Using Faust DSL to Develop Custom, Sample Accurate DSP Code and Audio Plugins for the Web Browser

Shihong Ren, Stephane Letz, Yann Orlarey, Romain Michon, Dominique Fober, Michel Buffa, Jerome Lebrun

Using Faust DSL to Develop Custom, Sample Accurate DSP Code and Audio Plugins for the Web Browser[1]

SHIHONG REN,[1] STÉPHANE LETZ,[1] YANN ORLAREY,[1] ROMAIN MICHON,[1] DOMINIQUE FOBER[1]
renshihong@hotmail.com, (letz, orlarey, michon, fober)@grame.fr

MICHEL BUFFA,[2] AND JEROME LEBRUN[2]
(buffa, lebrun)@i3s.unice.fr

[1]GRAME, 11 cours de Verdun LYON, FRANCE
[2]Université Côte d'Azur, CNRS, INRIA, FRANCE

The development and porting of virtual instruments or audio effects on the Web is a hot topic. Several initiatives are emerging, from industry-driven ones (e.g., Propellerhead Rack Extension running on the Web[2]), to more community based open-source projects [1]. Most of them aim at adapting existing code bases (usually developed in native languages like C/C++) as well as facilitating the use of existing audio Digital Signal Processing (DSP) languages and platforms. Our two teams previously presented an open format for WebAudio Plugins coined WAP [2]. It aims at: (i) improving the interoperability of audio/MIDI plugins developed using pure Web APIs, (ii) porting existing native code bases, or (iii) using Domain Specific Languages (DSL). In this paper, we present a solution based around FAUST DSL, its Web-based editor, and the integration of a plugin GUI editor allowing to directly test, generate and deploy WAP plugins. We also evoke our collaborative work: one team hatching and improving FAUST, the other working on the recreation of tube guitar amplifiers and pedalboards within Web browsers. So as to fully illustrate the FAUST online framework, a case study is detailed with complete workflow, from the FAUST DSP source code written and tested in a fully functional online editor, to a self-contained plugin running in a separate host application.

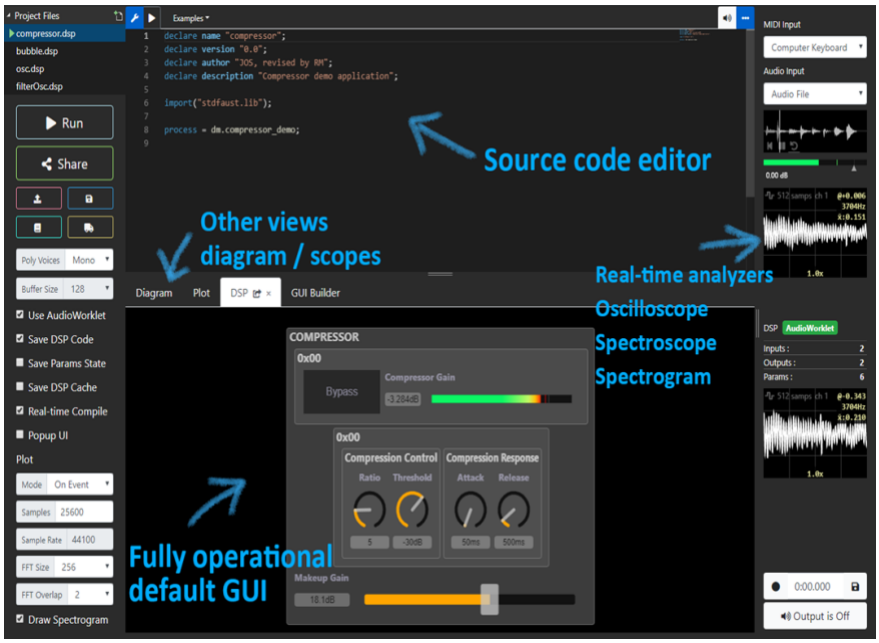

Figure 1: the FAUST IDE provides many embedded tools: oscilloscopes, spectroscope and spectrogram analyzer, functional default GUI, schema preview, etc.

[1] This is the Author's Accepted Manuscript. This manuscript has been accepted for publication on 2 September 2020: S. Ren, S. Letz, Y. Orlarey, R. Michon, D. Fober, M. Buffa, and J. Lebrun, "Using Faust DSL to Develop Custom, Sample Accurate DSP Code and Audio Plugins for the Web Browser, "J. Audio Eng. Soc., vol. 68, no. 10, pp. 703-716, (2020 October.). Please refer to the published version here: `https://www.aes.org/e-lib/browse.cfm?elib=20987`

[2] https://www.reasonstudios.com/press/275-reasons-flagship-europa-synth-now-available-as-a-plugin-for-other-daws-and-on-the-web -- All URLs were verified on June 4, 2020.

1. INTRODUCTION

There are many ways to develop software with the WebAudio API today. In pure JavaScript, the `genish.js` environment for instance [3] allows for the development of sample-level audio processing algorithms.[3] C/C++ written code can be transpiled to WebAssembly using Emscripten [4] and wrapped by additional JavaScript code to become ready-to-use audio nodes. Domain Specific Languages (DSL) for programming DSP algorithms that also compile to WebAssembly, like the mature Csound [5] with its set of WebAudio examples,[4] or the recently announced SOUL DSP language with its playground [5] can be used. 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 standards are currently available, including Steinberg's *VST* (Virtual Studio Technology, created in 1997 by Cubase creators), Apple's *Audio Units* (Logic Audio, GarageBand), Avid's *AAX* (ProTools creators), and *LV2* from the Linux audio community. Although the APIs offered by these formats are different, they share a common goal: to implement instruments or audio effects, and to allow a host application to load them.

In the first years after the birth of WebAudio (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 like SoundTrap, BandLab, or AmpedStudio, it became 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.

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 and identified by URIs) should also be available for plugins written in different ways. To this end, some initiatives have been proposed [6,7] and with other groups of researchers and developers we made in 2018 a proposal for a WebAudio plugin 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[6].

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 tools to validate the WAP standard: a virtual guitar "pedalboard" that discovers plugins from several remote repositories, and allows musicians 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 (see Fig. 1).[7] As of last year, WAP now includes support for pure MIDI plugins (a GM midi synthesizer, virtual midi keyboards, a MIDI event monitoring plugin, etc[8]). For more details about the WAP proposal, and how it is related to other approaches like Web Audio Modules (WAMs), WebAudio API eXtension (WAAX) [8], or JavaScript Audio Plugin (JSAP), see [2].

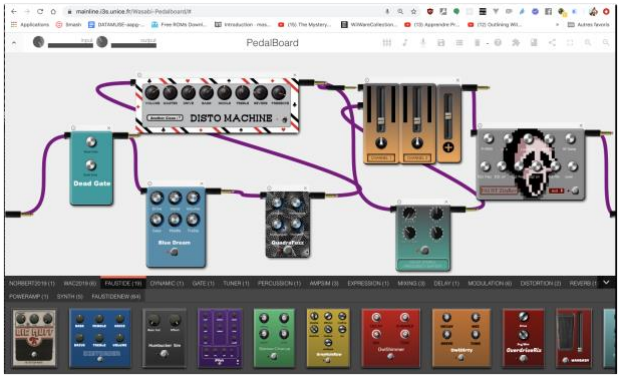

Figure 2: The virtual pedalboard host application scans multiple remote WAP plugin servers. WAP plugins can then be dragged and dropped and assembled in a graph.

