FAUST online IDE: dynamically compile and publish FAUST code as WebAudio Plugins

Shihong Ren, Stéphane Letz, Yann Orlarey, Romain Michon, Dominique Fober
GRAME, 11 cours de Verdun LYON
renshihong@hotmail.com
(letz, orlarey, michon, fober)@grame.fr

Michel Buffa, ElMehdi Ammari, Jérome Lebrun
Université Côte d'Azur
CNRS, INRIA
(buffa, lebrun)@i3s.unice.fr,
ammarielmehdi@gmail.com

ABSTRACT

The development and porting of virtual instruments or audio effects on the Web platform is a hot topic. Several initiatives are emerging, from business enterprise based ones (Propellerhead Rack Extension running on the Web[1]), to more community based open-source projects [10]. Most of them aim to facilitate adapting existing code base (usually developed in native languages like C/C++) as well as facilitating the use of existing audio DSP languages and platforms.

Our group previously presented an open format for WebAudio Plugins named WAP [11]. It aims to facilitate the interoperability of audio/MIDI plugins developed either using pure Web APIs, porting existing native code bases, or using Domain Specific Languages (DSL).

In the DSL category, we already did developments to use the FAUST audio DSP language. In this paper, we present a solution based around FAUST, its redesigned Web based editor, and the integration of a plugin GUI editor allowing to directly test, generate and deploy WAP plugins.

Recent improvements done in the toolchain, going from the DSP source to a ready-to-use WAP compatible plugin will be presented. The complete workflow, from the Faust DSP source written and tested in a fully functional editor, to a self-contained plugin running in a separate host application, will be demonstrated.

1.   INTRODUCTION

There are many ways to develop software with the WebAudio API today. In pure JavaScript, the genish.js environment for instance [13] allows to develop sample level audio processing techniques[2]. Already C/C++ written code can be transpiled to WebAssembly using Emscripten [12], or by using domain specific languages for programming DSP algorithms that also compile to WebAssembly, like the mature Csound [14] with its set of WebAudio examples[3], or the recently announced SOUL DSP

language with its playground[4]. They all provide a dedicated and usually self-contained working environment.

When audio effects or audio/MIDI instruments have to be shared between several DAWs or audio environments, a plugin model is usually preferred.

Several native audio plugin formats are now popular, including Steinberg's VST format (Virtual Studio Technology, created in 1997 by Cubase creators), Apple's Audio Units format (Logic Audio, GarageBand), Avid's AAX format (ProTools creators) and the LV2 format from the Linux audio community. Although the APIs offered by these formats are different, they all exist to achieve more or less the same thing: to represent an instrument or an audio effect, and to allow its loading by a host application. In the first years after the birth of the WebAudio standard (2011), there was no standard format for high-level audio plugins. With the emergence of Web-based audio software such as digital audio workstations (DAWs) developed by companies such as SoundTrap, BandLab or AmpedStudio, it was desirable to have a standard to make WebAudio instruments and effects interoperable as plugins compatible with these DAWs and more generally with any compatible host software.
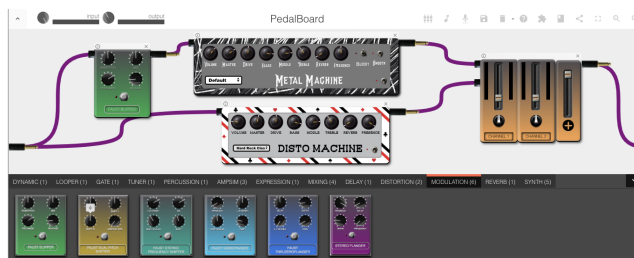
Figure 1: the virtual pedalboard host application scans multiple remote WAP plugin servers. WAP plugins can then be dragged and dropped and assembled in a graph.

