Programmation and Control of Faust Sound Processing in OpenMusic

Dimitri Bouche, Jean Bresson, Stéphane Letz

HAL Id: hal-01158363

https://hal.science/hal-01158363

Submitted on 31 May 2015

Programmation and Control of Faust Sound Processing in OpenMusic

Dimitri Bouche, Jean Bresson
STMS lab: IRCAM-CNRS-UPMC
1, place Igor Stravinsky, Paris F-75004
`dimitri.bouche@ircam.fr`
`jean.bresson@ircam.fr`

Stéphane Letz
Grame
11, cours de Verdun Gensoul, Lyon F-69002
`letz@grame.fr`

ABSTRACT

We introduce OM-Faust, an OpenMusic library including objects and functions to write, compile and control Faust programs. Faust is a domain-specific functional programming language designed for DSP. The integration of Faust in OpenMusic enables composers to program and compile their own audio effects and synthesizers, controllable both in real-time or deferred time contexts. This implementation suggests a more general discussion regarding the relationship between real-time and off-line processing in computer-aided composition.

1. INTRODUCTION

While the foundations of computer-aided composition environments initially focused on instrumental writing and symbolic music processing (i.e. at the "score" level) [1], its frontiers with the sound processing/synthesis realm have regularly been challenged and expanded [2, 3, 4]. Technological developments combined with composers' demand for flexible and high-quality audio rendering leads to further growth in this area of computer-aided composition software.

Our current work takes place in the OpenMusic environment [5, 6]. OpenMusic (OM) is a visual programming language allowing composers to design programs leading to the creation or transformation of musical scores, sounds or other kind of musical data. OM includes an audio engine for sound rendering, but generally does not operate at the level of sound signals. Instead, interfaces have been developed with sound processing and synthesis systems, such as Csound, SuperVP, Chant, Spat, etc. [7, 8, 9].

In this paper, we present a new interface developed between OM and the Faust real-time signal processing language. Our objective is not to offer new audio processor units in OM, but to provide a framework for composers to build their own audio effects and synthesizers in this language. The tight coupling between Faust and LibAudioStream, the current audio rendering engine in OM, allows it to achieve dynamic integration of the language in the compositional environment, and enables both real-time

and off-line ("composed") control of the effects and synthesizers.

After a quick presentation of Faust and the OM audio architecture, we will present the new OM objects and methods involved in this work, highlighting the distinction between the real-time and off-line approaches.

2. THE OM-FAUST CONNECTION

2.1 LibAudioStream as OM Audio Architecture

Since OM 5 [10], the audio architecture in OM is based on the LibAudioStream library [11]. LibAudioStream (LAS) provides a relatively high-level and multi-platform API for multi-track mixing and rendering of audio resources.

Audio resources in OM (*sound* objects) are converted to LAS *streams*, processed and combined together using an algebra of stream composition operations [1] and eventually loaded in one of the tracks of the LAS player for rendering. This process can be monitored thanks to a standard multi-track audio mixing console window.

The basic stream processing units of LAS are also available as functional units (boxes) to be used in OM visual programs: users can connect them together to process, combine and transform sounds algorithmically (see Figure 1). Eventually, an offline rendering utility performs a "virtual run" of the player and redirects the resulting stream to a sound file, hence allowing it to generate new sounds from the programmed audio processing chain [12].

2.2 The Faust Language

Faust (Functional AUdio STream) is a functional programming language designed for real-time signal processing and synthesis [13]. It targets high-performance signal processing applications and audio plug-ins development for a variety of platforms and standards.

This specification language aims at providing an adequate functional notation to describe signal processors from a mathematical point of view. Faust is, as much as possible, free from implementation details. Resulting programs are fully compiled, not interpreted: the compiler translates Faust programs into equivalent C++ programs taking care to generate the most efficient code. The result can generally compete with, and sometimes even outperform, C++ code written by seasoned programmers [14]. The generated code works at the sample level; it is therefore suited

