



PROJECT MUSE®

Max at Seventeen

Miller Puckette

Computer Music Journal, Volume 26, Number 4, Winter 2002, pp. 31-43 (Article)

Published by The MIT Press



➔ For additional information about this article

<https://muse.jhu.edu/article/37587>

Miller Puckette

Department of Music
University of California, San Diego
La Jolla, California 92093-0326 USA
msp@ucsd.edu

Max at Seventeen

I have worked for many years on a computer environment for realizing live electronic music that I named in honor of Max Mathews. Three currently supported computer programs—Max/MSP, jmax, and Pd—can be considered as extended implementations of a common paradigm I refer to here as “Max.” The Max paradigm appears to be stable enough now that a printed description of it no longer risks becoming too quickly outdated. We can now usefully assess what Max (the paradigm) does well, what it does less well, and what we can all learn from the experience. The Dartmouth Symposium on the Future of Music Software, organized by Eric Lyon, offers the perfect occasion to start this project. I apologize in advance if, for obvious reasons, parts of this article might not seem perfectly objective or impartial.

The Max paradigm can be described as a way of combining pre-designed building blocks into configurations useful for real-time computer music performance. This includes a protocol for scheduling control- and audio-rate computations, an approach to modularization and component intercommunication, and a graphical representation and editor for patches. These components are realized differently in different implementations, and each implementation offers a variety of extensions to the common paradigm. On the surface, Max appears to be mostly concerned with presenting a suitable graphical user interface for describing real-time MIDI and audio computations. However, the graphical look and editing functions in Max are not highly original, and most of what is essentially Max lies beneath the surface.

In my experience, computer music software most often arises as a result of interactions between artists and software writers (occasionally embodied in the same person, but not in my own case). This interaction is at best one of mutual enabling and mutual respect. The design of the software cannot help but affect what computer music will sound like,

but we software writers must try not to project our own musical ideas through the software. In the best of circumstances, the artists remind us of their needs—which often turn out quite different from what either of us first imagined. To succeed as computer music software writers, then, we need close exposure to high-caliber artists representing a wide variety of concerns. Only then can we identify features that can solve a variety of different problems when in the hands of very different artists.

Many of the underlying ideas behind Max arose in the rich atmosphere of the MIT Experimental Music Studio in the early 1980s (which became part of the MIT Media Lab at its inception). Max took its modern shape during a heated, excited period of interaction and ferment among a small group of researchers, composers, and performers at IRCAM during the period 1985–1990; writing Max would probably not have been possible in less stimulating and demanding surroundings. In the same way, Pd’s development a decade later would not have been possible without the participation of the artists and other researchers of the Global Visual Music Project, of which I was a part. My work on Max and its implementations has been in essence an attempt to capture these encounters in software, and a study of Max will succeed best if it considers the design issues and the artistic issues together.

Background and Influences

The Max paradigm and its first full implementation were mostly developed over the period 1980–1990, drawing on a wide variety of influences. The most important was probably Max Mathews’s RTSKED program (Mathews and Pasquale 1981), some of whose key ideas also appeared in Curtis Abbott’s earlier 4CED program (Abbott 1980). RTSKED attacked the problem of real-time scheduling of control operations for a polyphonic synthesizer.

Whereas Mathews's earlier GROOVE program (Mathews and Moore 1970) emphasized the handling of periodically sampled control voltage signals, the notion of control in RTSKED was one of sporadically occurring events that caused state changes in the synthesizer. For instance, starting a note in RTSKED might involve setting the frequency of an oscillator and triggering an envelope generator. This is similar in style to the "note card" approach of the Music N languages, which have an explicit, instantaneous action time. Hence, the Music N languages are better suited to describing a keyboard instrument than, for example, the continuously changing position and downward force of a violin bow.

The main problem addressed in RTSKED was how to replace the note card in the situation of real-time performance where the action time is not predefined. (Other parameters might not be predefined either, but the chief difficulty seems to be presented by the time parameter.) RTSKED modeled the performance as a collection of tasks running in parallel. For example, each task might have responsibility over a specific synthetic voice. The timing of the tasks was controlled by *wait functions* and *triggers*. Only one task ran at a time, but the running task could relinquish control at any time by calling a wait function and specifying a trigger for which to wait. Triggers could come from real-time input (the keyboard, for example) or be "fired" by other tasks.

This separation of the real-time control problem into separate tasks was a key advance necessary for the computer to act as a musical instrument. Previously, computer music programs (and indeed almost all computer programs in general) enforced a sequence of actions on the user. The separation of the program into parallel tasks permitted the user to control the sequence of execution of the program by selecting which task to trigger next. For example, in a multi-tasking environment, we could describe a piano as 91 tasks: one for each key and pedal. The performer—the piano "user"—chooses in which order to trigger the 91 tasks and how many times they will be triggered. The piano enforces no pre-determined sequence.

