NOTES ON MULTITIMBRALITY AND TEMPERAMENT

Albert Gräf

IKM, Musicology, Computer Music Working Group
Johannes Gutenberg University (JGU) Mainz, Germany
aggraef@gmail.com

ABSTRACT

Grame has been offering support for creating polyphonic instruments from (monophonic) Faust synthesizers for a while now. Unfortunately it lacks *multitimbrality*, i.e., the capability to maintain separate control data for different MIDI channels. JGU has led the way there, by providing proper multitimbrality in its LV2 and VST architectures, as well as a few other interesting features. On the other hand, Grame's MIDI implementation also offers some substantial advantages which the JGU architectures lack. This article discusses the most important special features of the JGU architectures, so that they may hopefully be incorporated into Grame's MIDI implementation in the future and thereby become available to a wider range of Faust architectures.

1. INTRODUCTION

A classic application for dsp programming systems are polyphonic synthesizers, i.e., dsp algorithms turning MIDI note and control data into audio signals. This is typically implemented by taking monophonic sound generators and running them in parallel. To make this work, a *voice allocation* algorithm has to be provided which manages a pool of dsp instances (called *voices*). This algorithm allocates dsp instances for simultaneously sounding MIDI notes (usually in a round-robin fashion) and deals with a bounded number of voices by employing *voice stealing* (i.e., dropping a note, typically the longest-playing one) if the number of notes to be played would exceed the number of available dsp instances.

Of course, these issues have all been well understood at least since the dawn of digital synthesizer technology in the 1970s. As far as Faust is concerned, at JGU we've been leading the way there by adding fairly comprehensive MIDI and polyphony support to our architectures around 2012 [1]. Grame recently followed suit by extending the Faust core with their own MIDI and polyphony support [2].

It should be noted that these facilities have always been optional. Even if they are not available directly in the Faust architecture that you use, you should be able to implement voice allocation in a suitable host environment such as Pd and Max instead. This is much less convenient, though, thus providing good MIDI support in Faust itself is an important feature.

The Grame and JGU MIDI implementations each have their own strengths and weaknesses. One big advantage of the Grame implementation is that it is available in Faust's core and can easily be added to *any* architecture, by simply including the corresponding header files and running a few initializations in the architecture code (details can be found on the Faust website). This makes adding MIDI support to existing architectures a walk in the park. Other major advantages of the Grame implementation are sample-accurate timing of control events on input, MIDI sync, support for all types of MIDI messages (including channel and key pressure a.k.a. "aftertouch") and more comprehensive MIDI *output* for all message types. (Most of these features are on the road-map for the JGU architectures as well, but have not been implemented yet.)

On the other hand, one major shortcoming of the Grame implementation is its lack of proper *multitimbrality*, i.e., the capability to maintain separate control data for different MIDI channels.[1] Consequently, a synthesizer created with Grame's facilities will *not* handle multi-channel MIDI data as one might expect. E.g., when processing a volume change (cc 7) on MIDI channel 3, say, the volume of *all* voices will be changed, no matter what their MIDI channels are. This may be adequate for simple real-time MIDI input, but is at odds with the MIDI standard which says that "Control Change messages, like other MIDI Channel messages, should only affect the Channel number indicated in the status byte" [3, p. 5]. And it causes trouble with sequenced MIDI data, the kind you find in most type 1 standard MIDI files, i.e., virtually every MIDI file you can download from the web.

In order to properly implement multitimbrality, the JGU implementation is necessarily more complicated, which may be the reason why Grame chose not to adopt it. But it is not all that difficult to understand either. Therefore in this article we make an attempt to clarify how it works, so that we may hopefully integrate it into Grame's polyphony implementation in the future and thereby make it available to a wider range of Faust architectures. While we're at it, we also briefly review another important feature offered by the JGU architectures which the Grame implementation lacks right now, namely support for the MIDI tuning standard (MTS). For applications dealing with micro-tonality and historic temperaments, this is a rather essential capability, so it would be good to have this in the Faust core as well.