---

[1] A stream is described as the result of combining others stream using several operators like *sequence*, *mix*, *cut*, *loop*, *transform...*
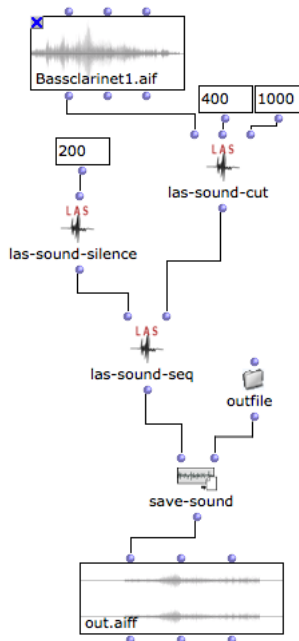
Figure 1. Processing sounds in OpenMusic using the LibAudioStream framework.

to implement low-level DSP functions, like recursive filters. The code is self-contained and does not depend of any DSP library or runtime system. It has a very deterministic behaviour and a constant memory footprint.

Faust allows users to program effects and synthesizers, and to use them in the environment of their choice. A significant number of libraries are available for import into the programs. These libraries cover most of the basic audio processing and synthesis functions (filters, reverbs, signal generators, etc.). The programs can be compiled for specific target "architectures" (e.g. Max, PureData, VST...) [15] and then provide corresponding entry points for control.

2.3 LAS-Faust Connection

The Faust compiler is generally used as a command line tool to produce a C++ class starting from the DSP source code. This class can be compiled later on with a regular C++ compiler, and included in a standalone application or plug-in. In the *faust2* development branch, the Faust compiler has been packaged as an embeddable library called *libfaust*, published with an associated API.

The complete chain therefore starts from the DSP source code, compiled into the LLVM intermediate representation (IR) [2] using the *libfaust* LLVM backend, to finally produce the executable code using the LLVM Just In Time compiler (JIT); all these steps being done in memory (see Figure 2) [17]. This dynamic compilation chain has been embedded in the LibAudioStream library, so that Faust effects or synthesizers can be dynamically complied and inserted in the LAS audio chain.

---

[2] LLVM (formerly Low Level Virtual Machine) is a compiler infrastructure designed for compile-time, link-time, run-time optimization of programs written in arbitrary programming languages [16].
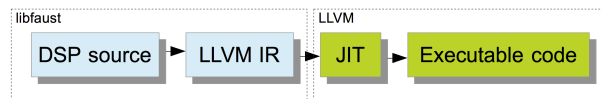

Figure 2. Steps of the compilation chain in Faust.

Faust and the LibAudioStream library are therefore compatible at different levels :

- LAS can read and connect compiled Faust effects in the audio processing chains (Faust signal processors can be used to transform audio streams),

- LAS connects to the control parameters of the Faust effects and provides an API to set or read their values (Faust signal processors can be controlled in real-time using LAS),

- LAS embeds the *libfaust* and LLVM technologies and can compile Faust code on-the-fly from a text buffer (Faust effects can be created and applied dynamically).

These cross-connections between Faust and LAS are the basis of the high-level control interface proposed in OM-Faust, which we will detail in the next sections.

3. FAUST OBJECTS

Two main objects have been developed in the OM-Faust library: *Faust-FX* and *Faust-Synth*.

3.1 Effects: *Faust-FX*

Faust-FX is a standard OM object represented in the visual programs as a "factory" box (see Figure 3). Its inputs allow to build an instance of the object upon evaluation: a Faust program (edited or imported in a *Textfile* object, the standard OM text editor), a name (optional) and a track number (to plug the effect on a specific track of the audio player – the assignation of an effect to a track can also be done through the mixing table, see section 3.3).
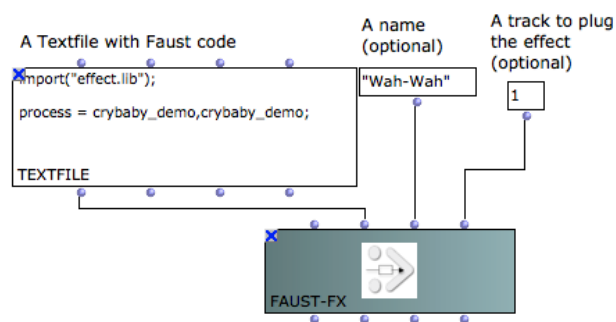

Figure 3. The *Faust-FX* object and it's input/initialization values.

The evaluation of this box handles the different steps that the architecture needs to compile and register the resulting audio processing unit:

- Compilation of the code (incl. error management),
- Registering in a pool of available effects,
- Connection to the audio system (if *track* specified),
- Generation of control interfaces (see section 4).