This notion of task can be seen clearly in the boxes of the Max paradigm, which "trigger" each

other via their connections. The idea predates the invention of MIDI (and Max was never conceived as a MIDI program), but the availability of MIDI input and output for Macintosh computers was convenient for early Max users. MIDI shares the sporadic quality of control events which have their roots much earlier in the Music N programs (and even much earlier still, in the clavier keyboard that MIDI models). The parallelism so visually apparent in a Max patch is intended to allow the user to make computer programs that follow the user's choices, not the program's. This was necessary so that Max patches could work as musical instruments.

A second, crucial influence was that of my teacher Barry Vercoe, under whom I studied between 1979 and 1987. Mr. Vercoe, himself a student of Mr. Mathews, is most widely known as the author of Csound (originally called Music 11), but his less-acknowledged work on real-time computer music systems—dating back to the mid 1970s with the design of a real-time digital synthesizer, and continuing today—will perhaps someday be acknowledged as his most important contribution to the field.

Most of the advances in Csound over previous Music N languages lay in its highly refined control structure. Although other Music N systems were in some ways much more flexible (Music 10, for instance), their flexibility derived from allowing the user to write code directly in the underlying implementation language—often the machine's assembly language. Csound, on the other hand, presents constructs in its own language for solving musical control problems. Some of these constructs look baroque (e.g., *tigoto*), but the central concepts—initialization, re-initialization, and performance—provide a way to specify control structure and sample computation in a tightly integrated way.

This thoughtful approach to control issues also guided Vercoe's development of the Synthetic Performer (Vercoe 1984; Vercoe and Puckette 1985), which first ran on a 4X machine at IRCAM. The Synthetic Performer demonstrated real-time *score following*, in which a computer performer automatically synchronized itself with a live player playing a different part. Roger Dannenberg discovered score

following independently and showed it the same year (Dannenberg 1984). Dannenberg's algorithm performed the analysis portion of the problem more robustly than Vercoe's. However, whereas Dannenberg's design was concerned only with controlling the tempo of a sequencer, Vercoe's system worked at the audio level, obliging him to tackle the problem of managing envelope generators in the face of changing tempi. A representative task would be to guide a glissando to reach a specific note on a future downbeat whose arrival time is constantly being re-estimated in the face of new tempo information.

By the 1986 International Computer Music Conference in the Hague, many researchers were showing their work on the abstract problem of scheduling real-time computer music performance (Anderson and Kuivila 1986; Boynton et al. 1986; Favreau et al. 1986; Puckette 1986). The systems proposed differed in their use or non-use of context switching and preemption, the design of queues for scheduled future tasks, the implementation platform and language, and many other respects; but many of the threads that fed into the Max design can be seen there.

Another aspect of Max is its graphical user interface (GUI). The Max GUI has many antecedents. In 1980, while studying under Barry Vercoe, I saw the Oedit system by Richard Steiger and Roger Hale. This system, apparently never described in a published paper, allowed users to design Music 11 orchestras using a visual "patch language." Many other graphical patch languages—both for music and for other applications—had appeared by 1987 when I started writing the Max "patching" GUI. Although several specific elements might have been novel—at least in the computer music context—the overall idea of a graphical patch language was not.

Development of Max, jmax, and Pd

The story of how the Max paradigm and its implementations progressed to their current state is probably not well represented by any one person's understanding of it, but the following first-person

history should at least shed some light on things. Others may wish to add to or amend this account.

The first instance of what might now be called Max was Music500, which I worked on in Vercoe's laboratory starting around 1982 and one aspect of which was reported in some detail (Puckette 1984). There was no graphical front end. The system consisted of a control structure inspired by RTSKED and a synthesis engine distilled from Music 11. The control and synthesis environments were separate. Control was considered as a collection of *processes* that ran in parallel (logically, at least) and could be implemented on parallel processors. The "orchestra" (or *synthesis process*) looked like any other control process, and its communication with the other control processes obeyed the same rules as they obeyed among themselves.

The system was intended to run on machines where the "number crunching" could not be realized in a high-level language. The specific hardware envisioned, the Analogic AP500 array processor, had a bit-sliced, micro-codable number-crunching engine and a "controlling" microprocessor. The orchestra language was written to be interpreted (in vector unit generators) in the arithmetic unit, and the control processes were to be multi-tasked in the microprocessor. The idea of using a multi-tasking control structure came directly from RTSKED.

Music500 never got to the point of making sound in real time. By 1984, when I was still reporting on Music500, Barry Vercoe had written the Synthetic Performer at IRCAM, and he invited me to work with him in Paris in Summer 1985. I got access to a 4X machine—then the most powerful digital synthesizer/audio processor in the world—just as it came out in its printed-circuit-board version. The 4X, having no jump instruction, could not be made to fit into the Music500 orchestra model. (Besides, there was an excellent, text-based "patch language" available, described by Favreau et al. in 1986.) As a result, I only kept Music500's control structure in programming the 4X, renaming it "Max" to acknowledge the fundamental influence of RTSKED. There was no graphical programming language, as the 4X's control computer had no GUI support, and I was still skeptical of GUI-based software at the

time. The “patch” was specified in Max’s command line by simply concatenating the creation arguments of all the objects desired. Because objects could have only one inlet and one outlet apiece in that version, connections could be specified simply by naming each object and listing its destination objects, by name, in its creation arguments.

