Revista Vórtex | Vortex Music Journal | ISSN 2317–9937 | http://vortex.unespar.edu.br/
D.O.I.: https://doi.org/10.33871/23179937.2021.9.2.11

The Null Piece and Reality Check

Miller Puckette

University of California, San Diego | USA

Abstract: Two frameworks are described, within the larger framework of Pure Data, that aim to facilitate the creation and preservation of electronic music that is performed live in real time. These are not part of Pd itself, because they are more directly tailored to a specific application space than Pd should be. They are nonetheless designed, like Pd, to minimize any unnecessary stylistic imposition on creators of electronic music. The Null Piece is a starting point for building live performance patches. It provides basic audio routing and parameter control while leaving other choices as free as possible. Reality Check is an attempt to aid in the long-term maintenance of a musical realization by verifying whether a piece still runs as intended despite the inevitable evolution of hardware and software.

Keywords: Pure Data, Software engineering, Electronic music.

Received on July 1st, 2021. Approved on: September 1st, 2021. Available online on: 30/11/2021. Guest Editor: Dr. Alexandre Torres Porres. Creative Commons CC-BY-NC.

Although Pure Data is now many things to many people, one central design goal has been to support the creation of long-lasting interactive works of electronic art, particularly music. This is perhaps the main reason that Pd has evolved so slowly over the years (much more so than many other such platforms). This is also why Pd's core has remained relatively small - small enough, I hope, to allow it to be maintained indefinitely in its present form.

A related design goal is to show what I call the "blank sheet of paper" quality: that Pd should avoid encouraging a particular work flow or design paradigm for creating artworks. This has complicated implications that I've written about elsewhere. The one that concerns us here is that a subset of users could easily emerge that find themselves solving the same problems. This situation sometimes calls for the creation of added functionalities, not a part of Pd's core, that can be used by these user subsets.

One such functionality might be the creation of music for stage performance, in which a computer has some number of audio inputs and outputs that are connected to a fixed network of audio analysis, synthesis, and processing modules. Very commonly desired capabilities include metering of inputs and outputs, presentation of test inputs from soundfiles (or just simple test tones) to the audio modules, synchronous recording of inputs and outputs during rehearsals or live performances, setting and displaying numerical parameters, controlling gains, and saving and recalling presets, either individually or in a sequence. I've helped composers develop pieces along these lines since 1985, and over time have arrived at a template I always start from, that I call the "Null Piece".

A long-standing and complementary need has been to maintain Pd patches for stage performance so that they can remain usable over periods of time measured in decades (and perhaps even longer). This is a hard problem for many reasons, but one fairly simple tool can help a great deal: a facility for simulating a stage performance to verify whether a given patch produces the historically correct output. To this end I've developed a package called "Reality Check" that uses the pd~ object to run a realization of a musical piece in a sub-instance of Pd and check the correctness of its output.

Neither of the two projects described here is especially novel. The Null Piece has an important antecedent in Integra Live, a platform built in support of the Integra Project (Rudi 2011). The Null Piece is quite modest in comparison. Its design trades off modularity and scaffolding in favor of easy extensibility and extremely low maintenance load.

There is a rich history of literature about, and prior efforts to address, the short lifespans of electronic music realizations (Vincent 2010; Bonardi 2015; Akkermann 2017; Lindemann 2020). The skeptical reader could predict that Reality Check will soon join the ash heap that houses so many earlier preservation projects. All I can propose here is to learn what we can from earlier experience and keep at it.

1. The Null Piece

The Null Piece comes in two versions, "0" and "1". The "0" version is intended for pieces with any structure (fixed-score, open form, improvised, algorithmic, whatever). The "1" version contains everything in "0", plus a facility for linear sequencing intended for through-written musical scores.