The Faust compilation process also outputs a SVG file containing a hierarchical block-diagram representation of the signal processing unit, which can be visualized and explored in a web browser.

3.2 Synthesizers: *Faust-Synth*

At first sight, the *Faust-Synth* object is quite similar to *Faust-FX* (see Figure 4), but it differs in its use and purpose in the environment. The function of the synthesizer in our system (producing sound) is different from the one of an effect (transforming an existing sound or audio *stream*). Effects, for instance, can be applied sequentially on top of one another in a processing chain. Synthesizers at the contrary should be considered as a source (which can also be processed by a chain of effects or treatments).
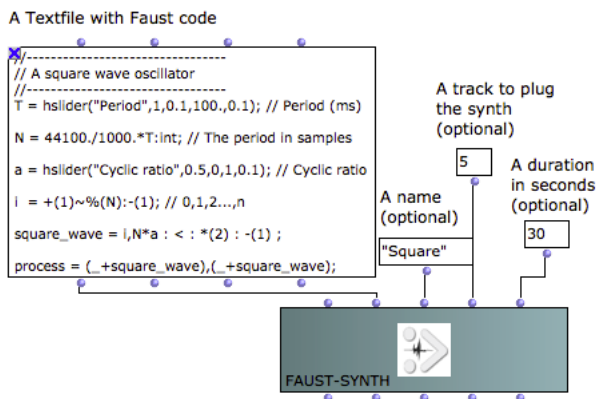

Figure 4. The *Faust-Synth* object and its input/initialization values.

This distinction implies a number of differences in the treatment of these two kinds of objects. It is possible for instance to assign several *Faust-FX* to a same audio track, but one single *Faust-Synth* will be accepted on a track.

Faust-Synth also has a *duration* as additional initialization parameter. Indeed, "off-line" processes such as the ones taking place in computer-aided composition must generally produce time-bounded structures, and therefore include a notion of duration.[3] Being an individual sounding object (and not a processing unit applied to an existing one), *Faust-Synth* is likely to be played and integrated in an audio mix or sequence (in an OM visual program, or in a sequenced context such as the *maquette* [18]) along with other sounds.

3.3 Connections to Audio Tracks in the Audio Mixer

The OM audio mixing table enables volume and panning control for each audio player track. OM-Faust extends this

interface, providing post/dynamic controls for the the Faust audio units' assignments to the audio tracks and resources. Figure 5 shows the *Audio Mixer* window with the OM-Faust plug system extension. The SYNTH and FX selectors complement the Faust objects initialization process, connecting compiled and registered instances of *Faust-FX* and *Faust-Synth* with the LAS player tracks.
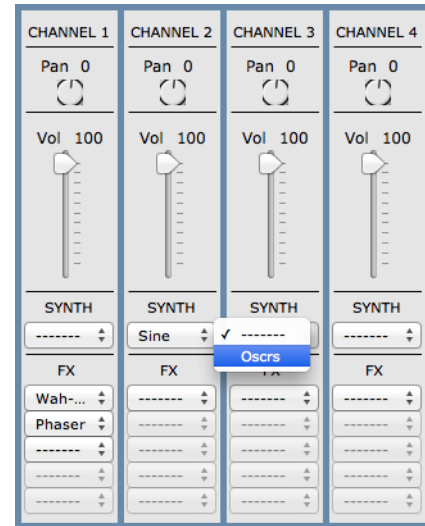

Figure 5. The *Audio Mixer* and its Faust extension.

4. SELF GENERATED INTERFACES : REAL-TIME CONTROL OF FAUST OBJECTS
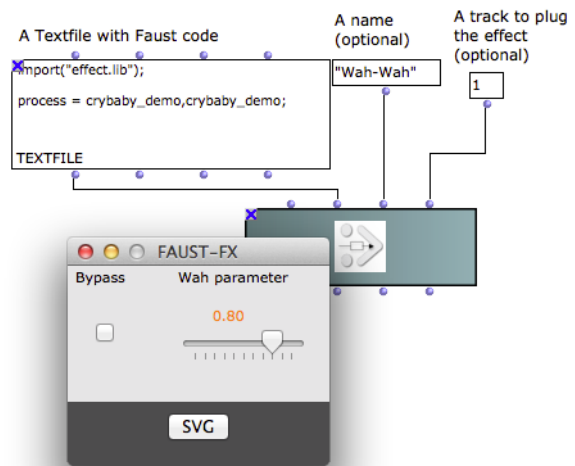

Figure 6. A *Faust-FX* object and its self-generated control interface (the "SVG" button opens the block-diagram representation of the effect in a web browser).