One of the first uses of Max was supporting Vercoe’s Synthetic Performer, a version of which we developed together at IRCAM in 1985. The command-line patch consisted essentially of four objects: a pitch tracker; the score follower itself (whose input was pitches and whose output was tempo); a tempo-controllable sequencer, and a controller for a sampling synthesizer running on the 4X (which acted as an output device roughly analogous to Max’s *noteout*).

This 4X version of Max made its stage debut in April 1987 in Thierry Lancino’s *Alone* and Philippe Manoury’s *Jupiter*, which were both premiered during IRCAM’s tenth anniversary concerts. That early version of Max imposed some almost maddening limitations. There was a complicated and lengthy compilation process to prepare the 4X parameter updates so that Max could use them. Debugging was sometimes very difficult.

After this experience, I tried implementing the Max control paradigm in two separate Lisp environments before once again rewriting it as a C program on a Macintosh starting in Summer 1987. (The Macintosh in question was brought to IRCAM by David Wessel, without whose efforts I and the rest of IRCAM might have entirely missed out on the personal computer revolution.) The new Macintosh version of Max, which eventually grew into what is now Max/MSP, was first used on stage in a piece by Frédéric Durieux in early 1988. But it was Philippe Manoury’s *Pluton*, whose production started in Fall 1987 and which premiered in July 1988, that spurred Max’s development into a usable musical tool. The *Pluton* patch, now existing in various forms, is in essence the first Max patch.

To realize *Pluton*, we connected a Macintosh to a 4X with a MIDI cable, using the Macintosh as a “control computer” and keeping all audio computations on the 4X. We thus did not have to confront the problem of specifying the audio signal patch

graphically; we still used the 4X patch language, this time together with ad hoc C code to convert MIDI messages to 4X parameter changes. Owing to the limitations of the day (for example, one could have a GUI or a programmable DSP engine, but not both in the same address space), Max became a “MIDI program.” There was nothing fundamental in Max that was derived from MIDI, however; as far as Max was concerned, MIDI was an input/output interface and nothing more.

The next two steps were taken in parallel. First, Max was commercialized through the efforts of David Zicarelli. Zicarelli has persisted in this project for about thirteen years, withstanding the bankruptcies of two companies for whom he worked, and finally starting his own, Cycling74. The world of computer music has rarely if ever seen a software developer of greater dedication and fortitude than Zicarelli.

At the same time, I joined a team at IRCAM directed by Eric Lindemann, charged with creating a successor for the 4X. As Manoury pronounced, “The 4X is an old lady now: still beautiful, but too expensive.” Lindemann designed an extremely powerful audio signal processor called the IRCAM Signal Processing Workstation or ISPW (Lindemann et al. 1991). Among its many advances over the 4X, the ISPW offered a jump instruction. I took advantage of this to add to Max a collection of “*tilde objects*” to do audio signal processing (Puckette 1991a).

This forced two changes on the IRCAM fork of Max: it was necessary to port the graphics layer to a different window system, and, more fundamentally, it became necessary to separate the “real-time” portion of Max from its GUI. Now, the real-time portion ran on the ISPW and the GUI on the host computer, a NeXT Cube. Moreover, the real-time component, which I called Faster Than Sound or FTS (Puckette 1991b), had to run on multiple processors without the benefit of shared memory.

By 1991, the time appeared ripe to make Max/FTS available on other architectures besides the ISPW. Over the next two years, Joseph Francis and Paul Foley got Max/FTS running under Unix and X Windows. The Max/FTS project has since been ex-

tensively reworked and extended at IRCAM by a team led by François Déchelle, who now distribute their version under the name jmax.

Meanwhile, I moved to UCSD in 1994, and, seeing several improvements I wished to make in Max, started writing a new program named Pure Data, or Pd (Puckette 1997). I made Pd an open-source program from the beginning (and IRCAM later did the same with jmax). For Pd, I wrote new tilde objects as in Max/FTS, which Zicarelli then used as a point of departure for what he called MSP, with which Cycling74's Max also became capable of audio signal processing. (By 1996 or so, generic personal computers had finally become powerful enough to make this worthwhile.)

In 1995, Mark Danks started developing GEM (Danks 1997), which has matured into a three-dimensional graphical rendering extension to Pd now maintained by Johannes Zmölzig. Other unrelated extensions to Max/MSP and Pd are available which deal in various ways with video. Thus, the Max programs are now addressing visual as well as audio domains.

Max/MSP, jmax, and Pd can now be seen as three very different implementations of the same fundamental idea, each with its own extensions that are not available on the others. It would be ill-advised to discuss the relative merits of the three implementations here, because the situation is constantly changing. However, it is actually possible to port patches back and forth between the three, so that users who work with their common features can at least feel that they will have some options if the specific implementation they depend on falls out of repair.

Design Issues in Max

By 1993, Max was in wide enough use to attract the bemused attention of computer science researchers, for example Desain and Honing (1993), who wrote, "For some time we have been surprised at the success of Max and the enthusiasm of its users." The design of Max breaks many of the rules of computer science orthodoxy, sometimes for reasons of practicality and sometimes of style. In the

following paragraphs I attempt to give a rationale and critique of the fundamental elements of Max's design (see also Puckette 1991a).