The design of the Null Piece assumes that most creators of musical realizations in Pd will want at least certain basic functions:

- audio input and output, with gain controls and level metering
- using a soundfile or a test tone as input to the patch
- recording the input and/or output of the patch to a soundfile
- setting and viewing numerical parameters by hand or via Pd messages
- ramping numerical parameters up and down over time intervals
- controlling audio gain at various stages in the patch
- saving the current values of parameters to a text buffer or file
- a reset mechanism to put all parameters into a fixed initial state

The "1" version also provides a sequencer that sends Pd messages, single-stepped and/or timed, to the rest of the patch.

Although these features all might seem to be rather easy to code, the precise way they are made can greatly affect how well the patch works. The particular design choices presented here have evolved substantially over the past thirty years, and, while they are nothing earthshaking, it is worth going into the reasoning behind them.

These features are intentionally not part of the design of Pd itself. Software programs such as

DAWs, oriented toward specific musical tasks, tend to have these functions already built in. Pd makes it possible to make media applications that do not even hew to an audio sample clock. In such applications some or even all of the above desiderata might be superfluous, and might even "nudge" artists in directions that don't directly serve their aims. Although providing affordances is always a positive, one needn't bundle every possible affordance together in a single package.

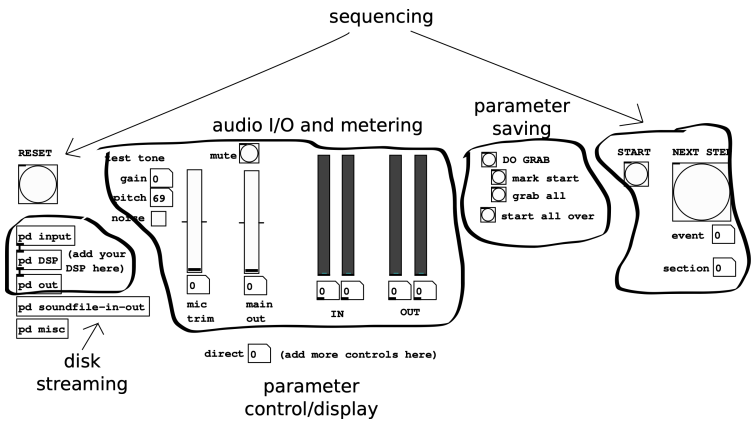
There is no assertion here that the Null Piece is made in the one and only "correct" style for building music performance realizations, but only that it is a pretty good style from which individual musicians will probably want to adapt their own personal one.

The Null Piece is published as part of the Pd Repertory Project, on the URL msp.ucsd.edu/pdrp.

1.1. Overview

Figure 1 shows the screen layout of the 1 version. (The 0 version is the same except that the only sequencing control provided is the "reset" button.) The entire "piece" consists of one audio transformation, which sends channel 1 of the input to two channels of output. One parameter, "direct", controls the gain. The layout is considered as a workable template for a performance patch, showing those elements which are to be seen and/or controlled during a rehearsal or performance and hiding most of the rest. A patch for a real piece might look much the same, but with more parameter controls.

FIGURE 1 – Pure Data patch for the Null Piece.



Source: the author

Much of the design is concerned with management of parameters. These come in two types: manual and automatable. Manual controls might be used to adjust for a particular acoustic situation without saving anything for future performances. They could include the input and output signal gain, test tone levels, and controls for the soundfile reader and writer; these are thought of as external to the piece but might be relevant to a particular rehearsal goal, speaker placement, and audience mass. They might also allow for live control of a mix using an external controller.

The display and control of manual controls is aided by the setctl abstraction (which, along with four other abstractions and one supporting text file, are placed in a sub-directory named "lib"). Whenever a message is sent to, for instance, "test-tone", the setctl abstraction catches the message, converts it to a "set" message, and sends it to any object bound to "test-tone-set". The test tone gain control (a number box) has the send signal "test-tone" and the receive signal "test-tone-set". So the test tone gain is both set and displayed by the control.

