality and personality. Emacs is like Richard Stallman and Csound is like Barry Vercoe. When you use software you are spending time with its creator. Your choice of one software package over another reflects whether your and the creator's personalities are compatible or not. For the software to be successful requires that the personality it projects be engaging and direct, not fussy or pretentious. This is yet one more trait that computer science lacks tools to measure or foster.

Except for those few occasions when we actually use one in public, when we're at the computer we're doing office work (Zicarelli 1992). Whether we're reading and writing e-mail or making a new patch, the basic operation, and its effect on our moods and thoughts, are the same. As an office tool, software should be functional and not attention-grabbing; the creator, virtually present as your office mate, should only speak when spoken to.

Our working habits, and perhaps our minds themselves, are warped by the experience of using a computer. The software designer must be mindful of this, and avoid behaviors or decorations that do not essentially contribute to the user's aims. For example, extensive use of color and animation, endemic in today's software, mainly serve to pollute the airspace between the artist and the computer. Ideally, software should present the the most aesthetically neutral surface possible.

Software and "larger" concerns

Beyond the question of how to make software useful in the here-and-now (which one may think about by imagining a single artist working on a single project using a computer somehow), there is the question of how the aggregate of all artists, with all their hardware and software, evolves over time and contributes to our culture. The thoughtful software designer must consider the global aspects of the work as well as the local.

Artists are caretakers and maintainers of our culture. When surrounded by good art, we think more freely and are more aware of ourselves and our environment. Although the presence of a healthy art scene is by no means sufficient for a society itself to be healthy, its lack would be a serious problem, like an essential mineral missing from one's diet. The value of having good software for art is indirect: the software designer plays a non-starring, but essential, role in the arts scene.

Some aspects of what an artist does may actually appear original; Western culture values originality, implicitly in agreement with the belief that civilization is on a large march out of darkness and ignorance into enlightenment. Another Western fancy is the idea that the artist should actually somehow receive credit for this originality; and in more general terms, that ideas should somehow be the property of their originators. In the U.S. one often hears that financially encouraging "originality" (by protecting intellectual property) will act as an incentive for people to innovate. Although those who say this are often thinking about discovering new drugs or weapons, considering that "entertainment" is now one of the major national products of the U.S. in economic terms, this would suggest that artists, too, will be encouraged to be more creative, therefore creating more cash value, if their ideas are reified into things that can be owned, bought, and sold.

Original ideas may also pop up in one or another aspect of the artist's software, and software itself sometimes attracts praise for being original. This is not true of most other artists' tools: one praises the quality of a paintbrush, not its novelty. And indeed, originality in a software tool for the artist can be a very bad thing, because the race to introduce original ideas into software destabilizes the software base itself. There is a tension between the desire to overcome limitations in the current state of the art (which presumably requires that we find new, original solutions to problems) and the need for stability in the tools (which we need so that we can maintain our stock of electronic documents, including works of art). As the work of art itself becomes more and more entangled with the technology that projects it, it becomes harder and harder to keep artworks alive in the face of the constant upheavals in software carried out in the name of finding novel solutions to our many vexing problems.

To borrow Zicarelli's metaphor, the artist and the software designer are two species within an ecology, mutually dependent, and both almost powerless to change the larger forces that move them. The user and his or her computer are locked in an embrace, with each party exercising partial control over the other.

The software designer always leads the artist down some path or other. The attempt to make "useful" functions easy to carry out always has the effect of privileging whatever functions the software architect thought would be useful in the first place. The artist is helplessly pulled along (except that he or she can always host another software parasite).

On the other hand, we may take the opposite point of view in which the artist is the breeder and the software designer is the dog. By the act of choosing between and combining software programs, the artist affords progeny to some set of software "genes" at the expense of others. The software designer has no real power; whatever the product of his or her hard work, the artist simply chooses either it or something else, leaving all non-selected software without issue.

A third point of view regards art and the larger culture that contains it as nothing but plumage, held up by members of the human species to attract mates. From this point of view, we're all being selected, artists and software developers alike, on the basis of concerns which at bottom have nothing to do with either skill or aesthetics, but rather our disposing of sufficient extra energy after feeding ourselves to carry on cultural activities.

Designing Max/MSP and Pure Data

I have been working on the programs Max/MSP and Pd (Puckette 2002) over a more than twenty-year period, during which time I also developed the opinions presented here; so it can't be said that Pd (the program I'm working on now) was designed with all the preceding considerations already carefully laid out. A much more traceable influence over the development of the software has been the long string of collaborations with many artists that I've had the good fortune to work with over these years. It's possible to examine whether and how well Pd reflects my opinions about software and art, but it's likely that the opinions are at least partly formed as a result of the experience of writing the software. In other words, the code may to some extent have shaped my own expectations for it—the exact opposite of the specification and realization process practiced by software engineers.