This paper is a further extended, fully revamped version of an awarded presentation that was given during the WebAudio Conference 2019 [9]. Our goal is here to highlight and detail some of the results presented, to introduce new perspectives and to flesh out newer developments. For example, the compilation chain of the Faust code on remote servers will be fully characterized and the GUI Builder will have a full section detailing a "practical case" of design with the full recreation of a novel tube guitar amplifiers PowerAmp stage simulation using a Faust approach. Also a complete user evaluation has been conducted and will be detailed with the obtained results analyzed.

In the next sections, we will focus on the new online IDE we developed, that is well suited for coding, testing, and publishing WAP plugins written in FAUST, directly in a Web browser. The IDE includes a graphical interface editor allowing 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

[3] http://www.charlie-roberts.com/genish/
[4] https://waaw.csound.com
[5] https://soul.dev
[6] https://github.com/micbuffa/WebAudioPlugins

[7] Videos presenting the results of this work can be found online: https://www.youtube.com/watch?v=pe8zg8O-BFs.
[8] See the midi folder in the github repository of the WAP SDK, video https://www.youtube.com/watch?v=jHftK3YxcjQ

W3C WebComponents and published on remote WAP plugin servers. The plugins will then be directly usable by any compatible host software, using their URLs. You can picture the WAP plugins as images in an HTML document, their URL is sufficient, and can be dynamically retrieved using APIs from a remote Web Service.

2. BACKGROUND CONTEXT AND TERMS

FAUST [10] is a functional, synchronous, domain specific programming language working at the sample level, designed for real time audio signal processing and synthesis.

FAUST programs can be efficiently compiled to a variety of target programming languages, from C++ to WebAssembly. 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 (e.g., for and while loops, if statements, 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 (typically in C++ or JavaScript) 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 [11].

Several prior developments have been done to use the language on the Web platform. By adding an asm.js[9] 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 [12,13].

With the rise of the more stable and better designed WebAssembly[10] format in 2017 (a portable binary-code format for executable programs, firstly to be used on the Web, but also on native environments) as a replacement of asm.js, the previous work done with asm.js was 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") [14]. When embedded in the FAUST compiler running on the Web, they allow us to dynamically compile FAUST DSP programs as pure Web applications. Additional JavaScript glue code is added to transform DSP modules in fully functional WebAudio nodes.

The node generated by the FAUST compiler can be an AudioWorklet or a ScriptProcessor. These nodes are connectable with other WebAudio nodes to create an audio processing graph, and are designed to allow the development of custom low-level DSP algorithms. The AudioWorklet currently becomes a standard in the WebAudio API as it allows the computation of audio buffers in a dedicated audio thread (different from the "main thread" where the GUI is being executed). An AudioWorklet is composed of a JavaScript wrapper called the *AudioWorkletProcessor* that communicates with the *AudioWorkletNode* [15].

FAUST also allows us to circumvent many important limitations of the WebAudio standard, like the buffer size issues, as loops appearing in the graph of connected WebAudio nodes always introduce a delay of at least one buffer (128 samples), and thus one sample delay recursive algorithms cannot be expressed. This limitation was encountered in our previous works on the implementation of feedback loops in the signal chain using pure JavaScript high-level WebAudio nodes. To illustrate how FAUST-based solutions fulfil this important need for temporally accurate solutions, this practical case study will be detailed in Section 4.

3. CURRENT STATE

3.1 The New FAUST Online Web IDE

The compiler module generated by Emscripten was previously implemented in the FAUST IDE[11] 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 toolchain is now used to ensure the efficiency and the compatibility of the wrapper to transform it into another JavaScript UMD[12] module which can be imported and used in either a Webpage's runtime or a JavaScript compiling environment.

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

[9] http://asmjs.org/
[10] https://webassembly.org
[11] https://faustide.grame.fr
[12] https://github.com/umdjs/umd

We added some new features to the module:

- *A virtual file system (VFS)*: 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 for libraries and imported source code locally, or generates DSP code for other targets/architectures. For instance, FAUST block diagrams in SVG[13] format 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.

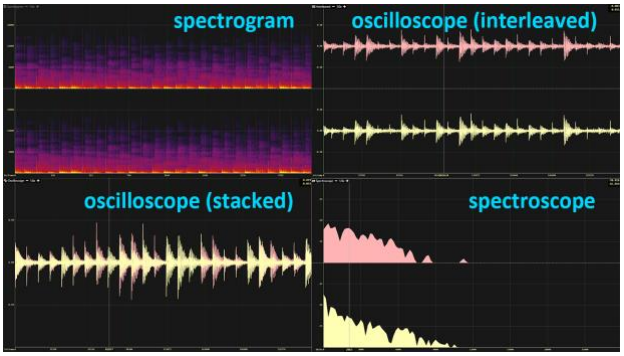

Figure 3: Signal visualizations: oscilloscopes, spectroscope and spectrogram analyzer provided by the FAUST IDE.

While a DSP developer needs to hear the sound produced by the algorithm, he should also be able to test it with other audio inputs, precisely display its time and frequency domain representation, etc. So, we emerged visualization, testing, and debugging tools such as MIDI/audio input simulation and output recording around the FAUST code editor to provide an integrated environment for DSP designing. This is the reason why we built a new online IDE (see Fig. 1) that provides more information and details on the generated DSP through graphical representation in a Web page (e.g. real-time FAUST block diagram, signal visualizations).

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;
- All options and displays related to DSP runtime, such as MIDI, audio inputs, and quick signal probing are placed in the right sidebar;
- The remaining central zone 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 of the WAP plugin version of the code, usable in external host applications.

Besides UI improvements facilitating code editing and compilation, audio probes are an important addition to the new editor. We designed four modes of signal visualizations to help FAUST users to debug their DSPs: data table, oscilloscope (stacked and interleaved by channels), spectroscope, and spectrogram analyzer (see Fig. 3). To implement all four probe modes, precise sample values are needed. In the browser environment, we have two ways to get audio output samples.

The first approach consists of using WebAudio *AnalyserNode* with its integrated functions to get the real-time audio data in frequency and time domain:

```
getByteFrequencyData,
getByteTimeDomainData,
getFloatFrequencyData,
getFloatTimeDomainData
```

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

The second approach consists of getting the sample values directly with a callback in a FAUST DSP node. 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[14] for its high performance when compiled to WebAssembly.[15] Thus, we perform the

[13] https://www.w3.org/Graphics/SVG/
[14] https://github.com/j-funk/kissfft-js/
[15] https://github.com/j-funk/js-dsp-test/

FFT computation in the FAUST online editor with two overlaps using a Blackman window function.