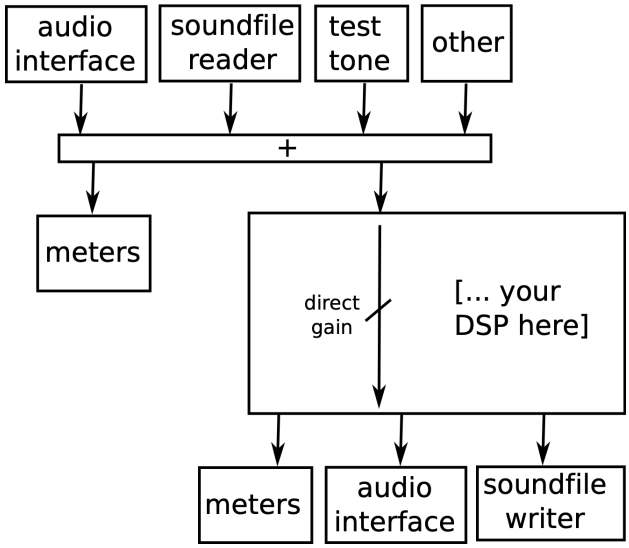
Controls meant to be sequenced or otherwise automated can use a more general abstraction named "genctl". This abstraction contains logic to display the progress of ramps in which the parameter is sent a pair of numbers, the second one giving a ramp time in milliseconds. Whatever object actually enacts the ramp might do so using a "line" object of arbitrary grain, or one of its two signal variants, line~ or vline. Using the same "set" mechanism as the setctl abstraction, genctl generates a separate ramp updated at a viewable rate. The genctl abstraction also interacts with a parameter-grabbing mechanism to aid in making presets or sequencer steps, described in the "parameter grabbing" section below.

1.2. Audio signal path

The audio signal path is shown in figure 2. The input of the patch may be taken from Pd's audio input (an audio interface or virtual audio device), and/or a test tone, a streaming soundfile reader, or via an auxiliary catch object (in case you want to use another patch such as Quacktrip as input, for example). Audio inputs and outputs have gain controls and level meters, which are post-fader. The input level control sets only the gain of the external audio interface, not the additional inputs, so that you can suppress the microphone input while using a different one. Both the input and output level use 4th-power (quartic) curves, with "50" set to unity; this scale is described under

"gain controls" below.

FIGURE 2 – Audio signal pathways in the Null Piece.



Source: the author

The test tone can be set either to a sinusoid or low-pass-filtered white noise (in which case the "pitch" control sets the cutoff frequency of the filter). A subpatch ("pd soundfile-in-out") contains a GUI for choosing, opening, and playing a portion of a soundfile through the patch, optionally looping it.

The null patch contains only one audio "process" (a direct gain from input 1 to outputs 1 and 2) with the understanding that you will add your own modules, perhaps keeping the "direct" as a convenient test of the audio path. You can also add more input and/or output channels. Output may be streamed back to disk, at the same gain as sent to the audio output. By default only the two outputs are streamed, but here again you can elect to add other channels which may be additional outputs, any or all inputs, and/or intermediate signals for testing. This is all done by editing the "input" and/or "output" subpatches.

1.3. Gain controls

Most patches are likely to contain numerous gain controls as various inputs are sent to various processing elements whose outputs are sent to yet other processing elements and/or output channels.

In Pd, gain is applied to an audio signal (consisting of one or more channels) by generating a continuously variable "gain" audio signal and multiplying it by each of the channels. An abstraction. amp4~, is provided to generate a gain signal. The design of amp4~ is intended to satisfy the needs arising in most gain-control situations without being any more complicated than necessary.

The amp4~ object is invoked with a parameter name, through which it receives messages that can come from anywhere else in the patch (and which can be shown on a corresponding number box via the setctl or genctl abstraction). A pair of values specifies a ramp, with the second value giving a ramp time in milliseconds. If no ramp time is given the ramp defaults to 20 milliseconds (this may be overridden using an optional argument). If a ramp time of zero is desired despite a non-zero default ramp time, it may be specified explicitly in a message.