2. POLYPHONY IN FAUST

To be able to be used as an instrument, a Faust dsp must follow a few simple conventions.[2] First, it must provide three special so-called *voice controls* named "freq", "gain" and "gate", which denote a note's frequency, amplification and on-off signal, and are in 1–1 correspondence with the parameters of a MIDI note (note number, velocity and on/off status). Secondly, the architecture must be informed that it's supposed to go through the necessary

[1] The term "multitimbral" is often used somewhat loosely to denote any MIDI synthesizer "capable of producing two or more different instrument sounds simultaneously" [3, p. 10]. However, such an instrument *will* normally have to provide separate control data for different MIDI channels, in all but the most minimalist implementations. Therefore (and for want of a better catchphrase), we use "multitimbral" as a moniker for this more specific capability throughout the paper.

[2] This design is nothing to write home about, it was born out of practical requirements when we first designed an interface for Faust-based instruments in Pd-Faust [4] and the JGU architectures [1], and somehow it stuck. Please check [5, Section 9] for the technical details.

incantations to turn the dsp into an instrument. Most architectures which support this will nowadays understand the `nvoices` declaration which tells it the number of voices that should be available. If the value of `nvoices` is zero, or the `nvoices` declaration is absent then the program is to be taken as is, i.e., it will be treated as an ordinary audio effect.

For instance, here is how we'd employ these conventions to turn a simple sine generator into an instrument with sixteen voices and ADSR envelop.

```
declare nvoices "16";
import("stdfaust.lib");

freq = hslider("freq", 440, 0, 10000, 0.01);
gain = hslider("gain", 0.3, 0, 1, 0.01);
gate = button("gate");

vol = hslider("vol [midi:ctrl 7]", 0.2, 0, 1,
              0.01) : si.smooth(0.99);

process = os.osc(freq)*
  en.adsr(0.01,0.1,gain,0.1,gate)*vol;
```

Note the voice (`freq`, `gain` and `gate`) controls; the latter two are used as the sustain level and the trigger signal of the envelop, respectively. To illustrate the use of MIDI controllers, we also added a "master volume" control which applies to all sounding voices. The meta-data `[midi:ctrl 7]` assigns that control to the customary MIDI controller #7.

The one big advantage of the approach sketched out above is its simplicity. The developer designing an instrument only has to worry about generating the sound of a single note. The necessary infrastructure to turn the monophonic dsp into a polyphonic instrument will then be provided by the architecture. However, there are situations in which this simple method falls short. E.g., an instrument requiring tight interactions between the different voices (such as a physical modeling synth which simulates sympathetic resonances between strings) will most likely require a more elaborate implementation than what the available polyphonic architectures currently provide.

3. MULTITIMBRALITY

In order to properly handle multi-channel MIDI data, the compiled Faust dsp has to allocate and maintain storage for control variables not only *per voice* (that's what code generated by the Faust compiler does anyway), but also *per MIDI channel.* On a conceptual level, this requires the collection of data structures depicted in the schematic diagram on the right (Fig. 1).

Let us take the instrument from the previous section as an example. When a MIDI controller event such as cc 7 127 (which sets the master volume control to its maximum value) arrives on MIDI channel 1, say, the new control value (1 in this case) will first have to be recorded in the parameter block for that MIDI channel. Then the architecture has to consult the voice allocation table in order to determine all voices which are currently active playing notes from MIDI channel 1, and update the value in the corresponding parameter blocks accordingly (this can be made efficient by implementing the voice allocation table as some kind of bidirectional multi-map). Subsequently, when a new voice is allocated for a MIDI note event on MIDI channel 1, the data from the parameter block for MIDI channel 1 will be copied over to initialize the newly allocated voice with the proper control values.

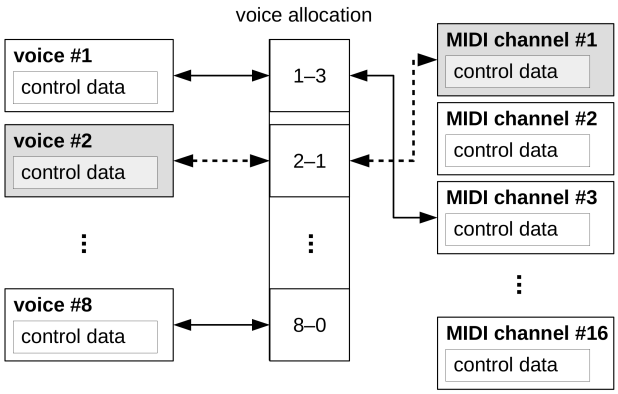

Figure 1: Multi-channel control data.