The first approach is used in the implementation of the two scopes in the right sidebar as it can also probe the audio input. The second approach is used for the larger scope at the bottom. It is more flexible and it 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*:

- *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 standard personal computer, the editor is able to draw up to 1 million samples continuously without significant rendering lag.
- *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.

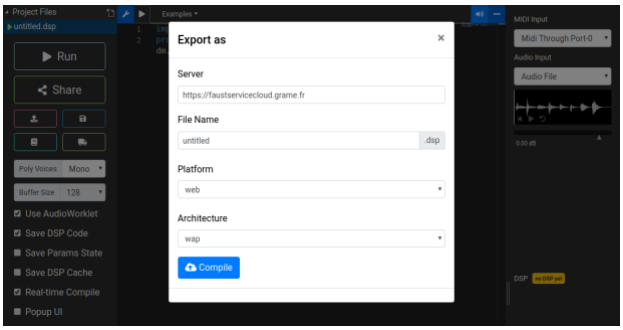

Figure 4: The export panel.

After a FAUST DSP is tested in the editor, users can export the DSP to different architectures including WebAudio Plugins (WAPs) which is compiled by an online compiling service (see Fig. 4). A dedicated GUI builder is integrated in the 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 publishing the plugin to a remote server. These points are detailed in the next two sections.

3.2 The Remote Compilation Service

In order to export and run FAUST programs outside the Web browser, a Compiler as a Service (CaaS) located at https://faustservicecloud.grame.fr/ has been designed, which can cross-compile executables binaries for most of the platforms supported by FAUST.

The service is accessible via the export button of the editor (see Fig. 4).

3.2.1 The Service API

The dialog between the editor and the remote compilation service is based on a very simple API (Application Programming Interface) of REST (Representational State Transfer) type which is detailed in the Appendix located at the end of this paper.

The first thing the editor does is to ask the server for a list of supported platforms (Windows, macOS, Web, ...) and architectures (VST, Max, ...). This information is needed in order to present possible choices to the user.
When the user requests an export for the first time, the FAUST code is sent to the compilation service which will check it, and if it is correct returns an SHA key which will represent the source code for all the following operations.

To trigger the compilation itself, and retrieve the binary code, a URL is dynamically forged by the editor from this SHA key, the platform, and the architecture. If the compilation goes well, a zip folder containing the binary code is downloaded from the server. The server maintains a cache, with a limited lifetime, to avoid unnecessary recompilations of the same code.

3.2.2 Implementation and Deployment

The server is written in C++ and based on `libmicrohttpd`,[16] a C library from the GNU Foundation allowing us to easily write a simple HTTP server.

It runs in a docker container built on top of Ubuntu 16.04 to which are added all the packages needed to compile and use FAUST, the FAUST distribution itself, all the needed SDKs, and several cross-compilers.

The server is almost exclusively limited to implementing the API we have just described. As for the FAUST compilation itself, it relies on a series of Makefiles, with precise naming conventions to perform all compilation tasks. This makes the service easily extensible just by adding Makefiles (i.e., without the need to modify it).

These Makefiles are organized in a two-level hierarchy, one folder per platform containing all the Makefiles related to that platform. By analyzing this folder hierarchy, the server can respond to the targets request seen above. The makefiles are pretty simple. They rely directly on the compilation tools of the FAUST distribution.

[16] https://www.gnu.org/software/libmicrohttpd/

When the POST of a file foo.dsp is processed, the SHA-1 key of the file is first calculated. This key is used as the name of a folder containing all the compilations related to this file. When afterwards a */<sha>/<platform>/<arch>/binary.zip* request is processed, it is first converted into a filename: *sessions/<sha>/<platform>/<arch>/binary.zip* to check if it has not already been generated. If it is not the case, the software will:

- Check that *makefiles/<platform>/Makefile.<arch>* exists
- Create the path *sessions/<sha>/<platform>/<arch>/*
- Copy *makefiles/<platform>/Makefile.<arch>* into *sessions/<sha>/<platform>/<arch>/Makefile*
- Cun the make command
- And return *session/<sha>/<platform>/<arch>/binary.zip*

The session folder thus serves as a cache and makes it easy to retrieve the compilation results. It is periodically emptied when the number of sessions becomes too large.

Figure 5: a default GUI is proposed from a FAUST code in the editor.

3.3 The GUI Editor

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

```
vslider(name+metadata,value,min,max,step)

Fuzz = vslider("Fuzz [style:knob]",
0.5, 0, 1, 0.01) : Inverted(1) :
si.smooth(s);

Level = vslider("Level [style:knob]",
default_level*2, 0, 1, 0.01) :
Inverted(1) : si.smooth(s);
```

Here, the code describes the definition of two parameters named “Fuzz” and “Level” along with some data defining the default value, min, max, step, unit type, etc. These parameters can be programmatically set/computed such as “default_level*2” in the second example, instead of using literal values.

As previously explained, the FAUST DSP code is compiled to a JavaScript wrapper and a WebAssembly module. This is all done on the 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 (see Fig. 5). It can be filled during the GUI edition process and used to generate the final GUI code (see Fig. 6). So far, we implemented only a generator for WebAudio plugins, using HTML/CSS/JavaScript code that complies with the W3C WebComponents specifications.[17]

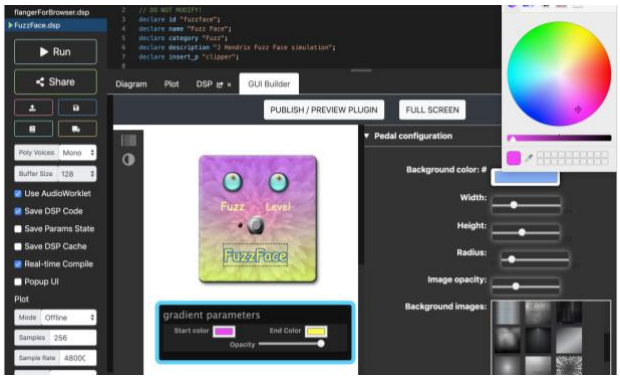

Figure 6: The default GUI can be customized: textures, knobs, sliders, switches positions size, appearance and labels, etc.

Figure 7: Other designs for the same DSP code.

At any time, the plugin (DSP + GUI) can be tested from within the IDE, without having 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 (Figs 6, 7 and 9 show 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

[17] 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.

saver, full sync between the FAUST code and GUI is planned as future enhancements.

The plugin can also be published on a remote plugin server, using standard Web services; this is shown in the life cycle workflow from Fig. 8. This allows us to preview it in its "running state" directly in the IDE (see Fig. 9).