Two scales are implied by this design: first, a scale whereby a numerical parameter specifies a non-time-varying gain; and a second scale that determines how ramps behave over time. These two scales have distinct criteria.

The input scale is chosen so that: (1) values stay comfortably in the range 0-100, with 50 corresponding to unit gain; (2) there is at least 20 dB of headroom; i.e., a parameter value of 100 specifies a gain of +20 dB or more; (3) an input of 0 results in a gain of 0; (4) visible changes of level correspond to audible changes in loudness; and (5) for values around 50, a parameter change of one unit make well under one dB of change, so that fine adjustments are possible using integer values.

A decibel scale would be wildly inappropriate; even if we allow that 100 dB below the maximum gain is inaudible (probably not true if we have 20-ish dB of headroom as well), the middle of the scale will be nearly as quiet as the bottom and almost all the useful "throw" will lie within a tiny range of parameter values. A linear scale won't work either, since if 50 corresponds to unit gain 100 will only reach +6 dB. The choice made here is a fourth-power curve, with 50 corresponding to unity. This gives a true 0 for an input of 0 and 16 for an input of 100, which is about 24 dB of headroom. In this scale, the difference between 50 to 51 is only about a 2/3 dB, so the control is probably fine-grained enough for most uses. (Non-integer values can be used if a finer grain is needed).

If incoming MIDI "control change" messages are used as gain inputs, they can be normalized to the range 0-100, or else they may be left unmodified to allow the control to reach 127 (for a gain of +32 dB).

The output of amp4~ ramps over time, both to avoid signal discontinuities and to provide fade-ins and fade-outs. The design of a fade-in and fade-out shape is, again, chosen to be a good compromise for most use cases. Fades are roughly linear in sones (that is, roughly proportional to amplitude raised to the 0.6th power). The most audible consequence of a choice of a fade shape occurs when fading between zero and a non-zero value. In this situation we posit that, halfway through the fade, it should be 1/2 as many sones as at the high end, that is, at relative gain of about 0.315. For simplicity and efficiency we simply use a second-power power law that achieves 0.25 instead (as if a sone were the 0.5th power of amplitude instead of the ISO standard 0.6th).

1.4. Startup, initialization, and reset

When the Null Piece patch is first opened, a "bang" message is sent to the global symbol "init". This is used, for example, to set the input trim and overall output gain to 50 (for unity gain). Twenty milliseconds later (to give time for all "init" level settings to take effect), a bang is sent to the "reset" symbol. This is intended to set all automated parameters to their initial value. Parameters that are managed by genctl objects catch this reset message and are ramped to 0 by default. An optional argument to genctl can override this to reset to a different value.

The reset bang message is also sent whenever the "stop" or "start" button (in version 1) is pressed.

1.5. Parameter grabbing

A parameter grabbing mechanism is provided so that you can save the values of any numerical parameters controlled by genctl objects, which can be put into message boxes and/or text or qlist objects. The mechanism has a user interface consisting of four buttons, the first of which is "do grab". Each time you click that, all the genctl-controlled parameters that have changed since the last grab or patch reset are printed, both to the Pd window and to a text window in "qlist" format. By pressing "do grab" at various points while making parameter changes, you can get an incremental sequence of messages that will recreate the parameter changes without unnecessary repetitions.

In case you really want it you can press "grab all" to get a complete snapshot of all parameter

values, even including ones that are at their reset value. You can also say "mark start" to direct the parameter-saving mechanism to report changes only from the current setting. For example, if you are using the "1" patch and building a qlist, you can advance to any spot in the qlist and hit "mark start" to record incremental changes from there onward. Finally, there is a button to clear the qlist text buffer and its event counter, in case you are ready to build a qlist from scratch.

Certain parameters might be sometimes set explicitly in a message box or qlist, and at other times controlled by an automaton. In this situation, changes made by the automaton should not be saved into a qlist or message box, since the automaton will presumably be running again when the current state is restored. For this reason, the genctl abstraction catches a corresponding message to suppress grabbing when its parameter is being changed automatically.