Perhaps the most fundamental consideration that went into Pd's design was generality. The design leans toward that of a programming environment, although in comparison with production programming languages such as C, the design of Pd feels more like a scripting language such as a Unix shell. Further, the widespread impression of Pd as a programming language has led in the past to confusing comparison with programming languages proper (Desain 1993). A better description of Pd might be that it's an interconnection mechanism, with which one puts applications together by configuring smaller primitives ("objects") such as adders or oscillators. Most users rely on pre-existing objects, of which some 200 come with Pd and thousands more are available from else-where on the web. (This is a much richer set of primitives than is provided, for instance, by the C programming language; it's more nearly comparable in size and range to an API such as OpenGL). Implicit in Pd's design is the choice of its particular point in the tradeoff between generality and ease of use.

Pd's design as an interconnection tool has made it enormously extensible, and this extensibility has been a major factor in its acceptance by visual artists, which one wouldn't necessarily expect of software designed for music production. The fact that Pd doesn't address any medium at its core (unless the passage of time itself can be considered a medium) has made it easy to introduce packages such as GEM (Danks 1997), which allows Pd patches to make OpenGL calls. This was not so much a windfall for Pd as the result of careful planning.

Pd's design reflects the environment of three busy music production houses (at MIT, then IRCAM, then UCSD) I worked in over my career in computer music. Throughout this period I've been in a series of music productions, working under time pressure and relying heavily on the software, both in the studio and on stage. The fact that most of the productions I've been involved in were for performance live on stage meant that my number one concern has been reliability. That as Pd's programmer I was also spending many long hours actually using Pd explains Pd's no-frills presentation; colorful "skins" and animations are things a programmer puts on to impress non-programmer users. I had no desire or ability to impress myself by presenting anything at Pd's surface that was not also at Pd's core, since that was the point of view I regarded Pd from. I myself was the user base I aimed Pd to suit. And so Pd now invites any other user to look directly upon its innermost functionality with no more mediation than seemed to me absolutely necessary.

As a tool for making music, Pd's design reflects a deep concern with issues of time and causality. Music is a thing that comes out of an instrument as the instrument is being played, and the gestures by which the instrument player causes the sounds to come out are not things that can be understood outside the flow of time that passes while the gesture is being made. Pd's design aims to make it easy to make the computer react as a musical instrument. This consideration is what leads Pd to deal mostly in flows of data, rather than mechanisms for storage and retrieval. (Indeed, data storage is one of Pd's most glaring weaknesses.) The particular way in which data, control, and the flow of time intermix in Pd was only arrived at after more than a decade of thought and experiment, and is at the center of Pd's design. Pd is as a pure expression as I can find of the idea of reactivity in a computer program.

Pd lacks many features considered essential in a modern programming language; perhaps the most obvious example of this is Pd's flat namespace. There is no scoping in Pd, no notion of "local variables". The omission is intentional; scoping would be appropriate in an environment that encouraged building large, complex systems; the systems to which Pd is best adapted are comparatively small, aiming to realize a single work of art or piece of music. Another feature common to programming languages that Pd lacks is a standard ensemble of mechanisms for making loops, jumping to subroutines, and whatnot. This omission was not my desire at all; it is simply hard to see how to introduce these capabilities naturally into a graphical tool such as Pd.

Another notable absence in Pd is that of any built-in semantics. In the place of pitches, dynamics, rhythmic values, and all the other musical quantities one finds in a musical notation editor, in Pd one has only numbers and an occasional string. These data can "trigger" actions as well as parametrizing them. (It's also possible to trigger an action without any data at all using the coyly named "bang" message.) This lack of built-in semantic is more reminiscent of C than of the Unix shell, and is the aspect of Pd in which its "level" is at its lowest. The intent was to avoid making any sort of suggestion to the user as to what quantities should be in play in a musical or artistic project.

Although Pd's data structures and storage mechanism are arguably its weakest aspect, this is not for lack of desire to manage that aspect better; indeed, Pd's name reflects my recent desire to focus on the structure and management of data. The process-like aspects of Pd are not fundamentally different from those of Max/MSP, and Pd's most novel features relate to "data". Since this aspect of things had to wait until everything else had been essentially set in concrete, the challenge has been to adapt the traditional approaches to data management in software environments to a structure that wasn't fundamentally oriented toward them. This is the software design question that most occupies me at present.