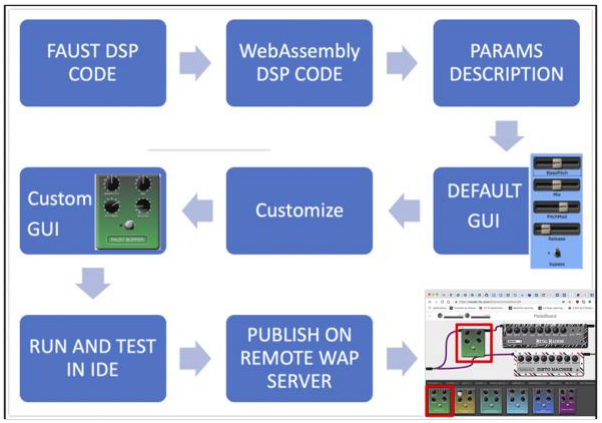

Figure 8: Workflow of the end-to-end design and implementation of a WebAudio plugin.

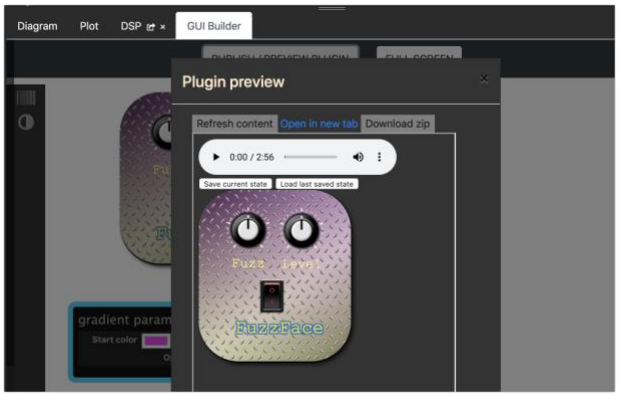

Figure 9: Other designs for the same DSP code

From this dialog, a WAP plugin generated by the FAUST IDE can also be downloaded as a zipped archive file that contains the DSP WebAssembly module, the standard JSON[18] WAP descriptor and the GUI code (HTML/CSS/JavaScript) 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 dynamically loaded and "unit tested" by different validation tools that come with the WAP SDK[19] (i.e., checked for compliance of their API to the specification, that the plugin is able to save/restore its state, etc.), or discovered by host web applications (e.g., DAWs, etc.)

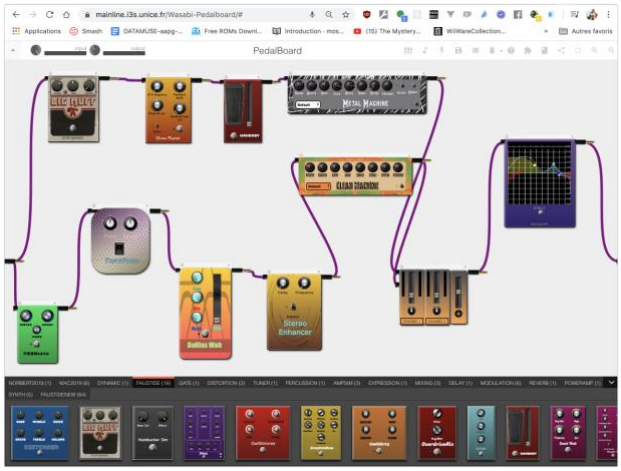

Figure 10: Host application example. The plugins set is discovered by requesting a remote WAP plugin server. Plugins are loaded dynamically when dragged and dropped in the main area.

One of these hosts is our "virtual pedalboard" Web application [16], a host for WAP plugins we developed to showcase the WAP standard (see Fig. 10), 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 IDE.[20]

4. A PRACTICAL USE CASE: MODELING THE POWER AMP STAGE OF A TUBE GUITAR AMP

Since 2015, we have been involved in the design, development and constant improvement of *perceptually* faithful simulations of existing tube guitar amplifiers like the Marshall JCM 800, used by many famous artists (e.g., AC/DC, Guns and Roses, Deep Purple, etc.), with a major constraint *that everything should run within just a web-browser* [17]. Our first versions were developed in pure JavaScript using the W3C WebAudio API and are available to play with.[21]

4.1 Limitations of the WebAudio API and high-level WebAudio nodes when processing feedback loops

Most tube guitar amplifiers have similar topology: a preamplifier stage, a so-called "tonestack" stage including bass, midrange, and treble controls, followed by a power stage (the Power Amp). The preamplifier beefs up the

[18] https://www.json.org/json-en.html

[19] Examples/demos of online validators can be tested online, see for example https://jsbin.com/jeretab/edit?js,output.

[20] An online tutorial explains how to create step by step a complete plugin, including its GUI and try it in the pedalboard host : https://tinyurl.com/wnw2yl7

[21] Demos on: https://mainline.i3s.unice.fr/AmpSim5/

high-impedance low-level signal coming from the guitar pickup microphones to a lower impedance mid-voltage signal that can drive the power stage. The preamplifier also shapes the tone of the signal; higher gain in the preamplifier leads to "overload", creating desirable crunch/distorted sounds. The power amplifier, with the help of the output transformer outputs a very low impedance, high current signal adequate to efficiently drive a loudspeaker and to produce loud amplified sounds. Another obvious and intended purpose of guitar amps is to produce interesting distorted sounds. For more details about all these stages, one can browse the AikenAmps website.[22]

In our previous implementations in JavaScript with the WebAudio API, the first stages were quite faithfully recreated (incl. harmonics and temporal dynamics) with very low latency [17,18,19]. However, the poweramp stage proved to be trickier to implement as it includes a Negative Feedback (NFB) loop. With pure JavaScript WebAudio API, major issues arise when dealing with loops, firstly due to the limitations imposed by its block-processing paradigm and also because of some divergences/bugs in how different browsers parse WebAudio graphs with loops. In the WebAudio API specs, the loops in a graph are required to include at least one delay node. Without this delay node, Firefox browser stops rendering the graph, while Chrome browser does not complain but adds, behind the scenes, a 3ms delay (the minimum size of an audio buffer being 128 frames, hence the minimal delay of 128/sampling rate: roughly 3ms at 44.1kHz). Now, to get a timewise precise implementation of loops, as the NFB with its simple RC networks induces delays usually much shorter than 3ms, a finer time precision at the level of some samples would be required to match this delay. With the current limitations, and quite un-consistently, this 3ms delay in the loop — so as to conform to the specs — brings also slightly different coloring of the amps between Firefox and Chrome.

4.2 Rewriting the Power Amp stage in FAUST, in order to bypass the above mentioned limitations

Our initial WebAudio implementations were based on high level nodes provided by the API: a bank of biquad filters in series (for the Presence in the NFB loop), a waveshaper node (for simulating tubes), gain nodes (master volume, negative gain in the NFB loop and for fine tuning), and a delay node (in the NFB loop). Consequently, the first step was recreating with FAUST our signal chain as faithfully as possible - i.e. using the same filters with the same parameters, the same transfer function, the same gain values, etc. Adapting FAUST filters to behave like WebAudio ones was not straightforward but after some interactions with the FAUST and WebAudio implementers, it proved to be easier to restart from the original C++ implementation of the WebAudio filter API (Chromium browser's source code). This porting done with the FAUST team ended up in a so-called `webaudio.lib` (available now in the FAUST distribution). To properly simulate the tube stages, we looked for any prior art available in FAUST (e.g., in `guitarix.lib` from the Guitarix project[23]), or waveshapers (as done by Oleg Kapitonov[24] in his plugins Pack or by Nick Thompson of Creative Intent for his Temper Distortion plugin[25]). Most tube simulations, like the Guitarix tube simulations, relied heavily on C/C++ code, making it quite difficult to run them in Web browsers as the FAUST toolchain does not support yet hybrid FAUST/C source code with compilation target set to WebAudio/WebAssembly.