Such a plugin standard needs to be flexible enough to support these different approaches, including the use of a variety of programming languages. New features made possible by the very nature of the Web platform (e.g., plugins can be remote or local

---

[1]
https://www.reasonstudios.com/press/275-reasons-flagship-europa-synth-now-available-as-a-plugin-for-other-daws-and-on-the-web
[2] http://www.charlie-roberts.com/genish/
[3] https://waaw.csound.com

[4] https://soul.dev

and identified by URIs) should also be available for plugins written in different ways. To this end, some initiatives have been proposed [3, 9] and with other groups of researchers and developers we made in 2018 a proposal for a WebAudio plugins standard called WAP (WebAudio Plugins), which includes an API specification, an SDK, online plugin validation tools, and a series of plugin examples written in JavaScript but also with other languages[5].

These examples serve as proof of concept for developers and also illustrate the power of the Web platform: plugins can be discovered from remote repositories, dynamically uploaded to a host WebApp and instantiated, connected together etc. The project includes examples of very simple plugins and host software, but also more ambitious software to validate the WAP standard: a virtual guitar "pedalboard" that discovers plugins from several remote repositories, and allows the musician to chain for example virtual audio effects pedal plugins, synthesizers, guitar amplifier simulators, drum machines etc. and to control them via MIDI in real time (Figure 1). The reader can get a "multimedia" idea of this work by watching online videos that present the results of this work[6]. Since last year, WAP now includes support for pure MIDI plugins (a GM midi synthesizer, virtual midi keyboards, a MIDI event monitoring plugin, etc[7]). For details about the WAP proposal, and how it is related to other approaches like Web Audio Modules (WAMs), WebAudio API eXtension (WAAX) or JavaScript Audio Plugin (JSAP), see [11].

In the next sections we will focus on a new online IDE we developed, that is well suited for coding, testing, publishing WAP plugins written in FAUST, directly in a Web browser. The IDE includes a graphical interface editor that allows developers to fine-tune the look and feel of the plugins. This editor offers a rich set of widgets that can be controlled by midi-learn. Once complete (DSP + GUI) the plugins are packaged in the form of standard W3C WebComponents and published on remote WAP plugin servers. The plugins will then be directly usable by any compatible host software, using their URIs. You can imagine WAP plugins as images in an HTML document, their URI is sufficient, and can be dynamically retrieved using APIs from a remote Web Service.

2. BACKGROUND CONTEXT AND TERMS

FAUST [6] is a functional, synchronous, domain specific programming language designed for real time audio signal processing and synthesis.

The FAUST compiler is organized in successive stages, from the DSP block diagram to signals, and finally to the FIR (FAUST Imperative Representation) which is then translated into several target languages. The FIR language describes the computation performed on the audio samples in a generic manner. It contains primitives to read and write variables and arrays, do arithmetic operations, and defines the necessary control structures (for and while loops, if structure etc.).

As a specification language, the FAUST code only describes the DSP part, and an abstract version of the control interface. It says nothing about the audio drivers or the GUI toolkit to be used. Architecture files are written to describe how to connect the DSP code to the external world.

Additional generic code is added to connect the DSP computation itself with audio inputs/outputs, and with parameter controllers, which could be buttons, sliders, numerical entries etc. Architectures files can also possibly implement polyphonic support for MIDI controllable instruments, by automatically dealing with dynamic voice allocation, and decoding and mapping of incoming MIDI events [15].

Several prior developments have been done to use the language on the Web platform. By adding an asm.js generating backend in the compiler, and compiling the compiler itself in asm.js/JavaScript using the Emscripten transpiler, the dynamic generation of WebAudio nodes from FAUST code has been demonstrated [16] [17].

With the apparition of the more stable and better designed WebAssembly format in 2017, as a replacement of asm.js, the previous work done with asm.js has been adapted. For the Web platform, two backends have been developed to generate WebAssembly text (so-called "wast" or "wat") and binary formats (so-called "wasm") [7]. When embedded in the FAUST compiler running on the Web, they allow to dynamically compile FAUST DSP programs in pure Web applications. Additional JavaScript glue code is added to transform DSP modules in fully functional WebAudio nodes.