In principle, MIDI *output* for passive Faust controls can be treated in the same fashion, by reversing the flow of control data so that it goes from the voices to the MIDI channel data, but this has not been implemented in the JGU architectures yet. There's also one complication which arises for MIDI output only, namely how control data from different voices on the same MIDI channel should be aggregated. Sum, average, median, minimum and maximum are some obvious choices there, but that's up to the application, so we'd probably want to make it configurable using per-control meta-data.

Other MIDI data, such as pitch bend and aftertouch, will have to be handled in an analogous fashion. Same goes for the tuning tables which are needed to map MIDI note numbers to frequencies. These aren't implemented in the Grame architectures which uses the standard 12-tone equal temperament, but the JGU architectures have them. We will discuss these in the following section.

While the basic design for multitimbrality is rather straightforward, it must be noted that the details of the data structures sketched out above may become rather intricate if they are to be implemented efficiently. For details, we refer the reader to the source code of the faust-lv2 architecture, which is included in the Faust distribution and also available as a separate project.[3]

4. MTS SUPPORT

The MIDI Tuning Standard a.k.a. MTS is an addendum to the MIDI Standard 1.0 which provides for customizable tuning tables in a manufacturer-independent way [6]. While the author is not aware of any implementation of the MTS in hardware (Roland and Yamaha each have their own implementations, which are very similar to the octave-based tunings in MTS, though), there have been a few realizations in software, most notably in Fluidsynth [7].

Like Fluidsynth, the JGU architectures implement the octave-based tunings of the MTS. These basically consist of a tuning table (12 tuning offsets in Cent[4] relative to 12-tone equal temperament) in the form of a sysex message. There are in fact four different variations of these, depending on the range and resolution of the Cent values and real-time processing options; we refer the reader to the standards document or the faust-lv2 README file for all the gory details. One aspect that is important in the light of our

[3] `https://bitbucket.org/agraef/faust-lv2`

[4] 1 Cent = 1/100 of an equal-tempered semitone = 1/1200 of an octave

preceding discussion of multitimbrality is that each MIDI channel can have its own tuning table. We also mention in passing that the author has written a little Pure script named *sclsyx* which makes it easy to create MTS tuning tables from Manuel Op de Coul's Scala format.[5]

The JGU implementation defaults to the usual 12-tone equal temperament, which corresponds to an MTS tuning with all Cent offsets equal to zero.

In the general case, the frequency $\hat{f}_n$ of a MIDI note $n = 12k + i$ at position i in the kth MIDI octave ($i = 0 \triangleq \mathrm{C}, 1 \triangleq \mathrm{C}\sharp, \ldots, 10 \triangleq \mathrm{B}\flat, 11 \triangleq \mathrm{B}$) will be calculated from the offset c_i in Cents stored in the tuning table and the frequency f_n of the note in equal temperament as follows:

$$\hat{f}_n = f_n \times 2^{c_i/1200}$$

For instance, the 1-byte tuning offsets for a quarter comma meantone temperament rooted at standard pitch A would be as follows, with the hexadecimal contents of the MTS message format shown below the human-readable form (note that in this format a zero Cent offset is encoded as hexadecimal 40).

C	**C♯**	**D**	**E♭**	**E**	**F**	**F♯**	**G**	**G♯**	**A**	**B♭**	**B**
10	-14	3	20	-4	13	-11	7	-17	0	17	-7
4A	32	43	54	3C	4D	35	47	2F	40	51	39

Thus, C4 would be at $440 \times 2^{-9/12} \times 2^{1/120} \approx 263.14$ Hz in this tuning (whereas it is at about 261.63 Hz in equal temperament). Likewise, E4 is at about 328.87 Hz, as the reader can easily verify, yielding a major third of 5/4 (up to rounding errors).[6]

The JGU architectures update the tuning tables in real-time when an MTS sysex message is received. In addition, an extra tuning control is provided that allows easier automation in DAW environments, which sometimes make it inconvenient to directly inject sysex messages into a MIDI track. (For this tuning control a folder with the MTS tunings must be prepared beforehand, please see the faust-lv2 README for details.)