1.6. Sequencing

The "1" version of the Null Piece has a sequencer based on Pd's qlist object. The sequencer considers the piece as having one "section", although you can add other sections and either make them automatically start each other in turn or treat them as separate movements. Each section has a corresponding qlist file. The one section in the Null Piece itself is stored in the "score" subdirectory. The contents of the qlist are as shown:

```
0 0 print section 1, event 1;
0 1 print event 2;
direct 50;
0 2 print event 3 - end;
500 direct 0;
```

There are three events. The first one is played when the "start" button is pressed; it is intended to perform any setup actions such as reading audio samples into wavetables. After pressing "start", you can activate the following events in order using the "next" button. Each event in the qlist should start with a line such as "0 1" for event 1. That line may also contain a first message (in this example, a helpful message to "print" which appears on the Pd window).

Lines starting with two numbers (zero and an event number) mark the beginning of the next event; the event numbers should be in numerical order. Other lines that begin with a single number specify delays before the message to be sent. These delays are cumulative, and by default are in milliseconds. (You can edit the patch to change the time units or even to set a time-varying tempo.) The messages themselves may start and stop sub-sequences in case a more flexible order of execution and/or multiple tempi are desired.

2. Reality Check

The first part of this paper considers a tool for quickly starting to develop a new piece of live electronic music. We now turn to a tool whose purpose is to address problems at the other end of the process: how to keep a piece of electronic music running once it's been made.

Many pieces of new music, both electronic and purely acoustic, never get a second performance after their premiere. Only a lucky few of them are deemed worth continuing to perform over time. Pieces that use electronic supports are even less likely to receive repeat performances than acoustic ones, both because of the fragility of real-time electronic systems in general, and also because of the rapid evolution of the technologies on which they depend.

A new tool called Reality Check, implemented in Pd, is intended to fill this need, at least for pieces that are realized in Pd and whose inputs and outputs are primarily audio.

In at least one respect, the preservation problem is easier than it might sound. Although it is widely bemoaned that nothing in computer technology is stable, there is in fact a growing body of file formats, APIs, and programming languages that are verifiably stable. The affordances of the computer seem to be in a state of perpetual explosion with the emergence each year of new technologies. For any such technology (for instance, 3-D rendering), there is a period of experimentation. If the technology proves widely useful this is then followed by a period of standardization. Luckily for computer musicians, most of the technologies needed for live electronic music performance (with the notable exception of electronic instrument interfaces) are now effectively standardized.

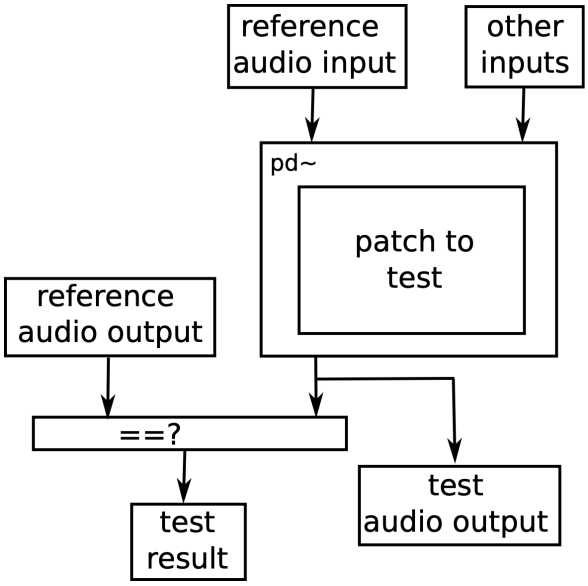
For it to be possible to preserve a piece exactly, the software that realizes the piece must be deterministic. Almost all of Pd is deterministic, but there are a few exceptions such as the realtime

and cputime objects, as well as any operation that reads a file that is not archived as part of the patch. Most of these situations are easy to avoid. (However, there could be other, unknown sources of indeterminism that could cause trouble at some unknown point in the future. If these exist, Reality Check could at least detect them.)