FAUST also allows to circumvent some important buffer size issues that we encountered in our previous works on the implementation of signal loops in WebAudio. For example, the Negative Feedback Loop (NFB) as in our push-pull tube amps simulations [1, 2] is a tricky issue due to some WebAudio API limitations and divergences/bugs in how browsers generally parse the WebAudio graphs with loops. In the WebAudio API specs, loops in the graph are required to include at least a delay node. Without this delay node, Firefox stops rendering the graph, while Chrome does not complain but adds, behind the scenes, a 3 ms delay (minimum size of an audio buffer is 128 frames hence a minimal delay of 128/sampling rate or roughly 3 ms at 44.1kHz). Now, to faithfully implement loops like the NFB with its RC network inducing short delays, finer precision at the level of some samples is required. With the current limitations, and quite strangely, a 3ms delay in the loop to conform to the specs, was bringing slightly different coloring of the amps between FF and Chrome. This example shows the need for solutions such as FAUST to circumvent these limitations of the WebAudio standard.

3. CURRENT STATE

3.1 The new FAUST Web editor

The Emscripten module was previously implemented in the FAUST IDE using a JavaScript wrapper which allowed the application to compile and transform FAUST source code into a WebAudio node. We recently restructured this wrapper, in order to take advantage of modern JavaScript development environments. An updated tool-chain is now used to ensure the efficiency and the compatibility of the wrapper to transform it into another JavaScript UMD module.

The past versions of this wrapper already provided the following features:

- Load WebAssembly version of the FAUST compiler and import its C functions into JavaScript

- Compile the code: the input is the FAUST source code, the output is the compiled WebAssembly binary version with some related data
- Load and wrap the module as a DSP processing function inside an *AudioWorkletProcessor* or a *ScriptProcessor* AudioNode

We added some new features into the module:

- A virtual file system: Emscripten supports a virtual file system (in memory) compatible with the C++ I/O standard library, but also usable from the JavaScript wrapper. This file system became important as the FAUST compiler searches libraries and imported source codes locally, or generates DSPs code for other targets/architectures. For instance SVG diagrams generated as additional files in the compilation process, are simply written on the fly in the VFS, then loaded, decoded and displayed.
- Data output: a callback has been added into the AudioWorklet node to support additional processing or analysis after the buffer has been fully calculated. This callback returns the current output buffer, the current buffer index and parameters change events. In addition, to be able to calculate audio separately with a FAUST DSP independent from the browser audio context, we created an "offline processor" which will be used exclusively for getting the very first samples calculated by a DSP. This allows us to debug the DSP code running with a different sample rate.

Based on the previous FAUST online editor, we built a code editor (Figure 2) with full IDE user experience that could provide more information and details of a DSP through graphical representation in a Web page. A DSP developer probably not only needs to hear how the DSP sounds, but also to test it with other audio inputs, or to precisely display the time domain and frequency domain data of outputs. We have added several testing, visualisation and debugging tools into a basic code editor.

The layout is responsive and configurable following the browser viewport dimensions:

- All options related to FAUST code compilation are located using controllers from the left sidebar panel
- All options and displays related to DSP runtime, such as MIDI, audio inputs and quick signal probing are placed in the right sidebar panel
- The remaining central region of the page is divided into two parts with configurable heights: a source code editor on the top and a multi-tab display panel which can display the logs from the compiler, a FAUST block diagram corresponding to the DSP code, a larger signal scope, a running GUI of the plugin being developed, and finally a GUI Builder / exporter for designing the user interface a WAP plugin version of the code, usable in external host applications