Max is oriented toward processes more than data. (There is no built-in notion of a musical "score," for example.) If we think of a Max patch as a collection of boxes interconnected by lines, the expressiveness of Max comes from its interconnection and intercommunication facilities, whereas the contents of the boxes themselves are usually hidden from the user. Because the only facilities for storing large amounts of data are encapsulated within individual boxes, one cannot use Max as such to see or edit collections of data. To address the clear need for data handling facilities, certain types of boxes (such as `table`) offer their own editors in their own windows, each in its own way. (Pd offers a new approach to the problem of data representation, but it would be premature to describe it here.)

Because each type of box is free to store data in its own way, there is no uniformity across Max in how data is stored. Certain ad hoc mechanisms are also invented on a box-by-box basis for data sharing; the lack of uniformity of approach is a problem, but on the other hand, enforcing a unified approach might have presented even greater problems.

Officially at least, communications between boxes all take place via Max *messages*, which are extremely simple in structure and standardized in content. Messages are linear lists of *atoms*, which in turn may be either numbers or strings. (In Max and jmax, the numbers may be of two distinct types; Pd eliminates this distinction.) Messages are always headed by a symbol, which acts as a Smalltalk-style message selector. (Messages may appear to begin with a number, but for these messages Max always provides an understood selector, such as `list`.) There is disagreement among commentators as to whether Max should be considered "object-oriented" in light of this message-passing interface, or whether it should be denied the title as lacking many other features common to object-oriented programming languages.

A box's leftmost inlet can take messages with any selector for which the box has a defined behav-

ior. In this way, boxes may have many more behaviors than inlets, and new ones may be added without changing the box incompatibly. Other inlets (besides the leftmost one) only handle one selector each and should be reasonably few in number; they are best reserved for messages that are frequently needed in a patch (such as the channel number in a MIDI output box). This design represents a compromise intended to increase the ease of interconnection without fundamentally restricting the number of selectors a box can take. Its most important inconveniences are probably the confusion caused by the hidden selectors, and the occasional necessity of “prepending” the selector `list` or `symbol` to a message.

The extremely simple structure of the messages, as well as the standardization of the use of selectors such as `bang`, are meant to maximize the interconnectability between boxes and to minimize the amount of “glue” needed. Messages are also easy to type and to read; indeed, the “message box” simply holds a message.

A glaring departure from this principle is in the handling of audio signals. The Max design originally looked at the problem of real-time computer music performance as essentially a control problem, and the facility for patching *audio* signals from one box to another departs radically from the scheme. Max provides a separate data type for audio signals which does not mix smoothly with messages; conversion from audio to messages is problematic, and the scheduler treats control- and audio-rate processing separately. I cannot offer a better solution than what Max provides. I did once design a message semantic that actually included both modalities in one common generalization, but it appeared to be too general to be useful in practice.

In general, the design of Max emphasizes *text boxes* (objects and messages) in which the user types a message to define the contents of the box. In this respect, Max is highly unusual among graphical programming environments. There are at least three rationales for this choice. First, users will always require a text interface of some sort to specify parameters (called *creation arguments* in Max), so to rely entirely on text appears to be the

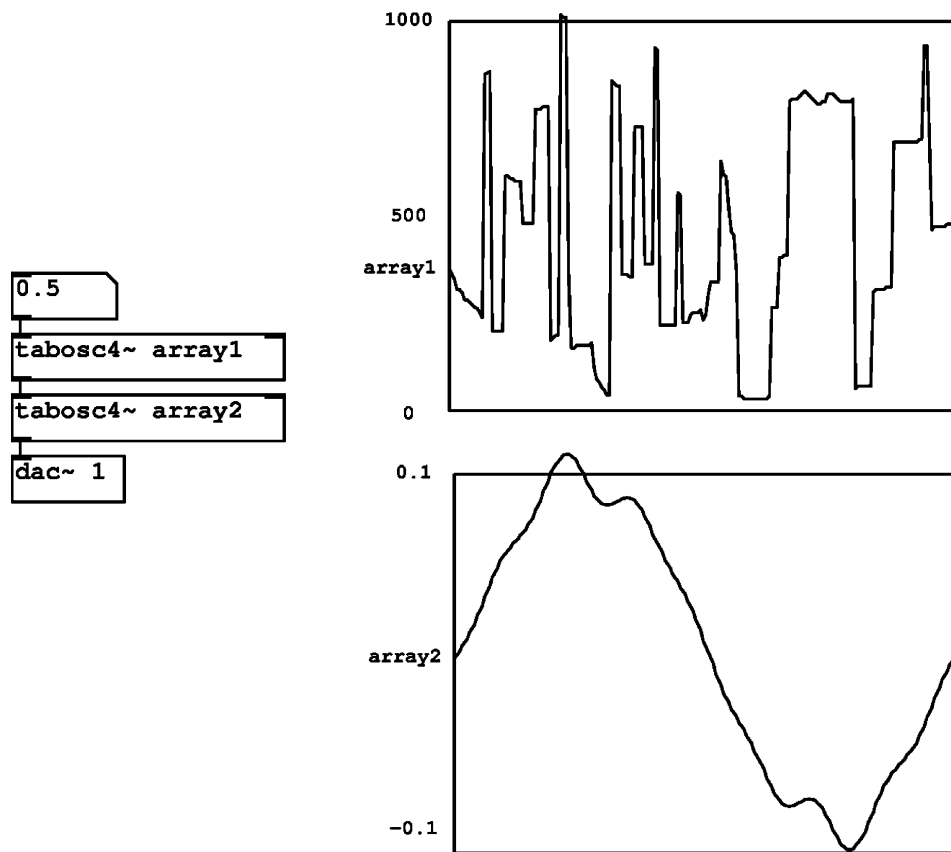
simplest solution possible. Second, having more than ten or twenty distinct icons makes it difficult to remember which icon corresponds to which functionality; text works better mnemonically. Third, the text representation appears to convey more information per unit area of screen than icons can. In most cases, boxes in Max do not rely essentially on dialog windows or other forms of display modality. Ideally, the text content of a box should determine it entirely—at least at the time of its creation. A central design principle of Max is that, to the extent possible, the patch that appears on the screen should provide a full documentation of what is happening. For this reason, boxes avoid hidden variables (that could be managed via dialog boxes, for example), and an object box’s entire state—at least on startup—should be seen in the text inside the box.