Faust code can embed elements of control interface specification (sliders, check-boxes, knobs or other kinds of components that will set the value of the effects or synthesizers' parameters). The compiler interprets this specification depending on the target architecture or can export it as a

---

[3] At this juncture lies one of the main paradigmatic divisions between computer-aided composition and real-time audio processing systems, which usually consider virtually infinite audio or data streams.

Json file [19]. The initialization of a *Faust-FX* or *Faust-Synth* in OM uses and parses this Json specification file to build a hierarchical representation of the user interface corresponding to the effect or synthesizer controls, presented to the user upon opening (double-clicking) the effect or synthesizer's box (see Figures 6 and 7).
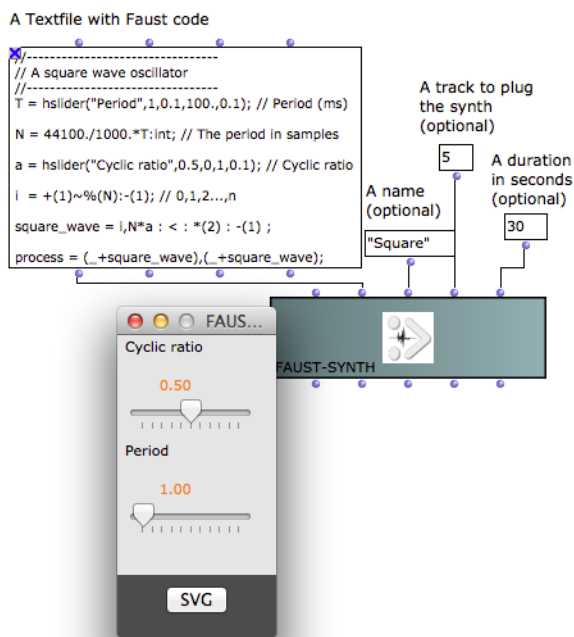

Figure 7. A *Faust-Synth* object and its self-generated control interface.

These control interfaces are connected to the control parameters of the Faust processing unit via LAS, and work in real-time depending on the connections of these units in the global audio setup.

5. FAUST-AUTOMATION : A BRIDGE BETWEEN OFF-LINE AND REAL-TIME CONTROL

One of the main interests in using a computer-aided composition environment like OM to perform sound processing or synthesis, is the possibility to design complex processes for the generation of DSP structures and controllers. *Faust-Automation* is another object included in the OM-Faust library, derived from the *BPF* object (break point function), bringing the capability of programming or hand-drawing curves that will serve as pre-generated controllers (automations) for the Faust effects or synthesizers (see Figure 8). A *Faust-Automation* controls one specific named parameter, and it is possible to use as many automations as the number of parameters of the targeted Faust object.

Faust automations can be used both in real-time or off-line sound processing contexts. In an off-line context (the "usual" context for computer-aided composition), it will be considered as a sequence of timed values for the different Faust processing controllers (see Section 6). In a "real-time" context (to control a Faust process currently running on the sound player, e.g. attached to a playing audio track),
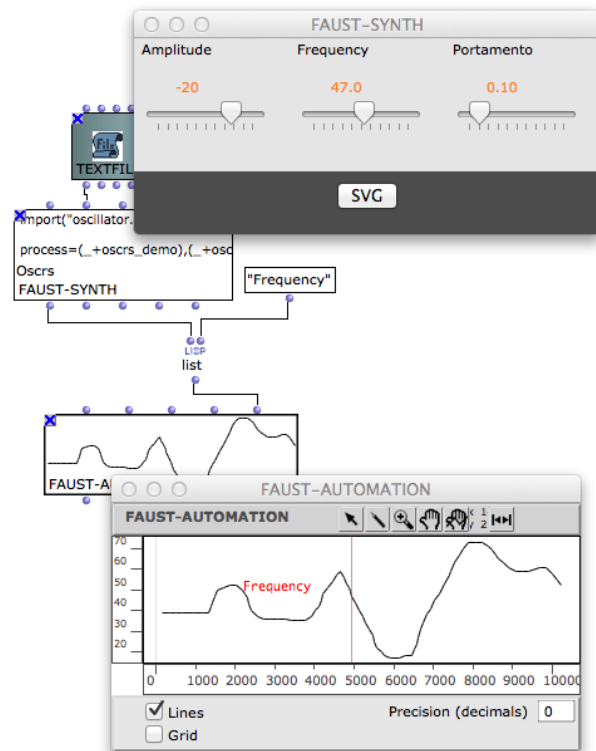

