Revista Vórtex | Vortex Music Journal | ISSN 2317-9937 | http://vortex.unespar.edu.br/
D.O.I.: https://doi.org/10.33871/23179937.2021.9.2.12

Pd and Audio Programming in the 21st Century

Eric Lyon

Virginia Tech | USA

Abstract: In celebration of the 25th anniversary of Pure Data, this essay discusses the development of audio programming up to the present, and considers the role that Pd can continue to play in the computer music of the future.

Keywords: Pure Data, sound design, computer music, audio programming.

Received on July 1st, 2021. Approved on: November 1st, 2021. Available online on: 30/11/2021. Guest Editor: Dr. Alexandre Torres Porres.

In celebrating the 25th anniversary of Pd, we discuss the role of audio programming in the practice of electronic music of the past, present, and future. Pd inhabits an important space within audio programming, being inherently oriented toward at least two kinds of programming – visual dataflow in the creation of Pd patches, and text-based procedural programming in the creation of externals in C code. The open-source code of Pd provides a valuable educational resource for audio programmers who wish to work at the procedural programming level. Thus, Pd remains a popular choice for composer/programmers, as well as composers who enjoy working with visual dataflow patching.

1. The Future of Music Software in 2001

20 years ago, I convened a symposium at Dartmouth College to consider the future of music software (LYON, 2002). Most of the audio programs created at that time disappeared within a few years. But a few audio software packages seemed more persistent, with the possibility of surviving into the future. To explore this possibility, the symposium invited the authors of Csound, Kyma, Max, Pd, and SuperCollider to discuss their ideas for the future of music software. 20 years later, all five of these systems continue to flourish. The results of the symposium were published in the Winter 2002 Computer Music Journal, volume 26(4).

In the opening address to the symposium, I stated:

> In the world of recorded music, which comprises most of what we listen to these days, the term "computer music" is redundant. The prevalence of the use of computers in today's music demands another distinction; at its outset computer music meant experimental music, carried out in laboratories and universities. This experimental work continues here and at many other institutions, but most of today's computer music is created in the field of entertainment, whether film music or the various technology-drenched genres of rap, techno, rock, and pop. (LYON, 2002, p. 13)

and

> The distinction between experimental music and what might be termed "normative music" – that is, music based on accepted stylistic norms – is mirrored in our software. On the normative side of software are utility programs such as mixers, sequencers, and reverberators. On the experimental side are the programs we discuss today. This software is open, extensible, and invariably used in ways unanticipated by its creators. While such

> software does not command a market on the scale of normative utility programs, it is arguably much more influential in the long run, as it facilitates the creation of music which today exists only in our collective imagination. And the experiments of today will lead invariably to the norms of tomorrow. (LYON, 2002, p. 13)

In retrospect, while my intuition in 2001 that the experimental audio programs Csound, Kyma, Max, Pd, and SuperCollider were gaining longevity has so far proved correct, the sharp distinction I drew between "normative" and experimental audio programs is no longer sustainable and represented an oversimplification even at the time. In 2001, there was already a long history of experimentalism in various forms of commercial and non-academic music that was not acknowledged in my address. The concept of experimentalism in electronic music requires a broader interpretation that includes artists and genres such as Sun Ra, The Beatles, Parliament Funkadelic, Disco, Detroit Techno, and many others. Within the practice of what is still referred to as "computer music," I think we will see increasing fluidity across the boundaries of commercial music vs. non-commercial experimental music. This will require greater fluidity across audio software programs, which previously maintained rather strict boundaries. We are already seeing hints of this fluidity, notably with the Max for Live model that turned Max/MSP into a library deeply integrated into the EDM-oriented popular software Ableton Live. Another example of such fluidity is the recent collaboration of Miller Puckette and Irwin, interfacing Pd with Ableton Live over the Internet (KIRN, 2021).