A major shortcoming in Pd's design is that there is no simple way to adapt it for multiprocessing. Multiprocessing was very important in the 1980s and early 1990s, when uniprocessors did not have enough power for serious computer music applications; an early version of Max I worked on was explicitly designed for multiprocessing. From the user's point of view, adapting a patch to use several processors efficiently was so complicated that it was a relief when uniprocessors came along that could replace the multiprocessors of that era; and from the outset in the design of Pd I focussed on uniprocessors. Recently, however, multiprocessing has become so cheap that most new PCs have at least two processors. Finding a better way to take advantage of this new possibility should be a high priority in the digital arts; but it is not one which Pd seems likely ever to address well.

One final aspect of Pd's design: it is fundamentally "Free" software, and this quality is inseparable from many elements of Pd's design style. The ease with which Pd can be adapted to run on a PDA or embedded controller derives partly from the fact that users can grab part of Pd without using the whole thing. Pd's Free status is insurance against "feature creep" in that, if users want a feature, they can add it themselves, so there's less pressure to add stuff to Pd itself. It's also insurance against incompatible changes or disappearance, in that a user who really wants the old Pd version 0.21 can simply hang onto the source. The boundary between user and developer is gone; this greatly increases flexibility.

What I learned from it all

Now that I know something about making software environments for making art, I should be in an excellent position to start writing the Next Big Thing in software and art. Unfortunately, I almost certainly won't. First, I have Pd to take care of; I'm hoping to keep it running compatibly with existing patches for decades to come, so that artists can finally have a platform capable of supporting long-lasting works of art. Second, I'm tired of writing software and want to do some other things (never mind exactly what). But most importantly, the things I learned are relevant to writing 1980s or 2000s era software; what's appropriate for 2015 is likely to be something entirely different.

Making interactive software environments for time-based arts is a problem space that was wide open in the 1980s but is nearly closed now. This is not to say that ideal solutions exist; but adequate ones do, and in the presence of an adequate solution the perfect solution can't develop. Like the 'ERTY keyboard, the software applications that exist now, assuming they can be maintained stably, will by their very presence oblige people to learn them, and the "user base" will perpetuate the software.

It is often said that computer techniques change constantly, but this is not strictly true. In areas of new exploration there is change and surprise, but in areas such as programming languages and processor design, where acceptable solutions have been proposed, further change is impossible. Whatever changes await us now will take place in new, unexplored areas, and for that reason the experiences learned from Pd's design will only be of limited help to people wanting to write new software. Some of the lessons reported here will certainly still apply, but it's impossible to predict which ones, nor in what way the others might fail to hold in some future context.

It is at least possible to suggest some things that might not be the same in the future as they have been in the past. As mentioned before, the boundary between "users" and "developers" is weakening and may either disappear or change into something else. Models of publishing are also changing in ways that may radically change the possibilities available to artists. Transportation will become much more expensive and communications will gradually become cheaper. We can't know how these forces will play on the arts scene, but their effects will be profound.

One trend in which the development of Pd has played a part seems likely to continue: software is no longer best considered as a fixed entity that operates on morphable "documents" like an office suite on spreadsheets. The "culture" that is at play lives less and less in the document and increasingly in functionality itself. The ideas that we exchange are less in the form of passive data and more in the potential to make, change, and use data in new ways. The functionality that software carries is itself information, of a harder-to-define but potentially very potent kind.

As culture is increasingly transmitted in functionality and not in passive documents, the days in which a few widely-used pieces of software dominate the landscape are numbered. While certain environments will continue to be used because of the 'ERTY effect, more and more of the real work will take place outside of the fixed environments, or in the spaces between them. The heroic era typified by emerging "environments", in which reputations were earned by inventing the spreadsheet or the multi-tasking operating system, is near its end. It may be that the big, widely influential cultural gesture itself is becoming rarer, or it may be that the gestures that will now be made will come in very different guises than software. Software itself in the future will be only as interesting as, say, automobiles are today: not all the same, but better regarded as useful than beautiful.

References

Danks, Mark, 1997. “Real-time image and video processing in GEM.” In Proceedings of the 1997 International Computer Music Conference (Thessaloniki), pp. 220-223.

Desain, Peter and Honing, Henkjan, 1993. “Letter to the editor: the mins of Max.” Computer Music Journal 17/2, pp. 3-11.

Puckette, Miller, 2002. “Max at 17.” Computer Music Journal 26/4, pp. 31-43.

Zicarelli, David 1992. “Music Technology as a Form of Parasite.” In Proceedings of the 1992 International Computer Music Conference (San Jose), pp. 69-72.