Figure 8. *Faust-Automation* object.

Faust automations are "played" as independent musical objects. A plug-in to the OM player system schedules Faust control value changes via LAS at every timed-point in the automation BPFs. They can also be integrated as components in more complex musical structures, for instance in a *maquette* (see Figure 9).

6. OFF-LINE APPROACH TO FAUST OBJECTS

We have seen in the previous sections how Faust objects can be used through the OpenMusic audio architecture and modify/build output streams in real-time.

They can also be used as off-line processing objects, and enable more complex compositional approaches.

6.1 Sound processing : *Apply-Faust-FX*

LAS offers the possibility of connecting effects on a player track, but also directly as a transformer on a sound stream. The *Apply-Faust-FX* method simulates a sound file going through a *Faust-FX* and produces a new sound file. A virtual run of the sound+effect rendering is performed and redirected to an output file.

Figure 10 shows an example of the off-line transformation of a sound by a *Faust-FX* via *Apply-Faust-FX*. The effect parameters can be controlled using optional inputs of *Apply-Faust-FX*, and specified either as constant values or as a set of one or more *Faust-Automation* objects gathered in a matrix structure. In the latter case, at each audio buffer writing operation during the "virtual rendering", a new parameter value is read in the *Faust-Automation* and set as control value. It is therefore possible to change an
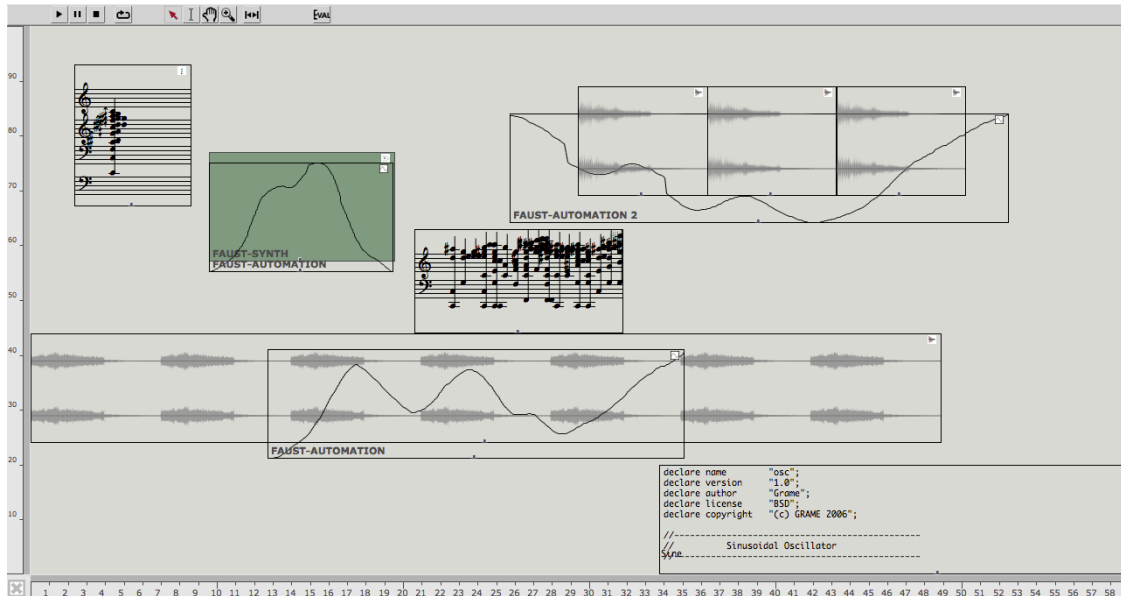
Figure 9. *Faust-Automations* and other *SOUND* and OM-Faust objects in a maquette. Two automations control parameters of an effect applied to sound files, and an other a parameter of *Faust-Synth* process. Automations are bounded and scalable to specific intervals on the maquette time-line.

automation grain by setting the length of the buffers (in *Apply-Faust-FX*, the default buffer length is set to 256 samples, which corresponds to 5.8 ms for a 44.1 kHz sample rate). An optional gain control can also be specified to *Apply-Faust-FX*, for instance to avoid clipping in amplified effects.