Besides UI improvements that facilitates code editing and compiling, audio probes are an important addition to the new editor. We designed four modes of signal visualisations: data table, oscilloscope (stacked and interleaved by channels), spectroscope and spectrogram, to help FAUST users to debug their DSPs. To implement all four probe modes, precise sample

values are needed. In the browser environment, we have two ways to get audio output samples.
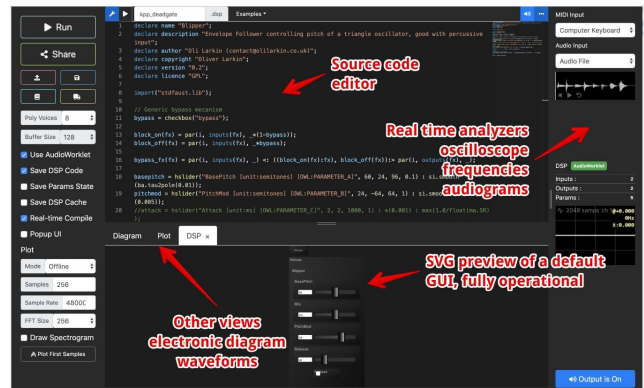
Figure 2: the FAUST IDE provides many embedded tools: oscilloscopes, spectroscope and spectrogram, functional default GUI, schema preview, etc.

The first method consists of using WebAudio *AnalyserNode* with its integrated methods:

> `getByteFrequencyData`, `getByteTimeDomainData`, `getFloatFrequencyData`, `getFloatTimeDomainData` (which does not exist in Safari)

This method provides both sample values and a spectrum given by the FFT of the current audio buffer. However, it has several drawbacks. Firstly, the *AnalyserNode* has only one input, which means it needs an additional *ChannelSplitterNode* to retrieve the correct channel from the FAUST DSP Node. Secondly, as we cannot tell when the *AnalyserNode* does an analysis, the audio data are provided only on demand. Thus it is impossible to get precise data in a specific buffer calculated by the FAUST DSP.

The second method consists of getting the sample values directly with a callback in a FAUST DSP AudioNode. These values are associated with its buffer index and an event list containing all parameter changes occurring in this buffer. To get the corresponding frequency domain data, an additional FFT is required. We chose the JavaScript version of KissFFT[8] for its high performance when compiled to WebAssembly[9]. Thus, we perform the FFT computation in the FAUST online editor with 2 overlaps using a Blackman window function.

The first method is used in the implementation of the two scopes in right sidebar as it can also probe the audio input. The second method is used for the larger scope at the bottom. It is more flexible and can adapt itself to continuous or on-demand signal display.

Developers may need to have options to select which part of the signals they want to display: we provide four modes to trigger differently the drawing function of the scopes: *Offline*, *Continuous*, *On Event* and *Manual*:

---

[8] https://github.com/j-funk/kissfft-js
[9] https://github.com/j-funk/js-dsp-test/

- *Offline*: FAUST WebAudio wrapper offers an "offline processor" which is useful to allow a DSP to calculate the first samples at any sample rate independently of the actual audio context one.
- *Continuous*: similar to normal audio scopes, this mode draws in real time the most recent samples processed by the FAUST DSP. Parameter change events will be shown in the scope. On a mainstream personal computer, the editor is able to draw up to 1 million samples continuously without significant rendering lagging.
- *On Event*: as the FAUST DSP usually comes with a GUI to control its parameters, it is important to visualize the part of signals while parameters change. In this mode, the scope draws only when it captures parameter change events, which is useful for debugging.
- *Manual*: in Manual mode, the scope displays the latest samples when a user clicks on a button.

After a FAUST DSP is tested in the editor, users can export the DSP to different architectures including WebAudio Plugins (WAPs). A dedicated GUI builder is integrated in the online IDE that receives FAUST DSP's GUI definitions while it is compiled. Then, a default GUI is proposed and users can start customizing the GUI, testing the plugin functionalities, and finally export the plugin to a remote server. This is detailed in the next section.