This is also the reason behind the choice not to make state changes in objects persistent. When a previously saved Max patch is reopened, all the values that had been sent to boxes’s inlets are forgotten and replaced anew by values supplied by the creation arguments (or, failing that, by their default values). This is done because otherwise a patch’s behavior could depend critically on the fact that a “+” box, for example, had some specific value left over in its inlet, even after all visible traces of its provenance were gone.

In certain cases (tables, qlists, etc.) it becomes necessary to show more state than the user might wish to see in the context of its use. In these cases, Max departs from principle and offers specialized editors that live in sub-windows to manage the data. This is an aesthetically unfortunate compromise, but it is necessary because Max’s fundamental design does not deal with issues of data management.

The Max design strives for simplicity and explicitness throughout. Figure 1 shows an example, in this case implemented in Pd version 0.35, of a patch with two oscillators, the upper one controlling the frequency of the lower one. The economy of gesture in this patch is achieved not by designing modules such as `tabosc4~` to be musically intelligent, but rather by making them as transparent as possible—transparency being an opposite of in-

Figure 1. A concise patch in Pd version 0.35, showing two oscillators. The higher oscillator controls the frequency of the lower one.



telligence. The lack of “intelligence” in no way limits the expressiveness of the visual language. Instead, expressiveness is enhanced by concreteness (the opposite of abstraction), directness, and straightforwardness.

Scheduling

The scheduling algorithm of Max is exceedingly simple, although it took several years to develop. In its final form, it differs from the scheduling algorithm of RTSKED primarily in its approach to data handling. In Max, objects are usually activated by passing messages to them from other objects. The mechanisms for “triggering” an action and for passing data back and forth between objects are unified. (Messages, like bang for example, might be empty

of data, in which case they may be considered pure triggers.) The unification of data “flow” and triggering was suggested by the piano metaphor, in which the velocity of playing a note is naturally included as part of the fact that the note was played. (This does not make Max a true dataflow language, however; in general, dataflow languages seem not to capture real-time event processing well.)

In addition to the visible patch cords, another source of triggering is the passage of *logical time*. Formally, this appears to the object as just another trigger, although for speed and simplicity it is coded as a plain function callback. Logical time marches forward in a purely deterministic way, irrespective of any real-time delays engendered by CPU load or the operating system. This determinism is crucial for making Max patches “debuggable.” If something goes wrong, the problem is usually easy to reproduce and track down, and if a

patch works in rehearsal, it has a good chance of working in performance.

Max follows the piano metaphor also in that objects do not specify what they wait for, but instead they explicitly choose which other objects to trigger. In other words, the choice is up to the triggering object, not the triggered one (the player, not the piano). Graphically, this is indicated by fan-in at an object's main inlet; the object does not control which line will activate it next. Thus, the graphical design and the scheduler match well.

On the other hand, a severe problem arises for connections that fan out, for which no solution seems wholly adequate. The order in which the fanned-out triggers are fired may affect the aggregate result. Unless the fanned-out messages are given an explicit order of execution, fan-out can give rise to indeterminate behavior. One possible solution to this problem, implemented in Cycling '74's Max, is to sort the recipients of each single outlet according to their screen position. This has the advantage that one can in principle see how the patch will work directly from its screen appearance, but the disadvantage that merely moving boxes on the screen (to tidy them up for example) can change a patch's behavior. (Furthermore, it is not clear what rule non-local connections such as `send/receive` should obey.) Jmax and Pd each behave in other ways which have their own advantages and disadvantages. The best solution is for the user to place trigger objects to disambiguate the situation when the order of execution of fan-out matters. Perhaps a good way will one day be found to detect these situations automatically and warn the user of them.