Luckily, the "Temper Distortion" simulation included a 100% FAUST-based implementation of a simple but efficient waveshaper that produced a warm, adjustable, controllable distortion that would easily fit our needs. We used it as a starting point for further developments. The code included the implementation of a simple transfer function whose curve could be adjusted (i.e., leading to more subtle/less aggressive curves). The transfer function from the Temper Distortion waveshaper code[26] is based on a parameterized *tanh* function :

$$transfer(x) = tanh(k \cdot x) \,/\, tanh(k)$$

here with $tanh(x) \simeq x \cdot (27 + x^2) \,/\, (27 + 9 \cdot x^2)$

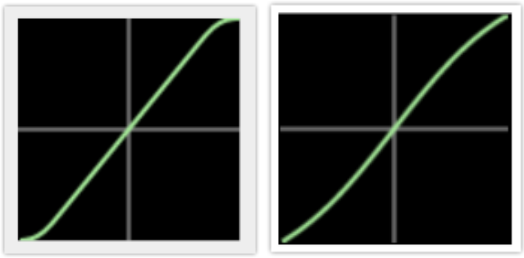

Figure 11: Left, the transfer function for our JS implementation, Right, the one from the FAUST implementation[27] with k=1.

This is very close to the one we used in our JavaScript implementation (see Fig. 11 for comparison) with parameter *k* driving the S-shape of the curve.

In this way, our current JavaScript poweramp implementation (made of a dozen of high level WebAudio nodes) was replaced by a simple FAUST generated AudioWorklet node.[28] Figs 12 and 13 show the final diagram of the FAUST implementation of this poweramp based on the Temper Distortion source code.

[22] More details on: http://www.aikenamps.com/index.php
[23] http://guitarix.org/
[24] https://github.com/olegkapitonov/Kapitonov-Plugins-Pack
[25] https://github.com/creativeintent/temper
[26] See http://www.musicdsp.org/showone.php?id=238

[27] Check our online comparison tool: https://jsbin.com/qifexor/edit?js,console,output,
[28] We did that in the past by replacing the tonestack stage by some FAUST implementations [17].

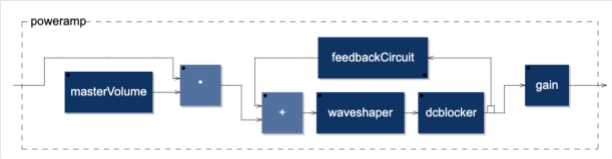

Figure 12: Diagrams of the final implementation.

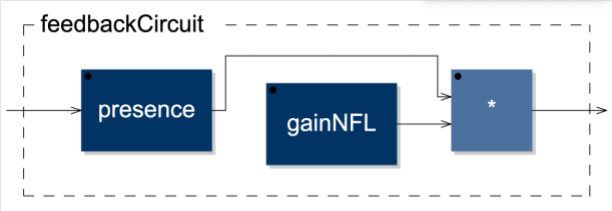

Figure 13: The feedback circuit. The presence filter is a set of peaking filters ported in FAUST from the WebAudio API implementation.

Major differences are the introduction of the Presence control (made of two peaking filters, at 2kHz and 4kHz) in the feedback loop, the introduction of an adjustable NFB gain and the removal of some unnecessary elements (like the now pointless resonant lowpass filter at the input).

4.3 Generating the graphical interface, publishing and integrating the plugin in a host software, testing the plugin in the amplifier simulation chain

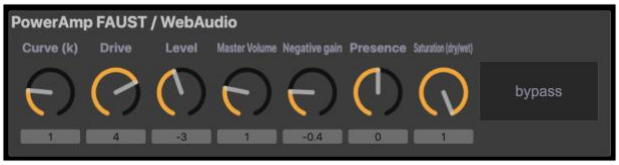

Figure 14: GUI generated by the FAUST IDE, some extra parameters (negative feedback gain, waveshaper; drive, curve, saturation) are tweakable, enabling fine tuning of the NFB loop.

Finally, we added some GUI elements (knobs) in order to allow for the fine-tuning in real-time of the different parameters (see Fig. 14), in particular the ones that control the waveshaper (i.e., drive, curve, and distortion), the NFB gain, and the Presence filters.

The current "Temper-inspired" implementation of the waveshaper does not rely on pre-calculated point tables, but on a transfer function applied to each sound sample. Obviously, this leaves some room for further optimization. The dynamic time response of the tubes is obtained using a simple amplitude-follower before the waveshaper to drive an allpass filter, altering the DC offset, and thus the slope of the curve. This simple approach would benefit from being fully confronted with our previous JavaScript implementation where both the slope and the S-shape were dynamically changed.[29]

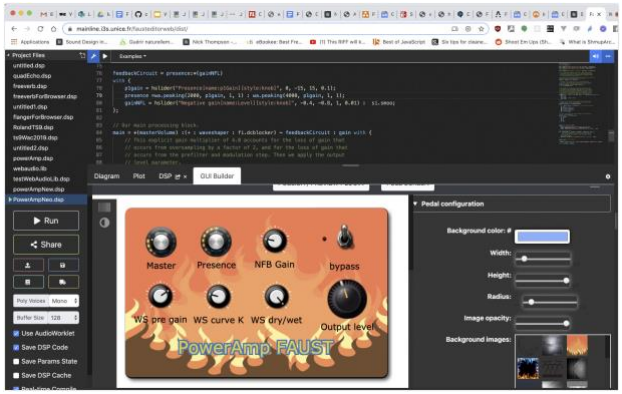

Figure 15: For testing purposes, we created a WebAudio plugin from the FAUST code, using the WAP GUI Builder we developed, integrated in the FAUST IDE.

This FAUST poweramp proved to be fully functional and adjustable, so we could proceed to the evaluation part. The first step of the evaluation were careful measurements and critical listening to assess the dynamical and harmonic possible shortcomings of this FAUST implementation in standalone mode, tweaking the different parameters (i.e., master volume, presence, NFB gain, presence, transfer function parameters), and also comparing its behavior with our WebAudio/JavaScript implementation. Either with dry guitar sound samples or real guitars as inputs, the Master volume and Presence controls reacted quite similarly in both implementations. Again, the typical oscillatory effects (Larsen) we got when going towards positive feedback within our JavaScript implementation were similar, slightly more controlled, in the FAUST version when pushing some parameters (Presence and NFB gain) close to their limit values.