2. 21st Century Fluidity and Hybridity

In addition to inviting Miller Puckette, David Zicarelli, Carla Scaletti, Barry Vercoe, and James McCartney to the Dartmouth symposium, I also invited two audio programmers whose audio software was already on its way to obsolescence, but whose audio programming ideas were seminal: Max Mathews and Gareth Loy. Mathews's Music series is well-known. Perhaps less well known is Loy's CARL system, developed for the Computer Audio Research Lab at UC San Diego. Every CARL audio processing program, of which there were many, could be connected to any other CARL program using Unix pipes. This universal connectivity for audio signal is a powerful idea which has not been satisfactorily solved on modern systems. The JACK Audio Connection Kit, which has been around since 2002, does provide a model for interconnection of both audio and MIDI data. While

the JACK project is extremely important as an early realization of the idea of universal connectivity for digital audio, in practice several factors have prevented the software from being adopted with the same ubiquity as MIDI, including the complexity of using the JACK software, and the somewhat unreliable support for MacOS. This is not intended as a criticism of JACK, but rather an observation that in order to achieve ubiquity, most likely a broader-based coalition of programmer-designers would be required. The ReWire protocol jointly developed by Steinberg and Propellerheads (later Reason Studios) allowed for greater ease of interoperability between audio programs, routing both MIDI and audio data. ReWire was discontinued in 2020 in favor of a plugin app model. And ReWire was never a universal standard. Some applications implemented the protocol; many others did not. The BlackHole software provides a virtual audio driver that allows any audio application to pass audio through to other applications. BlackHole is easy to use, offers both stereo and 16-channel drivers, and can easily be recompiled for larger numbers of channels. This is a neat solution, but BlackHole is only available for MacOS.

Individual solutions for inter-application communication have proved robust, such as Max for Live, or coding applications as VST plugins, which can then be integrated into any other app that supports VST plugins. Max for Live provides a great model of seamless integration, but it only supports interaction between two audio programs that are owned by the same company. The VST app idea is a piecemeal solution. Theoretically this VST solution could adopted by every major audio program, which would solve the problem of universal connectivity for audio signal across different audio applications. Alternatively, a universal protocol might be designed, taking lessons learned from JACK, ReWire, BlackHole Virtual Audio Driver, and other inter-application audio communication programs. What is still needed is a reliable, easy-to-use protocol implemented with the same universality as MIDI was in the early 1980s. The potential benefits of such a "MIDI for audio" protocol have been evident from at least as far back as the early 1990s. Nevertheless, such a protocol has not yet emerged, and creating one would not be an easy task. It would require participation from most of the major audio software development companies. And it would need to accommodate multichannel signals, given the importance of spatial audio, and doubtless need to solve many other problems, not all of which might be obvious. But the benefit of such a protocol, should it emerge, would be that every audio app would be able to seamlessly integrate with every other one, such that audio apps could function as nodes on a net within the computer (and perhaps eventually on the

Internet as well). A plugin that works on one audio application would work on all other audio applications.

Although the proposed protocol would be difficult to implement, and currently there does not seem to be any great pressure for it to emerge, its utility might lie in a coming musical future that is increasingly hybrid. Even in the early versions of Max, the predecessor of Pd, Miller Puckette was aware of different kinds of users for the same software – composers, who could make pieces with Max but could not write C code; researchers who could write Max and other software systems for the composers to use; and realizers - power users of Max, who could also drop down into C to write externals for composers, when needed (PUCKETTE, 2020). If we consider computer musicians today to be anyone who uses a computer to make music, then the range of use cases across every available piece of audio software becomes a data explosion of possibilities. In such an environment, with every manner of audio program in use, often in very different ways, a maximally frictionless connection of audio across arbitrary audio programs would be useful to many different computer musicians and might accelerate the exploration of musical and technological hybrids in a positive feedback loop.

3. Audio Programming: Old Problems Fade, New Ones Take Their Place

From the viewpoint of academic computer music, which has traditionally been very interested in developing innovative approaches to creating sound in the digital domain, audio programming faces two important challenges in the 21st century: first, the major inventions of new sound synthesis and processing methods seem to have peaked by the mid-1990s; and second, there are so many powerful audio DSP plugins for every manner of synthesis and processing that one may ask if computer musicians still need to learn how to write procedural code. This is in sharp contrast to the situation of the 1980s when as a composer of computer music, to engage in timbre research on a computer, one needed to either write one's own code, or find someone else who could write the code for you. In the 2020s, coding is an option, but certainly not a requirement for computer musicians. So, why keep that skill going?