Users of Max are often surprised by the right-to-left order in which multiple outlets of a single box are customarily fired. This is done because a box's "main" inlet (the leftmost one) is used to trigger outputs, and the other inlets are almost always used only to set internal state in the box. For a calculation to work correctly, the leftmost inlet must therefore be triggered *after* all relevant messages have been sent to the other inlets. If one accepts that the first inlet should be the "hot" one, it fol-

lows that it should be the last to receive a message, and if a single object is furnishing two or more of the messages, the right-to-left order will be the correct one.

Many thorny scheduling problems arise from the uneasy marriage of audio and control computation in Max. The fact that audio signals are computed in blocks of at least one sample (and usually 64 or more for efficiency's sake) implies that control computations have an inherent grain, in that they can only affect DSP computations on discrete sample block boundaries. This can be a serious problem when building granular synthesis patches, for example, in which control computations might be required to decide the onset time of individual grains.

Another problem is simply that it is inelegant to have audio computation obey different semantics from control computations. Control computations are inherently order-dependent, whereas the audio sample clock is predictable so that dataflow semantics are more appropriate for audio computations than the message-passing model that Max adopts for control calculations. The two worlds coexist somehow in the same patch, but not without causing confusion and offending computer scientists. In Max/MSP and jmax, audio connections are drawn in a different style from control ones, which at least helps reduce confusion.

Yet another problem is that it is extremely clumsy to perform sample-dependent decision making of the sort that one would use, for example, to trigger a control action at a zero crossing in an audio signal. When control computations are allowed to interrupt signal computations, the results may be difficult to predict or interpret.

All these problems grow in severity when video or computer graphics computations are added. Trying to overcome them might open up fruitful areas of future research. At the moment, however, the Max paradigm seems at least to handle the majority of situations well enough, despite all of its limitations, to be a useful system in practice. To paraphrase Winston Churchill, Max is the worst possible solution, except for all the others that have been tried.

Programming in Max

Musicians have often used Max as a programming environment, at which Max succeeds only very awkwardly. There is no concept of scoping or namespaces in Max; all symbols and their bindings live in one flat space. This decision was made to remove a layer of complexity that did not seem strictly necessary in the context of computer music production, in order to make Max as accessible as possible to people who are not professional computer programmers.

Further, Max lacks any notion of linear “control flow” such as is fundamental in any real-world programming environment. The whole notion of control flow, with loops, conditionals, and subroutines, is easy to express in text languages, but thus far, graphical programming languages have not found the same fluency or economy of expression as text languages have.

Rather than a programming environment, Max is fundamentally a system for scheduling real-time tasks and managing communication among them. Programming as such is better undertaken within the tasks than by building networks of existing tasks. This can be done by writing “externs” in C or C++, or by importing entire interpreters (Forth or Scheme, for example). The exact possibilities depend on the implementation.

Musical Constructs

The design of Max goes to great lengths to avoid imposing a stylistic bias on the musician’s output. To return to the piano analogy, although pianos might impose constraints on the composer or pianist, a wide variety of styles can be expressed through it. To the musician, the piano is a vehicle of empowerment, not constraint. The fact that the output is almost always going to be recognizable as piano music is not a cause for criticism. The computer software analogue is that Max (for example) should not so restrict the musician’s creativity that the result sounds like “Max,” although perhaps we should forgive it for sounding like a computer, as

long as sounding like a computer is understood not to imply any stylistic tendency.

To what extent does Max achieve this? It may be too early to know, because Max is still a recent phenomenon when considered in the timescale of musical stylistic development. Still, it is possible to discuss how the Max design aims for stylistic neutrality and where it might succeed or fail.

On starting Max, the user sees nothing but a blank page—no staves, time, or key signatures—not even a notion of “note,” and certainly none of instrumental “voice” or “sequence.” Even this blank page carries stylistic and cultural freight in at least one interesting respect: the whole idea of incorporating paper in the music-making endeavor is central to Western art music, for which many other musics have no use at all. Musical practices that do not rely on paper may in some cases have much less use for computers than have the more Westernized ones.

Perhaps we will eventually conclude that no matter how hard one tries to make a software program culturally neutral, one will always be able to find ever more glaringly obvious built-in assumptions. But this is not a reason to stop designing new software; on the contrary, it gives hope that there may be many exciting discoveries still to be made as more and more fundamental assumptions are questioned and somehow peeled away.

Software and Technique

On one level, the process of making computer music involves first writing software and then making music with it. A first measure of the utility of the software is its longevity. However, merely writing software with longevity in mind does not directly serve the music maker. Computer music software can also encode ideas that reach beyond any individual implementation of them. Computer music software’s offerings are of two types. They empower the user directly through their presentation as a working environment; but also, they encode advances in computer music practice. The software designer’s most direct challenges are to make the

software useful to a wide variety of tasks and to make it last as long as possible. But the good, perennial ideas will jump from implementation to implementation anyway.

For example, the wavetable oscillator used in Figure 1 made its first appearance in Mathews's Music II (two, not eleven) in the late 1950s. Music II was only one in a long sequence of Music N programs, but the idea of wavetable synthesis has had a pervasive influence throughout the computer music discipline.