3.2 The GUI Editor

FAUST code can include abstract definitions of GUI controllers, such as in this source code extract:

```
basepitch = hslider("BasePitch
[unit:semitones]", 60, 24, 96, 0.1) :
si.smooth(ba.tau2pole(0.01));

pitchmod = hslider("PitchMod
[unit:semitones]", default_pitch*2, -64, 64,
1) : si.smooth(ba.tau2pole(0.005));
```

Here, the code describes the definition of two parameters named "BasePitch" and "PitchMod" along with some data that define the default value, min, max, step, unit type, etc. These parameters can be programmatically set/computed such as "default_pitch*2" in the second example, instead of using literal values.
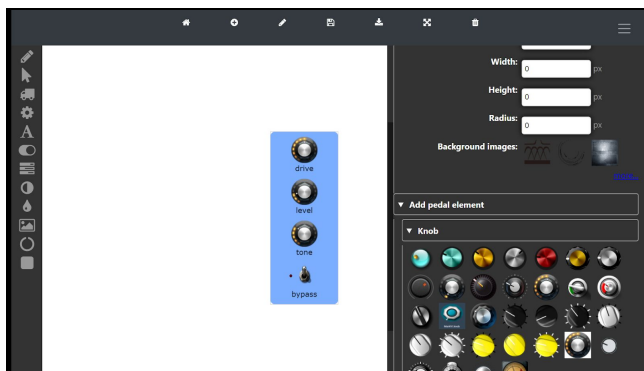

Figure 3: from the FAUST code, a WAP default GUI is proposed in the editor.

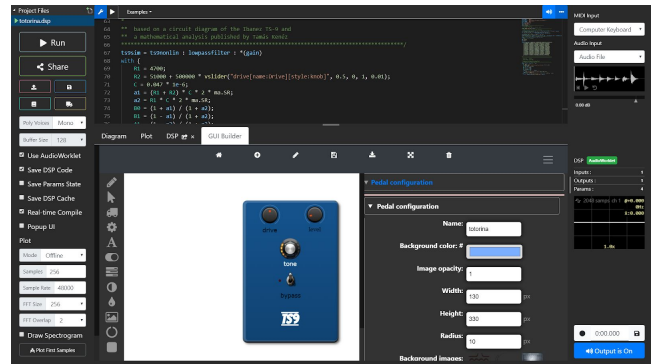
Figure 4: The default GUI can be customized: change textures, knobs, sliders, switches positions size, apparence and labels etc.

As explained in the previous sections, the FAUST DSP code is compiled to a JavaScript wrapper and a WebAssembly module. This is all done client-side. The GUI builder shares a JavaScript parameter descriptor variable that has been generated after the compilation step and that can be statically interpreted. From this parameter descriptor, a "GUI pivot descriptor" is created and a "default GUI" displayed in the GUI builder (Figure 3), that can be enriched during the GUI edition process and that will be used to generate the final GUI code (Figure 4). So far, we implemented only a generator for WebAudio plugins, using HTML/CSS/JS code that follows the W3C WebComponents specifications[10].

Figure 5: Other designs for the same DSP code

At any time, the plugin (DSP + GUI) can be tested from within the IDE, without the need to download it on a local disk. It is then possible to refine the GUI, adjust the layout, appearance of the controllers among a rich set of knobs, sliders, switches (Figure 5 shows different looks and feels that can be created from the same DSP code). The editor is not yet 100% bijective with the FAUST definition of GUI controllers (that serve as a "hint" to bootstrap the GUI design process). For example, if you change the type of controller (i.e. slider to knob), it does not change the FAUST code back. However, having a way to build and customize a GUI this way is a great time saver, full sync between the FAUST code and GUI is planned as future enhancements.