2.1. Design of the patch

Reality Check consists of a Pd patch that runs a music realization in a separate instance of Pd, in which inputs can be precisely controlled and outputs can be checked against a pre-computed reference output. Figure 3 shows a block diagram of the reality-check patch.

FIGURE 3 – Reality check block diagram.



Source: the author

The reality-check patch is used in two phases: preparation and verification. To prepare a patch, you start the reality-check patch and feed it the pathname of the patch you wish to prepare. This will be run by the reality-check patch in a sub-instance of Pd using the pd~ object. You also feed the reality-check patch the audio inputs you wish to present to the piece, along with a file containing time-tagged messages that capture any user interface gestures and control inputs your piece should receive. (You might have to add receive objects to your patch to enable it to receive virtual inputs

corresponding to incoming MIDI messages or whatnot.) In the preparation phase, the reality check patch simply passes all this input to the sub-process and records the resulting audio output. The creator of the piece should be consulted to verify that the recorded output can indeed be considered correct.

At any later time you can then run the reality-check patch again to verify that the patch still produces the same output as it did before. In this mode the reality-check patch subtracts its reference output from the patch's new output, and if the two differ by more than a fixed tolerance (one part in a million by default), the difference is reported and the test is considered as having failed. Otherwise the test has succeeded and you have verified that the patch works as it did before, at least when given those particular test inputs.

Once the testing regime for a particular piece has been set up, you can run it whenever either (1) the patch for the piece is modified; (2) a new version of Pd appears; (3) a new version of some dependency such as an external library is updated; or (4) new computing hardware is introduced. In some situations a software upgrade might occur, perhaps to a system library, without your even knowing how it could affect the output of your patch. For this reason it is advisable to run the test on a regular schedule in addition to any time a change is known to have occurred.

It is wise always to have implementations of any piece on at least two computing platforms (for instance, MacOS and linux), both to get early warning if something operating-system-dependent has crept into the patch, and also so that, when a failure occurs, it is easy to trace the source of the unwanted change.

2.2. What to do when Reality Check reports a failure

When a failure is reported, that is, when the patch no longer produces the same output as it once did, it is necessary to track down the source of the change. In the easiest case this is done by going back to a working version and incrementally reintroducing whatever changes have been made until a culprit can be isolated. (If a working version is not available this will be much harder to do and might come down to guesswork.) If the problem arises because the patch itself has changed it is time to revert to the older version; presumably the patch has changed for some good reason but the changes should be reworked so that the test can pass again.

If the problem is that something in Pure Data has changed incompatibly this is a bug in Pd and should be reported quickly. Pd is also expected to hide changes in whatever operating system calls and other APIs it makes use of, and again Pd's own implementation would then need work. (One of the happy side effects of the Reality Check project is that it will act as a verification suite for Pd itself that can give early warning when a change to Pd has caused an incompatibility. Similarly, an update in Reality Check itself can cause problems; this is regarded as a compatibility failure in Reality Check that must be corrected).

The situation regarding non-standard dynamic libraries loaded by a patch is more delicate. For any piece to be maintainable, all such "externs" must be provided with source code and with the necessary licensing to permit updating and recompilation. This is often neglected in the music production process, particularly when under time pressure or if making use of experimental techniques. Simply advising composers not to use experimental techniques is not a good answer because that is sometimes at the very heart of a musical project. A middle way must be negotiated in which the composer and whoever is doing the research agree on a plan to create a stable, maintainable implementation for use in the piece. More on these issues can be found in (Lindemann 2020).