Figure 10. The *Apply-Faust-FX* method, used to process a sound file, using a Flanger effect with two automated parameters.

6.2 Synthesis rendering : *Flatten-Faust-Synth*

The *Flatten-Faust-Synth* method is similar to the *Apply-Faust-FX*, applied to a *Faust-Synth*. Instead of using a sound as input, it simply uses a duration setting to limit the output length (see section 3.2).

With the same rendering utility as described in section 6.1, *Flatten-Faust-Synth* can be controlled with constant or varying parameter values (using automations) it outputs a sound file of the required duration to the disk. Figure 11 shows an OM patch where a sound file is synthesized from a Faust program and a set of parameter values and automations.
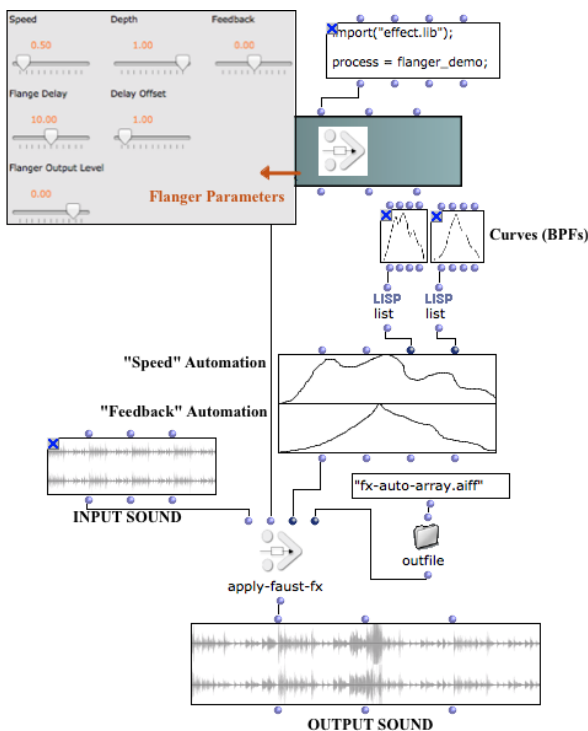
7. CONCLUSION

OM-Faust allows to write, compile and control Faust objects in OpenMusic. This functional DSP programming framework offers users a low-level control of the audio processors they build, coupled with dynamic real-time control interfaces and powerful offline rendering facilities.

While common music software mostly process audio signal in real-time, OM-Faust objects can be used both in real-time and deferred-time in the same work session. These objects can produce or transform musical material (audio streams), and/or be considered as primary musical material themselves. This duality introduces interesting issues in the compositional environment, which we plan to address in future works from the user point of view and with the development of hybrid scheduling strategies for computation and rendering.

OM-Faust is an open-source library and is distributed with a set of tutorial patches. [4]
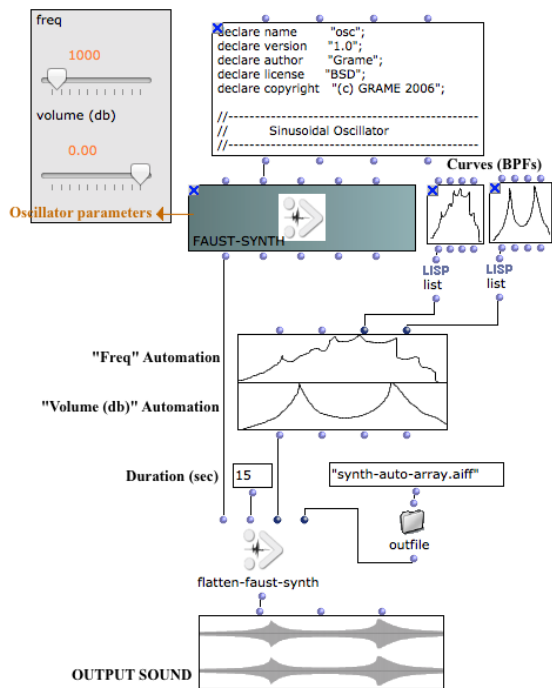
---

[4] http://forumnet.ircam.fr/product/openmusic-libraries/

Figure 11. The *Flatten-Faust-Synth* method, used to "synthesize" a 15 seconds sound file from a Faust oscillator, with a volume and a frequency automations.