---

[10] The WebComponents W3C standard (now in the HTML 5.2 specification) defines a way to easily distribute components with encapsulated HTML/CSS/JS/WASM code without namespace conflicts. See https://www.webcomponents.org

The plugin can be published on a remote plugin server, using standard Web services, this is shown in the life cycle workflow from Figure 6. A WAP plugin generated by the FAUST online IDE is a zipped archive file that contains the DSP WebAssembly module, the standard JSON WAP descriptor and the GUI code (HTML/CSS/JS) wrapped as a WebComponent. It also includes a host HTML page for trying and testing the plugin, making the plugin usable by humans as well as by client applications. In fact, once published on a server, this file is unzipped in a remote directory. The plugin is associated with a "remote URI" and can be "unit tested" by different validation tools that come with the WAP SDK[11] (i.e. check that their API is following the specification, that the plugin is able to save/restore its state etc.).

Figure 1 shows the "virtual pedalboard" Web application, a host for WAP plugins we developed to showcase the WAP standard, that targets guitar and keyboard players. This application scans remote WAP servers for available plugins and makes them accessible to final users that can drag and drop and assemble them in the main part of the screen. In this example, all virtual pedal effects at the bottom of the screen have been coded and compiled with their GUI designed and tested in the FAUST online IDE.

4. DISCUSSION

The authors' short term plans are to complete and stabilize the presented workflow, to add support for polyphonic MIDI controllable instrument plugins, and to develop more features within the WAP GUI Builder that currently provides basic editing tools. The FAUST IDE itself needs to be extended to include sound file management[12], so that plugins using audio samples for instance could be implemented. To do that, we plan to expose more of the already C++ written architectures files on JavaScript side thanks to Emscripten[13]. This will also require to extend the FAUST remote compilation service. Deploying the resulting plugins in other host applications (like more traditional DAW running on the Web) should be straightforward if they comply with the WAP specification. Concluding tests have been conducted with the AmpedStudio DAW, for example.

---
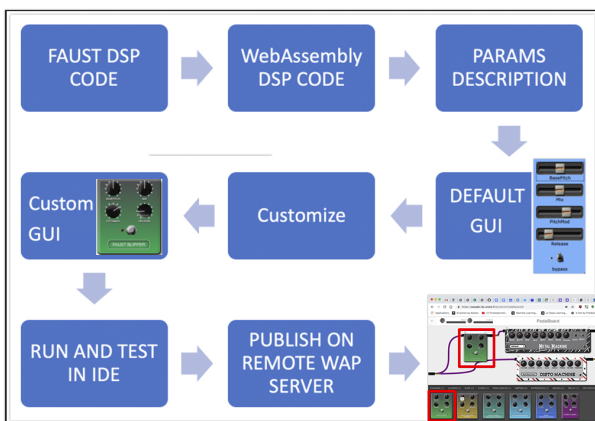
[11] Normally, there should be no bad surprises as the FAUST workflow generates valid WAPs. Examples/demos of online validators can be tested online, see for example https://jsbin.com/jeretab/edit?js,output
[12] Thats is handling the language 'soundfile' primitive which requires to implement a proper audio resources loading architecture
[13] C++ code using the libsndfile library can directly be compiled to WebAssembly and ported in JavaScript

Figure 6: workflow of the end-to-end design and implementation of a WebAudio plugin, from FAUST DSP code to a host application that uses the fully functional plugin with its GUI.

5. CONCLUSION

This paper presented the combined work of two teams deeply involved in the development of an audio DSP programming language and its complete ecosystem on the one hand, and the definition of a WebAudio plugin standard (WAP) and its complete surrounding environment on the other. Recent native to Web porting technologies like Emscripten and WebAssembly, as well as recognised Web standards (like WebComponents) have been heavily used. Combining client side and shared remote services is also part of the presented solution.

The complete workflow from the initial DSP source code, testing and running it in an integrated editor, polishing its user interface in another specialized GUI editor, to the finalized plugin running in an external host has been presented. Many examples of audio effects have been ported to WAPs directly by copying and pasting existing code from the Guitarix project[14], from the OWL pedal project[15], or from diverse open source projects, into the FAUST online IDE. Once compiled, the GUI has been customized within the GUI builder part of the IDE and published to remote WAP servers. Then, they can be tested online in the host web applications such as the pedalboard host presented in Figure 3[16].