4.4 Evaluation, latency measures

The FAUST IDE allowed also for the creation of a bona-fide WebAudio plugin from the FAUST implementation of the PowerAmp (see Fig. 15). Inside our virtual pedalboard, we chained a special version of our JavaScript tube guitar amplifier simulation with bypassed power amp and cabinet simulation stages, and replaced the last stage with our novel FAUST-based poweramp stage (see Fig. 16), and finally we compared it with the full featured, JavaScript based simulations. These results can be seen/heard in a video we published online,[30] using the online pedalboard WebAudio application.[31]

[29] Simulation of tube amp Sag/Squish: https://youtu.be/zBhn7odezUQ

[30] https://youtu.be/uNp-0hzveeo

[31] https://mainline.i3s.unice.fr/Wasabi-Pedalboard/#, check the PowerAmp tab at bottom, drag'n'drop in the main area.

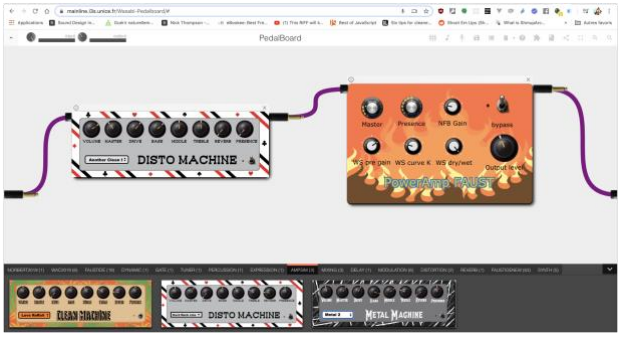

Figure 16: The PowerAmp plugin in our pedalboard application, with a version of our AmpSim in which we bypassed the JS PowerAmp and cabinet simulator stages.

The differences in terms of sound/timbre and playing dynamics proved to be very small and subtle. However, we noticed that the FAUST implementation was much more stable and versatile when pushing the internal parameters of the feedback loop. When pushed towards positive feedback loops, the FAUST poweramp produced cleaner oscillations with less intermodulation distortion suggesting a smaller latency from the NFB loop.

The last step was to get a closer look at the behavior of the NFB loop whose delay ought to be lower than the undesired 3ms lower limit in our previous JavaScript implementation (due to block-based processing imposing delays of 128 samples in the back-fed signal). In the FAUST implementation, the measurement tools (Fig. 17) proved the sample-wise nature of the processing with a delay of just one-sample for the NFB/Presence loop.[32] This also explains the increased stability of this loop. Now, in terms of aggregated latency for the Power Amp, we did measurements of the "end-to-end" latency, from guitar to cabinet and obtained consistently better values for latency with the new FAUST implementation: around 20-21 ms compared to the 23-24 ms latency of our previous finely-tuned JavaScript implementation (both using a Firefox Nightly 75.0a1 browser with an external Focusrite Scarlett and a Macbook Pro 16 under 10.14). This confirms a saving of 3ms in accordance with the difference of processing of loops between FAUST (sample-wise) and WebAudio API/JavaScript (block based).

To conclude, we have been able, using only the online FAUST IDE, to completely redesign our Power Amp stage, to fine tune it using the embedded measurement tools, to build its GUI and to thoroughly test the final version before publishing it as a plugin for Web-based host applications.

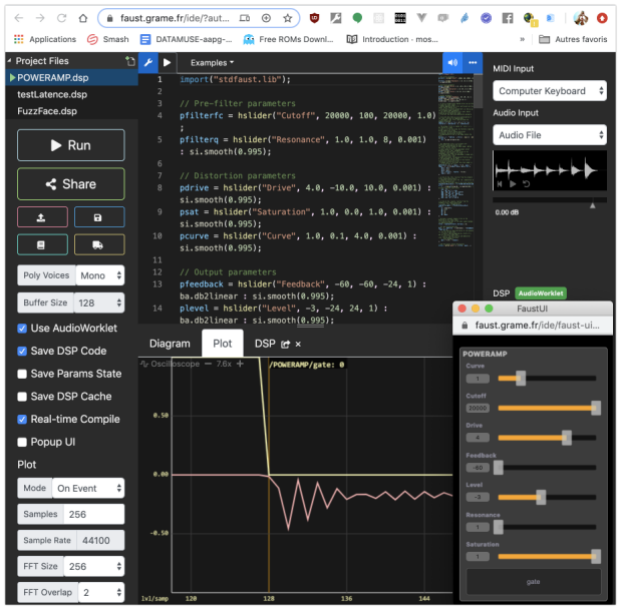

Figure 17: The PowerAmp plugin latency can be measured achieving sample-wise accuracy using the tools embedded in the IDE. Yellow: a gate signal, Pink: the output from the PowerAmp. X-axis is in samples.

5. DISCUSSION

Now, the short term goals of the authors 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, so that plugins using audio samples/wavetables, for instance could be implemented. To do that, thanks to Emscripten, we plan to expose more of the already written C++ architecture files used to access sound resources on the JavaScript side.[33] 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. For example, some concluding tests have already been conducted with the AmpedStudio DAW.[34]

In addition, we asked a group of six audio plugin developers, with different level of expertise with the FAUST language, to follow a tutorial[35] that guided them through the creation and publication of a phaser effect only using the IDE. They did this exercise remotely and after the completion of the proposed tasks they answered an online form[36] where we asked them to evaluate their experience. These questions started by "did you manage

[32] As measured in the FAUST IDE, using `process = button("gate") <: ((poweramp), _);` style code and the embedded visualisation tools.

[33] C++ code using the `libsndfile` library can be directly compiled to WebAssembly and ported to JavaScript.

[34] https://ampedstudio.com/

[35] Tutorial "Create your own WebAudio Plugins": https://tinyurl.com/wnw2yl7

[36] Evaluation form (french): https://tinyurl.com/y9vny8qu

to create, publish and test the created plugin in an online host?", and included ratings and comments of different aspects.

The results are summarized below:

- 100% of the developers managed to create a plugin (including its GUI), to publish and to test it,
- Average time to complete the tasks: 37 minutes,
- 83.3% of them used the visualization tools and considered them an important addition to help understanding the DSP code behavior,
- 100% appreciated the fact that a default GUI is generated automatically. They found it useful both for testing the plugin during development and for prototyping its ergonomics.
- 100% managed to customize the GUI as suggested in the tutorial (change size, position, colors, textures, shape of elements in the GUI)
- 100% tried to do things that are not yet supported in the GUI builder (change a knob into a slider or into a push button, change default value for knobs and sliders, change the behavior of the bypass switch). All these things are possible but they require changing the source code.
- 100% found the tool very useful for rapid audio plugin development/prototyping.