Coding is a problem-solving tool that by virtue of its need for precisely specified algorithms requires the composer/programmer to explicitly understand the musical problem under

investigation. This can bring new insights to the compositional process. The flexibility of code allows the composer to explore an idea as deeply as desired without the risk of running into structural limitations in audio signal chain paradigms characteristic of modern DAWs (e.g., synthesis algorithms with a fixed number of oscillators). Procedural coding is sometimes the most direct way to implement a complex audio algorithm. In addition to sound synthesis and processing, there are other computer music problems that are best solved in code, such as writing drivers for new hardware interfaces, extending networking capabilities, or working with machine learning algorithms. While it may be true that most of the major techniques for digital sound synthesis and processing have already been discovered, there is always room for refinement and further exploration. Even rich, expressive, established languages with large vocabularies acquire new words from time to time (Merriam-Webster added 520 new words to its English dictionary in January 2021[1]). If the preceding arguments are valid then Pd will continue to play an important role in the future of audio programming, given its free, open-source code base that facilitates both visual dataflow and procedural coding. In the next section, we'll discuss an example of using procedural coding to develop a processor for Pd that would be difficult or impossible to realize through visual dataflow programming alone.

4. A New FFTease External for Pd

Spectral processing is one problem domain for which procedural programming is currently the best level of abstraction for audio programming, given the large amount of data that must be processed, especially when bin-level state needs to be maintained across multiple FFT frames. These kinds of coding problems lend themselves well to the concise iterative loops and easily manipulated data structures afforded by text-based procedural languages such as C. The FFTease collection of externals for spectral processing was created by Christopher Penrose and me in 1999 (PENROSE, C.; LYON, Eric, 2000). This collection of externals was preceded by about a decade of research by the both of us into non-real-time spectral processing on Unix systems. Many non-real-time spectral processing algorithms that we created and used in our compositions in that early period never made it into FFTease, sometimes because they did not easily translate from non-real-time to real-time

[1] https://www.merriam-webster.com/words-at-play/new-words-in-the-dictionary (accessed July 15, 2021)

processing, but mostly because there were so many processors in our two Unix collections PVNation and POWERpv (LYON, 1996) that it was not practical to port all of them to FFTease. But all of that old code still exists, some of it might be worth porting, and at the very least, it could be suggestive of new processor models. A new FFTease of external will illustrate this point.

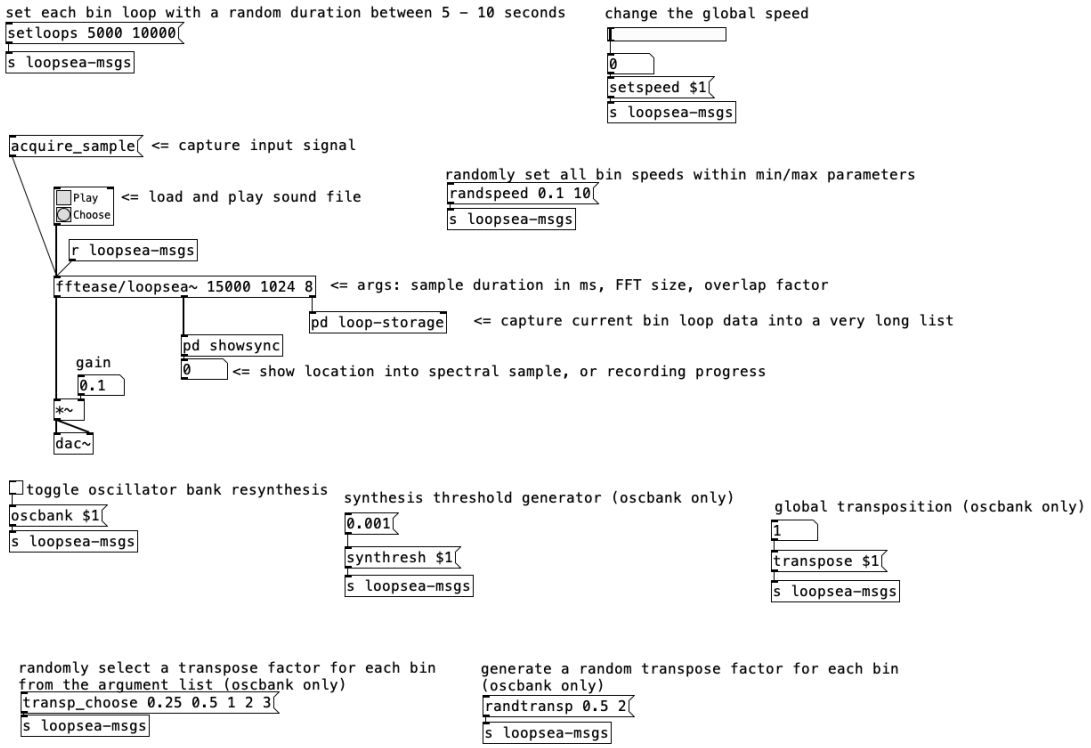
I recently implemented a new FFTease external in Pd, called *loopsea~*. This object was inspired by a Penrose processor, "aphrodite" from his Unix collection PVNation. The original "aphrodite" algorithm independently loops blocks of bins from a chunk of stored FFT frames. "Aphrodite" is an impressive algorithm, but it's not really set up for real-time use. Rather than try to reimplement "aphrodite" I took inspiration from the idea of looping bins from a stored series of FFT frames, and rather than looping blocks of bins, decided to loop each bin independently, creating a "sea of loops." With this idea in hand, the implementation in C was straightforward. I had previously created two FFTease objects, *residency~* and *resent~* which operate on a stored series of FFT frames with arbitrary and independent control over speed and frequency scaling. The architecture of *loopsea~* is similar, except that start and end frames for individual loops are stored for each FFT bin. The *loopsea~* object is instantiated with a fixed amount of spectral memory, defined in milliseconds. An input audio signal can be directly recorded into this spectral memory. Individual loops are then generated with the message "setloops" with two parameters, a minimum and a maximum loop size. If both parameters are identical, every loop will be the same length. Otherwise, each loop will be of randomly different size, and the individual bins will drift out of phase, with each bin loop having a potentially quite noticeable periodicity.