Having the authoring tools as well as the deployment platform as pure Web applications facilitates the workflow and interoperability of the components. We also think that the presented toolchain could be adapted to other plugin formats or audio DSP production tools.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] M. Buffa and J. Lebrun. 2017. "Real time tube guitar amplifier simulation using WebAudio".In *Proc. 3rd Web Audio Conference* (WAC 2017). London, UK.
[2] M. Buffa and J. Lebrun. 2017. "Web Audio Guitar Tube Amplifier vs Native Simulations". In *Proc. 3rd Web Audio Conference* (WAC 2017). London, UK.
[3] N. Jillings and al. 2017. "Intelligent audio plug-in framework for the Web Audio API". In *Proc. 3rd Web Audio Conference* (WAC 2017). London, UK.
[4] M. Buffa, M. Demetrio, and N. Azria. 2016. "Guitar pedal board using WebAudio". In *Proc. 2th Web Audio Conference* (WAC 2016). Atlanta, USA.
[5] M. Buffa and al. 2017. "WASABI: a Two Million Song Database Project with Audio and Cultural Metadata plus WebAudio enhanced Client Applications". In *Proc. 3rd Web Audio Conference* (WAC 2017). London, UK.
[6] Y. Orlarey, D. Fober, and S. Letz. 2004 . "Syntactical and Semantical aspects of Faust". *Soft Computing* 8, 9 (2004). 623–632.

---

[14] https://guitarix.org/
[15] https://www.rebeltech.org/product/owl-pedal/
[16] https://wasabi.i3s.unice.fr/dynamicPedalboard/#

[7] S. Letz, Y. Orlarey, and D. Fober. 2018. "Faust Domain Specific Audio DSP Language Compiler to WebAssembly". In *Compagnion Proc ot the Web Conference, International World Wide Web Conferences Steering Committee, Lyon France 2018.*

[8] H. Choi and J. Berger. 2013. "WAAX: Web Audio API eXtension". *In Proc. International Conference on New Interfaces for Musical Expression (NIME'13)*. Daejeon, Korea.

[9] Kleimola, J. and Larkin, O. 2015. "Web Audio modules". In *Proc. 12th Sound and Music Computing Conference* (SMC 2015), Maynooth, Ireland.

[10] Larkin, O., Harker, A., and Kleimola J. "iPlug 2: Desktop Audio Plug-in Framework Meets Web Audio Modules". *4th Web Audio Conference* (WAC 2018), Berlin, Germany

[11] Buffa, M., Lebrun, J., Kleimola, J., Larkin, O. and Letz, S. "Towards an open Web Audio plugin standard". In *Companion Proceedings (Developer's track) of the The Web Conference 2018* (WWW 2018), Mar 2018, Lyon, France. 〈hal-01721483〉

[12] A. Zakai, "Emscripten: an LLVM to JavaScript compiler", In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications*, pages 301–312. ACM , 2011

[13] Roberts, C. "Strategies for Per-Sample Processing of Audio Graphs in the Browser". In *Proceedings of the Web Audio Conference* (WAC 2017). London, UK.

[14] Yi, Steven., Lazzarini, Victor., Costello, Ed "WebAssembly AudioWorklet Csound". *4th Web Audio Conference, TU Berlin*). (WAC 2018). Berlin, Germany.

[15] Letz, S., Orlarey, Y., Fober D., Michon R. "Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files "*In Proceedings* of *Linux Audio Conference*). (LAC 2017). St Etienne, France.

[16] Denoux, S.,Orlarey, Y., Letz, S., Fober D. "Composing a Web of Audio Applications" *1th Web Audio Conference, IRCAM, Mozilla Foundation*). (WAC 2015). Paris, France.

[17] Letz, S., Denoux, S., Orlarey, Y., Fober D. "Faust Audio DSP language on the Web" "*In Proceedings* of *Linux Audio Conference*). (LAC 2015). Mainz, Germany.