The questionnaire also asked to suggest improvements. Here many proposed to improve the overall ergonomics of the GUI builder ("the most important features should be presented in a more compact way", "where is the undo button?", "the dimensions in pixels should be always visible") and complained about the lack of a magnetic grid or alignment tools. Actions like changing a slider into a knob or changing the default parameter values from the GUI builder are also missing and have been asked by all users (while this can be done by changing the source code of the plugin parameter declarations). The GUI builder is still at an early stage and a more comprehensive user needs analysis will be conducted, focusing mainly on the ergonomics of the GUI builder. This first version however, has been considered as "very useful" for prototyping rapidly an audio plugin, prior to polishing its GUI by hand, editing the HTML/CSS code.

6. CONCLUSION

In this paper, we 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. We made extensive use of recent technologies like Emscripten as well as recognized Web standards (like WebComponents) to achieve the porting of native source components to higher performance versions using WebAudio and WebAssembly API. 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[37] and the OWL pedal project,[38] or from various open-source projects, into the new FAUST online IDE. Once compiled, the GUI has been customized within the GUI builder part of the IDE and published to remote WAP servers. They can now be tested online in host web applications such as our Virtual Pedalboard host presented in Figs 2, 10 and 17.[39]

Having the authoring tools as well as the deployment platform as pure Web applications facilitates the workflow and interoperability of these components [20]. We also think that the presented toolchain could be easily adapted to other plugin formats or audio DSP production tools.
Source code for the FAUST and IDE projects is located at:

- https://github.com/grame-cncm/faust
- https://github.com/grame-cncm/faustide

7. ACKNOWLEDGMENTS

This work was partially supported by the French Research National Agency (ANR) and the WASABI [21] team (contract ANR-16-CE23-0017-01). The authors would particularly like to thank Jordan Sintes, Guillaume Etevenard and Elmahdi Korfed for their contributions to the WAP standard.

8. REFERENCES

[1] O. Larkin, A. Harker, and J. Kleimola, "iPlug 2: Desktop Audio Plug-in Framework meets Web Audio Modules," presented at the *4th Web Audio Conference* (WAC'18), Berlin, Germany, (2018 Sep.). ISSN 2663-5844

[2] M. Buffa, J. Lebrun, J. Kleimola, O. Larkin, and S. Letz. "Towards an Open Web Audio Plugin Standard," presented at *The Web Conference 2018* (WWW2018), Lyon, France, (2018 Apr.). doi:10.1145/3184558.3188737

[3] C. Roberts, "Strategies for Per-Sample Processing of Audio Graphs in the Browser," presented at the *Web*

[37] https://guitarix.org/
[38] https://www.rebeltech.org/product/owl-pedal/
[39] https://wasabi.i3s.unice.fr/dynamicPedalboard/#

Audio Conference (WAC'17). London, UK, (2017 Aug.). ISSN 2663-5844

[4] A. Zakai, "Emscripten: an LLVM to JavaScript Compiler," presented to the *ACM Int. Conf. Object Oriented Programming Systems Languages and Applications* (OOPSLA'11), Portland, OR, USA, (2011 Oct.). doi: 10.1145/2048147.2048224

[5] S. Yi, V. Lazzarini, and Ed. Costello, "WebAssembly AudioWorklet Csound," presented at the *4th Web Audio Conference* (WAC'18). Berlin, Germany, (2018 Sep.). ISSN 2663-5844

[6] N. Jillings, Y. Wang, R. Stables, and J.D. Reiss, "Intelligent Audio Plug-in Framework for the Web Audio API," presented at the *3rd Web Audio Conference* (WAC'17), London, UK, (2017 Aug.). ISSN 2663-5844

[7] J. Kleimola and O. Larkin, "Web Audio Modules," presented at the *12th Sound and Music Computing Conf.* (SMC2015), Maynooth, Ireland, (2015 Jul.). ISBN-13: 978-0992746629

[8] H. Choi and J. Berger, "WAAX: Web Audio API eXtension," presented at the *International Conference on New Interfaces for Musical Expression* (NIME'13), Daejeon, Korea, (2013 May). doi:10.5281/zenodo.1178494

[9] S. Letz, S. Ren, Y. Orlarey, R. Michon, D. Fober, E. Aamari, M. Buffa, and J. Lebrun, "FAUST online IDE: Dynamically compile and publish FAUST Code as WebAudio Plugins," presented at the *5th Web Audio Conference* (WAC'19), Trondheim, Norway, (2019 Dec.). ISSN 2663-5844

[10] Y. Orlarey, D. Fober, and S. Letz, "Syntactical and Semantical Aspects of Faust", *Soft Computing*, vol. 8, (2004). doi:10.1007/s00500-004-0388-1

[11] S. Letz, Y. Orlarey, D. Fober, and R. Michon, "Polyphony, Sample-accurate Control and MIDI Support for FAUST DSP using Combinable Architecture Files," presented at the *Linux Audio Conference* (LAC'2017), St Etienne, France, (2017 May).

[12] S. Denoux, Y. Orlarey, S. Letz, and D. Fober, "Composing a Web of Audio Applications," presented at the *1st Web Audio Conference* (WAC'15). Paris, France, (2015 Jan.). ISSN 2663-5844

[13] S. Letz, S. Denoux, Y. Orlarey, and D. Fober. "Faust Audio DSP Language on the Web," presented at the *Linux Audio Conference* (LAC'2015), Mainz, Germany, (2015 Apr.).

[14] S. Letz, Y. Orlarey, and D. Fober, "Faust Domain Specific Audio DSP Language Compiler to WebAssembly," presented at *The Web Conference* (WWW2018), Lyon, France, (2018 Apr.). doi:10.1145/3184558.3185970

[15] H. Choi, "AudioWorklet: The Future of Web Audio," presented at the *International Computer Music Conference* (ICMC'18), Daegu, South Korea, (2018 Aug.).

[16] M. Buffa, M. Demetrio, and N. Azria, "Guitar Pedal Board using WebAudio," presented at the *2th Web Audio Conference* (WAC'16), Atlanta, USA, (2016 Apr.). ISSN 2663-5844

[17] M. Buffa and J. Lebrun, "Real Time Tube Guitar Amplifier Simulation using WebAudio," presented at the *3rd Web Audio Conference* (WAC'17), London, UK, (2017 Aug.). ISSN 2663-5844

[18] M. Buffa and J. Lebrun, "Web Audio Guitar Tube Amplifier vs Native Simulations," presented at the *3rd Web Audio Conference* (WAC'17), London, UK, (2017 Aug.). ISSN 2663-5844

[19] M. Buffa and J. Lebrun, "Real-Time Emulation of a Marshall JCM 800 Guitar Tube Amplifier, Audio FX Pedals, in a Virtual Pedal Board," presented at *The Web Conference 2018* (WWW2018), Lyon, France, (2018 Apr.). doi:10.1145/3184558.3186973

[20] M. Buffa, J. Lebrun, S. Letz, Y. Orlarey, R. Michon, D. Fober, and S. Ren, "Emerging W3C APIs opened up Commercial Opportunities for Computer Music Applications," presented at *The Web Conference 2020* (WWW2020), Taipei, Taiwan, (2020 Apr.).