A few other refinements were implemented. First, the speed of the loops can be set globally with the "setspeed" method. A method called "randspeed" allows each loop speed to be randomly set, so that even if the loop durations are all identical, each loop will progress at a different speed. A further refinement was coded to allow for oscillator bank resynthesis. Given that most of the loops are likely to contain negligible amounts of noise, it is useful to have a synthesis threshold to eliminate bins that contain relatively little energy. This is controlled with the "synthresh" method. The "transpose" method allows for the entire resynthesized sound to be pitch-scaled without changing speed. The "randtransp" method generates a random transposition factor for each bin between minimum and maximum values provided by the user. Finally, the "transp_choose" method followed by a list of transposition options randomly assigns one of the transposition options to each bin. This

method can create interesting harmonizer-like effects. So that users can store a loop setting, with the loop point, transposition factor and speed factor for each bin, two methods are provided: "printloops" dumps all of this data from a list outlet on the object, and "readloops" loads all of this data back into a *loopsea~* object.

A typical usage of *loopsea~* is shown in Figure 1. As this example demonstrates, it is still possible to come up with DSP ideas that are relatively easy to implement in Pd with procedural coding but might otherwise be considerably more complex.

FIGURA 1 – A typical use of the FFTease loopsea~ external in Pd



Source: FFTease (Lyon, 2021)

5. Conclusion

In celebrating 25 years of Pd, we considered the creative and research space in which Pd emerged and speculated about developments beyond that origin for audio programming, and how Pd might fit into future developments of computer music, broadly construed. In that developing future, new audio programming contexts are emerging for which Pd should continue to serve as an excellent platform for an increasing range of computer music scenarios.

REFERENCES

Future of Coding podcast interview with Miller Puckette, 2020-05-12. <https://futureofcoding.org/episodes/047.html> (accessed July 15, 2021)

KIRN, Peter. *Miller Puckette, Creator of Max and Pd on How He's Patching His Way to Remote Collaboration*. https://cdm.link/2021/03/miller-puckette-creator-of-max-and-pd-on-how-hes-patching-his-way-to-remote-collaboration/ (Accessed July 19, 2021.)

LYON, Eric. Dartmouth Symposium on the Future of Computer Music Software: A Panel Discussion. *Computer Music Journal*, 26:4, pp. 13-30, 2002.

LYON, Eric. POWERpv: A Suite of Sound Processors. In: *Proceedings of the International Computer Music Conference*, pp. 285-286. ICMA, 1996.

PENROSE, C.; LYON, Eric. FFTease: A Collection of Spectral Signal Processors for Max/MSP. In: *Proceedings of the International Computer Music Conference*, pp. 496-498. ICMA, 2000.

ABOUT THE AUTHOR

Eric Lyon is a composer, audio programmer, curator, and educator whose work focuses on chaos music, oracular sound processing, post-hierarchies, and spatial orchestration for high-density loudspeaker arrays. Lyon's creative work has been recognized with a ZKM Giga-Hertz prize, MUSLAB award, the League ISCM World Music Days competition, and a Guggenheim Fellowship. Lyon teaches in the School of Performing Arts at Virginia Tech, and is a Faculty Fellow at the Institute for Creativity, Arts, and Technology. E-mail: ericlyon@vt.edu