5. CONCLUSION

Grame's MIDI implementation, as described in [2], is very comprehensive and is certainly the recommended way to implement any new architectures requiring MIDI support in the future. The JGU architectures, having been there first, do not offer quite the same feature set, and are tied to the LV2 and VST environments. But they do have some compelling advantages when it comes to multi-channel support and MIDI tuning capabilities. There are a few other minor bits and pieces supported in these architectures which we didn't mention in this paper, such as all notes/sounds off, pitch bend range and master tuning messages. These are utilized by some DAW, MIDI player and tuning software, so it seems appropriate to have them supported in a comprehensive MIDI synthesizer implementation as well.

It remains to be seen how easily the extra JGU features can be incorporated into Grame's MIDI support, but it certainly seems to be doable and a worthwhile endeavor. This would instantly make those extra features available in all Faust architectures that have already been ported to the Grame MIDI and polyphony machinery, and it would also allow the faust-lv2 and faust-vst architectures to be ported over and take advantage of that infrastructure.

One area that still needs to be tackled in Faust's MIDI support is dynamic voice allocation. Right now all MIDI-enabled architectures require a fixed upper bound on the number of voices. While the upper bound can be chosen generously large to mitigate this restriction, it makes sense to just get rid of it, at least as an option. Of course, this must be implemented carefully in a real-time environment, but other dsp languages like SuperCollider and ChucK have been offering this for a long time, so there is no reason why the Faust project should not be able to follow suit.

6. REFERENCES

[1] Albert Gräf, "Creating LV2 plugins with Faust," in *Proceedings of the 11th International Linux Audio Conference*, IEM, Graz, 2013, pp. 145–152, IEM.

[2] Stéphane Letz, Yann Orlarey, Dominique Fober, and Romain Michon, "Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files," in *Proceedings of the 15th International Linux Audio Conference*, Jean Monnet University, St. Etienne, France, 2017, pp. 69–76, JMU.

[3] The MIDI Manufacturers Association, Los Angeles, CA, *The Complete MIDI 1.0 Detailed Specification (3rd ed.)*, 2014, online at `www.midi.org/specifications`.

[4] Albert Gräf, "Pd-Faust: An integrated environment for running Faust objects in Pd," in *Proceedings of the 10th International Linux Audio Conference*, Stanford University, California, US, 2012, pp. 101–109, CCRMA.

[5] GRAME, Centre National de Création Musicale, Lyon, *FAUST Quick Reference*, 2017.

[6] MMA, Los Angeles, CA, *MIDI Tuning Messages*, 1999, online at `www.midi.org/specifications`.

[7] David Henningsson, "FluidSynth real-time and thread safety challenges," in *Proceedings of the 9th International Linux Audio Conference*, Maynooth University, Ireland, 2011, pp. 123–128, NUI Department of Music.

[8] Gioseffo Zarlino, *Theorie Des Tonsystems : Das 1. Und 2. Buch Der Istitutioni Harmoniche (1573) / Gioseffo Zarlino. Aus d. Ital. Übers., Mit Anm., Kommentaren u.e. Nachw. Vers. von Michael Fend*, Lang, Frankfurt am Main, 1989.

[5] See `https://bitbucket.org/agraef/sclsyx` for the sclsyx script and `http://www.huygens-fokker.org/scala/` for information on the Scala program. A comprehensive archive of musical scales of all kinds and origins can be downloaded at `http://www.huygens-fokker.org/docs/scales.zip`.

[6] This shouldn't come as a big surprise to anyone who knows the theory, because the net offset of the zeroth and fourth scale positions in the above table amounts to 14 Cents, which matches the accumulated offset if you follow the first four fifths in the circle of fifths ($3 + 4 + 3 + 4 = 14$). Add to that a third of a Pythagorean comma of about 8 Cent and you get the size of the syntonic comma of about 22 Cent. (The numbers look a bit out of whack because we've rounded to integral Cent values. It goes without saying that a 2-byte encoding of the tuning would give much better accuracy.)

Reducing each perfect fifth by a quarter of the syntonic comma is in fact how Zarlino constructed quarter comma meantone in 1571 [8] and is also what gives the tuning its name. Note that we describe this here in modern-day terminology using A. J. Ellis' Cent scale, because that's what MTS uses. The Cent scale of course wasn't known at Zarlino's time (logarithms hadn't really been invented yet).