[21] G. Meseguer-Brocal, G. Peeters, G. Pellerin, M. Buffa, E. Cabrio, C. Faron-Zucker, A. Giboin, I. Mirbel, R. Hennequin, M. Moussallam, F. Piccoli, and T. Fillon, "WASABI: a Two Million Song Database Project with Audio and Cultural Metadata plus WebAudio enhanced Client Applications," presented at the *3rd Web Audio Conference* (WAC'17), London, UK, (2017 Aug.). ISSN 2663-5844

APPENDIX: Remote Compilation Service API

The dialog between the editor and the remote compilation service (workflow in Fig. below) is based on a very simple API (Application Programming Interface) of REST (Representational State Transfer) type that we will detail here:

`GET /targets`

The GET https://faustservicecloud.grame.fr/targets allows for the querying of the service on supported platforms and architectures. The response is in json format, organized in platforms, with a list of supported architectures for each platform. Here is a simplified example of a response:

```
{
"ios": ["ios","ios-osc"],
"web": ["wap","wap-poly",...],
"android": ["android","smartkeyb",...],
...
}
```

Three platforms are indicated here: ios, web and android with a list of available architectures for each of them. This information is then used to forge compilation requests by indicating the desired platform and architecture, for example web/wap.

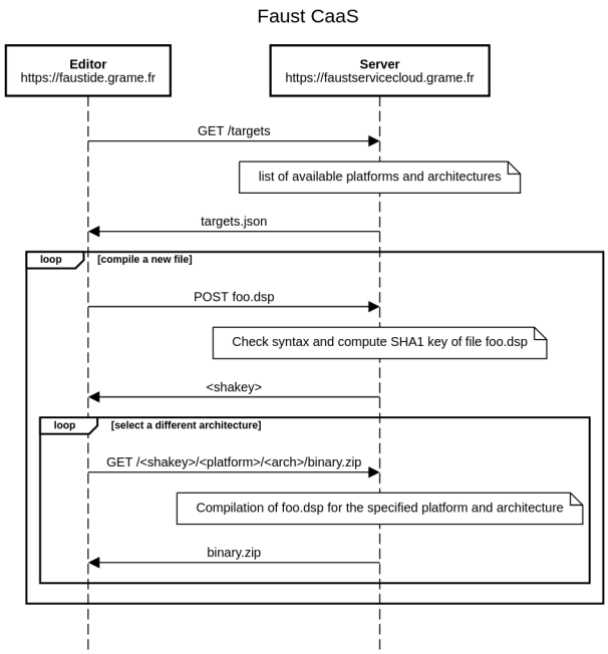

Figure: Communication protocol between the editor and the remote compilation service.

`POST /`

The POST request https://faustservicecloud.grame.fr accompanied by a file foo.dsp allows to transfer this file to the compilation service. In response, it receives a 160-bit SHA-1 key, for example cf55531c580cc7d3485a5161259f0571d3e6bdef, which uniquely identifies the contents of the file and will be used to represent this file in some requests.

`GET /<sha>/<platform>/<arch>/binary.zip`

The GET request */<sha>/<platform>/<arch>/binary.zip* launches the compilation of a source file (previously posted and represented by its SHA-1 key) for the specified platform and architecture. The result is rendered as a compressed folder that always has the same name: binary.zip.

`GET /<sha>/<platform>/<arch>/binary.apk`

The GET request */<sha>/<platform>/<arch>/binary.apk* is identical to the previous one but retrieves an .apk (Android Package) folder that can be directly installed on Android phones.

`GET /<sha>/<platform>/<arch>/precompile`

The GET request */<sha>/<platform>/<arch>/precompile* launches the compilation of a source file (previously posted and represented by its SHA-1 key) for the specified platform and architecture, but without downloading the result. It allows to know the readiness of the compilation result (depending on the architecture, it can take several minutes) and only then will it trigger a download request.

`GET /<sha>/diagram/process.svg`

The GET request */<sha>/diagram/process.svg* launches if the representation of a source file (previously posted and represented by its SHA-1 key) as a block diagram in SVG graphical format is needed. It returns the main process.svg diagram (the diagram can be composed of several files, but process.svg is always the entry point).

`POST /compile/<platform>/<arch>/binary.zip`

The POST */compile/<platform>/<arch>/binary.zip* combines in a single query the transfer of the file to be compiled and the retrieval of the compilation result. This request is useful when the compilation service runs on a server in stateless mode, as with the new “cloud run” service from Google.

THE AUTHORS


Shihong Ren


Stéphane Letz


Yann Orlarey


Dominique Fober


Romain Michon


Michel Buffa


Jerome Lebrun

Shihong Ren is a composer/researcher in computer music. He entered the Conservatoire national supérieur musique et danse de Lyon in 2011 in the electroacoustic composition class, and graduated in 2016 as the youngest DNSPM and Master Degree owner in the composition major. He got the Artist Diploma in 2018, and followed the cursus of composition in IRCAM in the same year. He attended an internship at GRAME-CNCM (Lyon, France) in 2019.

●

Stéphane Letz is a researcher in Computer Music at GRAME-CNCM (Lyon, France), a member of the Audio Working Group of the W3C, and a specialist in real-time audio architectures and music protocols. He is the designer of Jack2, the popular low latency audio server. He is also one of the main developer and maintainer of the FAUST compiler.

●

Yann Orlarey is the scientific director of GRAME-CNCM (Lyon, France). His research work focuses on the design and implementation of programming languages for musical and sound creation, with a particular interest in lambda-calculus, functional programming, and real-time and compilation techniques. Yann Orlarey and his colleagues at Grame are the designers of FAUST, a functional programming language for sound synthesis and audio processing with a strong focus on the design of synthesizers, musical instruments, audio effects, etc.

●

Dominique Fober is a researcher in Computer Music at GRAME-CNCM (Lyon, France). His research is concerned mainly with software architecture for real-time music systems, languages for musical composition, and music notation and representation systems.

●

Romain Michon is a full-time researcher at GRAME-CNCM (Lyon, France) and a researcher and lecturer at the Center for Computer Research in Music and Acoustics (CCRMA) at Stanford University (USA). He has been involved in the development of the FAUST programming language since 2008. Beside that, Romain's research interests involve embedded systems for real-time audio processing, Human Computer Interaction (HCI), New Interfaces for Musical Expression (NIME), and physical modeling of musical instruments.

●

Michel Buffa is a professor/researcher at University Côte d'Azur, a member of the WIMMICS research group, common to INRIA and to the I3S Laboratory (CNRS). He contributed to the development of the WebAudio research field, since he participated in all WebAudio Conferences, being part of each program committee between 2015 and 2019. He actively works with the W3C WebAudio working group. With other researchers and developers, he co-created the WebAudio Plugin standard.

●

Jerome Lebrun is a full-time CNRS researcher at University Côte d'Azur, heading the Biomedical Signal Processing group of the I3S laboratory. His current interests and research fields include signal processing for computer music, ciphered speech secure communications, multimodal EEG/NIRS/f-MRI acquisitions and the analysis of animal vocalizations