At its best, software creation is about enabling people to use the inventions of our day and not about presenting them with alternative inventions that might spring from the mind of the software developer. Invention is an important part of cultural evolution, but we should not confuse invention with software design. Nor should software designers build their private inventions into the software. In situations such as Music II where it was necessary to invent something as part of the software development, the best inventors (such as Mathews) present the invention and its carrier software as two separate, distinct things.

Durability

Two competing desiderata tug at either side of the software developer. On the one hand, we want software to do everything, and our notion of "everything" grows broader every year. On the other, we require stability, so that we can open last year's electronic mail or play a piece of computer music composed six months ago. Every software designer tries to provide the widest possible choice of functionality and simultaneously minimize the risk of falling out of maintenance. The breadth of functionality offered by a new piece of software seems to depend mostly on the ambition of its designer. Its longevity, on the other hand, depends partly on luck, partly on the perseverance of designers (and their employers), and partly on designers's foresight.

The durability of the musical ideas expressed in a medium such as Max is not necessarily limited by any specific implementation of Max. The patch of

Figure 1, for example, offers a nearly complete description of its functionality, and any remaining imprecision would be quickly resolved by looking at the document as a text file. (This would reveal the exact numeric content of the two tables.) One would not need a running copy of Max or Pd to re-interpret those few lines of text. Ideas expressed in Max can in principle outlast their implementation.

This transmissibility of practice and ideas does not depend in any essential way on reuse of code, in contradiction to the preaching of the computer science community. Indeed, if object-oriented programming in the sense of inheritance structures is essentially about code re-use, there does not seem to be any essential reason to use object-oriented coding constructs either. Of course, elegance in code does matter and is a good thing, but it has nothing to do with the creative adaptation of the great underlying ideas that make up the true contribution of computer music to our larger culture. And as to elegance, we can leave it to others to argue the relative merits of different programming techniques and languages.

Musical practice is not hopelessly embedded in music software, any more than the concept of powered flight is embedded in the original Wright brothers's airplane. If we could dig up the original plane, it would in no way help us fly any better than we do now. The essential thing is to maintain the practice. The artifact is fun to have around, of course, but only as a museum piece.

External Issues

Often, aspects of software that are external to its theoretical design have a strong positive or negative effect on its ultimate usefulness. This may partly depend on the way the software is placed in the social fabric. In this respect, the various implementations of Max have occupied a variety of niches. The original program named "Max" was written in a very different world with different modalities of software use and dissemination from today. The existence of three present-day implementations is partly a reflection of the many changes that have taken place over the last 17

years. One major change is that computer music software has now broken out of the confines of what, in 1985, was a rather insular "computer music community."

Today's low computer prices promise to make possible more subtle encounters than in the past between, for example, Europeans and inhabitants of their former colonies. Instead of heading out into the forest with a machine to record the local artists (as we of Euro-centric cultures have been doing at least since Bartok's time), we can now initiate electronic mail conversations. People of almost any community on earth can now record their own music without the help of any modern Bartoks. And people almost anywhere can or soon will be able to acquire a computer and involve it in their music-making.

This will make an increasingly audible change in the music of the world. No longer will we hear tapes in which Westerners invite non-Westerners into the recording studio and later manipulate recordings of them playing their instruments. It is too early to predict what the new music will sound like, but it is clear that the door is now open for non-Western practices to make a much more profound imprint on electronic music than as mere source material. I expect non-Western approaches to electronic music to be a source of much energy in the near future.

But we have not as yet caught up with the promises held out by the rapid drop in the price of computer hardware. Complete computer systems can be built for under US\$ 400, and the addition of an amplifier and speakers gives one the tools to start making computer music. But this assumes one has the necessary knowledge to build a system, locate and install good free software, and then run it. Many commercial interests work against such users, because they would prefer to make the computer cost several thousand dollars and the software several thousand more.

An essential element in the democratization of computing and of computer music is the nurturing of local knowledge bases. Certain forward-looking music educators are investing many hours trying to encourage students to build their own computers. I hope someday to see an international culture of

home-built computers and homemade computer music software. In the past, the affluent West has developed its software, stamped out millions of CDs, and sold them to whomever would buy them. In the future, I would like to see the centers of research and learning import software from the rest of the world.

Communities are necessary for knowledge to grow, especially with non-commercial operating systems such as Linux. (If one does not have friends also running Linux, installation and configuration can be a daunting task.) But I can imagine a future in which computer music expertise (including how to assemble a machine, install an operating system, and run software) is at least within a village or two of most people of the world.

The computer music community is similar to the Linux community in that it can grow among small groups with only occasional need for outside contact beyond what a modem can provide. To empower this, it is important that the software not come with its own cultural freight but that it adapt to whatever realities exist where the users live. This is exactly what typifies good software development in general.

This is not what typifies the products of today's software giants. For example, upon starting a commercial spreadsheet program, the user sees something very different from the empty sheet of virtual paper that Max offers. And since the page is not blank at the outset but is structured, the user will be constrained to move within the ordained structure. The whole business of the software hegemonists is to impose context in the form of proprietary file formats, operating system features, and other such constructs. Their concern is *not* in letting people do their own thing, and their products will always impede—rather than encourage—progress.