Acknowledgments

8. REFERENCES

[1] G. Assayag, "Computer Assisted Composition Today," in *1st symposium on music and computers*, Corfu, 1998.

[2] J. Bresson and C. Agon, "Musical Representation of Sound in Computer-aided Composition: A Visual Programming Framework," *Journal of New Music Research*, vol. 36, no. 4, pp. 251–266, 2007.

[3] M. Laurson, V. Norilo, and M. Kuuskankare, "PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control," vol. 29, no. 3, 2005.

[4] J. McCartney, "SuperCollider: A New Real Time Synthesis Language," in *Proceedings of the International Computer Music Conference*, Hong Kong, China, 1996.

[5] G. Assayag, C. Rueda, M. Laurson, C. Agon, and O. Delerue, "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic," vol. 23, no. 3, 1999.

[6] J. Bresson, C. Agon, and G. Assayag, "OpenMusic. Visual Programming Environment for Music Composition, Analysis and Research," in *ACM MultiMedia 2011 (OpenSource Software Competition)*, Scottsdale, AZ, USA, 2011.

[7] C. Agon, J. Bresson, and M. Stroppa, "OMChroma: Compositional Control of Sound Synthesis," *Computer Music Journal*, vol. 35, no. 2, 2011.

[8] J. Bresson and M. Stroppa, "The Control of the Chant Synthesizer in OpenMusic: Modelling Continuous Aspects in Sound Synthesis," in *Proceedings of the International Computer Music Conference*, Huddersfield, UK, 2011.

[9] J. Bresson and M. Schumacher, "Representation and Interchange of Sound Spatialization Data for Compositional Applications," in *Proceedings of the International Computer Music Conference*, Huddersfield, UK, 2011.

[10] J. Bresson, C. Agon, and G. Assayag, "OpenMusic 5: A Cross-Platform release of the Computer-Assisted Composition Environment," in *Proceedings of the 10th Brazilian Symposium on Computer Music*, Belo Horizonte, MG, Brasil, 2005.

[11] "Libaudiostream," http://libaudiostream.sourceforge.net/, [online] Retrieved Oct. 29, 2013.

[12] J. Bresson, "Sound Processing in OpenMusic," in *Proceedings of the International Conference on Digital Audio Effects*, Montreal, Canada, 2006.

[13] Y. Orlareya, D. Fober, and S. Letz, "Faust: An Efficient Functional Approach to DSP Programming," in *New Computational Paradigms for Computer Music*, G. Assayag and A. Gerzso, Eds. Delatour France / Ircam, 2009.

[14] N. Scaringella, Y. Orlarey, D. Fober, and S. Letz, "Automatic Vectorization in Faust," in *Actes de Journées d'Informatique Musicale*, Montbéliard, France, 2003.

[15] D. Fober, Y. Orlarey, and S. Letz, "Faust Architectures Design and OSC Support," in *Proceedings of the International Conference on Digital Audio Effects*, Paris, France, 2011.

[16] C. Lattner, "LLVM : An Infrastructure for Multi-Stage Optimization," Master's thesis, University of Illinois at Urbana-Champaign, USA, 2002.

[17] A. Gräf, "Functional Signal Processing with Pure Data and Faust using the LLVM Toolkit," in *Proceedings of the Sound and Music Computing Conference*, Padova, Italy, 2011.

[18] J. Bresson and C. Agon, "Temporal Control over Sound Synthesis Processes," in *Proceedings of Sound and Music Computing Conference*, Marseille, France, 2006.

[19] D. Crockford, "RFC 4627. The application/json Media Type for JavaScript Object Notation (JSON)," The Internet Society, Tech. Rep., 2006.