2.3. Managing evolution

Often the creator of a piece will want to revise it. This is easily managed as long as the revised piece is then reincorporated into the reality-check framework, with a new patch and new reference output (and perhaps other changes as well). A more complicated situation arises when some aspect of the technology is improved. For example, analysis techniques such as pitch detection and score following are constantly being improved upon. For such improvements to be incorporated into an existing piece it is necessary for the creator to be invited back to verify that the new outputs still represent the piece correctly. If the creator is not available it might be desirable to maintain two distinct versions of the piece, one "period" implementation that reproduces the original audio outputs, plus an updated one that reflects whatever improvements seem worth introducing. This situation closely mirrors that of classical music where there can be trade-offs between authenticity and modernity in the choice of instruments.

2.4. Current status

An implementation of Reality Check is available on the URL http://msp.ucsd.edu/tools/reality, along with a small but growing collection of reality-checked pieces: a song from Philippe Manoury's *en Echo*, Kerry Hagan's *of pulses and times*, and, still in preparation, Rand Steiger's *Ecosphere*. These were chosen to incorporate a wide range of different patching styles, and together they demonstrate that the Reality Check framework is able to accommodate them. (Also, *Ecosphere* lasts nearly 30 minutes and boasts a channel count of 14 channels in and 6 out, so it verifies the practicality of using Reality Check on large pieces).

Although these pieces require some user interface actions on the part of the person running the patch, they do not exercise any input devices other than audio inputs and GUI actions resulting in Pd messages (such as to advance a patch through a piece). Inputs from MIDI, OSC, and various hardware devices can also be incorporated but only by recording all the inputs with time stamps and making a monster timed list of messages that Reality Check sends (exactly repeatably) to the patch. This will likely be an unwieldy process that will merit adding some new functionality.

Musical output other than audio (such as MIDI messages) are not handled at all, and neither is video input or output.

3. Conclusion

The Null Piece is by no means a definitive or unique way to start in on a music production, and Reality Check is not at all the last word in electronic music preservation. They both make steps toward adapting Pd to the specific purpose of creating electronic music that is performed live. In this sense they are directed at a more narrowly defined set of tasks than is Pd itself, but in exchange they offer a measure of labor saving over what Pd alone can offer. They nonetheless remain fairly abstract, in order to support the live performance of as wide a range of possible musical styles and approaches as can be managed, while at least mostly using audio computations, inputs, and outputs.

REFERENCES

AKKERMAN, Miriam. 'This hardware is now obsolete.' Marc-André Dalbavie's Diadèmes. Lansing, Michigan: International Computer Music Conference Proceedings. 2017.

BONARDI. Alain. Rejouer une oeuvre avec l'\électronique temps réel: enjeux informatiques et musicologiques. Paris: Presses de l'Université Paris-Sorbonne, Genèses Musicales, pp.239-254, 2015.

LINDEMAN, Eric; PUCKETTE, Miller; MANOURY, Philippe. Proposal: Keeping Real-Time Electronic Music Alive. Available on http://msp.ucsd.edu/tools/reality/, 2020.

RUDI, Jøran; BULLOCK, James. The Integra Project. New York: *Proceedings of the Electroacoustic Music Studies Conference*, 2011.

VINCENT, Antoine; BONARDI, Alain; BARTHÉLEMY, Jérôme. Pourrons-nous préserver la musique avec dispositif électronique? *Sciences humaines et patrimoine numérique*. Paris: Archives Ouvertes (hal-01156710). 2010.

ABOUT THE AUTHOR

Miller Puckette is known as the creator of the Max and Pure Data real-time computer music software environments. As an MIT student he won the Putnam mathematics competition in 1979. He received a PhD from Harvard University in 1986. He was a researcher at the MIT Media lab from its inception until 1986, then at IRCAM (Paris, France), and is now distinguished professor of music at the University of California, San Diego. He has been a visiting professor at Columbia University and the Technical University of Berlin, and has received two honorary degrees and the SEAMUS award. Puckette has performed widely in venues including Centre Acanthes, Carnegie Hall, the Pulitzer Arts Foundation, the Ojai Music festival, and a cistern beneath Guanajuato, Mexico. His installation, *Four Sound Portraits*, was shown at the 2016 Kochi-Muziris Biennale. E-mail: msp@ucsd.edu