With that said, however, there is no reason people should not sell software. (As Joel Chadabe once told me, that is a very effective way of getting it in front of people who would not be able to get it otherwise.) We are extremely lucky to have a company like Cycling74 so that even if one lives in Madagascar, one can download a version of Max, run it, and send an electronic mail to someone if there is a problem. The existence of software companies is a

good thing, as long as they do not engage in predatory practices.

This is complemented by the other kind of empowerment that open-source software provides: one can customize it to a much greater extent than is possible with commercial software, simply owing to the absence of the constraints faced by any distributor of commercial software. The marketplace forces commercial software vendors to load their products with features. Each of these features is a potential barrier to portability—to one's ability to make the software run on a handheld computer or on the microprocessor in a refrigerator or a car. Generally, the more "bells and whistles" a piece of software has, the less easily it can be customized with features the software designer never considered. Even though good software writers can themselves dream of many things, the software user can always think of something else. Although we software writers try above all else to avoid imposing restrictions or obstacles, we never succeed entirely. It might be that some of the obstacles presented by today's software are so fundamental that we cannot even see them.

Computer Science or Computer Music?

Style is important in software (not so much the internal style of programming, but the style with which the software engages the user). We welcome software if its external style pleases us. Well-designed software enhances the workspace in the same way that well-designed furniture does, not only in functionality but also in the stylistic choices that enhance, and do not depress, the quality of our environment.

It is difficult to find rational arguments to explain Max's underlying stylistic tendencies toward black and white over color, blank pages over forms, and plasticity over hierarchical structures. Max is not about computer science but about computer music. Although many computer science results make their way into Max's design, that design does not bow to the computer science orthodoxy and its rigid dogma about software design and implementation. Many other software projects in computer

music have kept much closer to the computer science line; however, the most successful ones (Csound, for example) have primarily obeyed stylistic over computer science criteria.

If the success of Csound (and the as yet less proven success of Max) baffle and sometimes annoy the computer science community, the failing may come from a lack of understanding of the importance of style and even aesthetics in software design and implementation. Computer science has never found a metric for determining whether or not a computer program is fun to use.

We speak of "playing" a violin, not "working" it. Although music making entails a tremendous amount of work, it has to look like play, even to feel like play, if the musician is ever to survive the ordeals of practice and rehearsal (not to mention the privation of working for little or no pay). If using a computer program feels too much like "work," musicians will not—and should not be asked to—do it.

The computer should ideally feel in the musician's hands like a musical instrument, needing only to be tuned and then played. Has Max reached this ideal? Certainly not, and neither has any other piece of computer music software. I hope at least that, in the long term, it will prove to have been a step in a good direction.

References

- Abbott, C. 1980. "The 4CED Program." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 278–304.
- Anderson, D., and R. Kuivila. 1986. "A Model of Real-Time Computation for Computer Music." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 35–41.
- Boynton, L., et al. 1986. "Midi-Lisp: A Lisp-Based Music Programming Environment for the Macintosh." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 183–186.
- Danks, M., 1997. "Real-Time Image and Video Processing in GEM." *Proceedings of the International Com-*

-
- puter Music Conference*. San Francisco: International Computer Music Association, pp. 220–223.
- Dannenberg, R. 1984. "An On-Line Algorithm for Real-Time Accompaniment." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 193–198.
- Desain, P., and H. Honing. 1993. "The Mins of Max." *Computer Music Journal* 17(2):3–11.
- Favreau, E., et al. 1986. "Software Developments for the 4X Real-Time System." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 369–373.
- Lindemann, E., et al. 1991. "The Architecture of the IRCAM Music Workstation." *Computer Music Journal* 15(3):41–49.
- Mathews, M. V., and F. R. Moore. 1970. "GROOVE: A Program to Compose, Store, and Edit Functions of Time." *Communications of the ACM* 13(12):715–721.
- Mathews, M. V., and J. Pasquale. 1981. "RTSKED: A Scheduled Performance Language for the Crumar General Development System." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, p. 286.
- Puckette, M. S. 1984. "The 'm' Orchestra Language." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 17–20.
- Puckette, M. S. 1986. "Interprocess Communication and Timing in Real-Time Computer Music Performance." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 43–46.
- Puckette, M. S. 1991a. "Combining Event and Signal Processing in the Max Graphical Programming Environment." *Computer Music Journal* 15(3):68–77.
- Puckette, M. S. 1991b. "FTS: A Real-Time Monitor for Multiprocessor Music Synthesis." *Computer Music Journal* 15(3):58–67.
- Puckette, M. S. 1997. "Pure data." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 43–46.
- Vercoe, B. 1984. "The Synthetic Performer in the Context of Live Musical Performance." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, p. 185.
- Vercoe, B., and M. S. Puckette. 1985. "Synthetic Rehearsal: Training the Synthetic Performer." *Proceedings of the International Computer Music Conference*. San Francisco: International Computer Music Association, pp. 275–278.