Web Audio Modules 2.0: An Open Web Audio Plugin Standard

Michel Buffa
buffa@univ-cotedazur.fr
University Côte d'Azur, CNRS, INRIA
France

Shihong Ren
shihong.ren@univ-st-etienne.fr
Shanghai Conservatory of Music
Université de Saint-Étienne, ECLLA
China, France

Owen Campbell
owen.campbell@gmail.com
webaudiomodules.org
USA

Tom Burns
tom@burns.ca
sequencer.party
Canada

Steven Yi
stevenyi@gmail.com
webaudiomodules.org
USA

Jari Kleimola
Oliver Larkin
{jari,oli}@webaudiomodules.org
Finland, UK

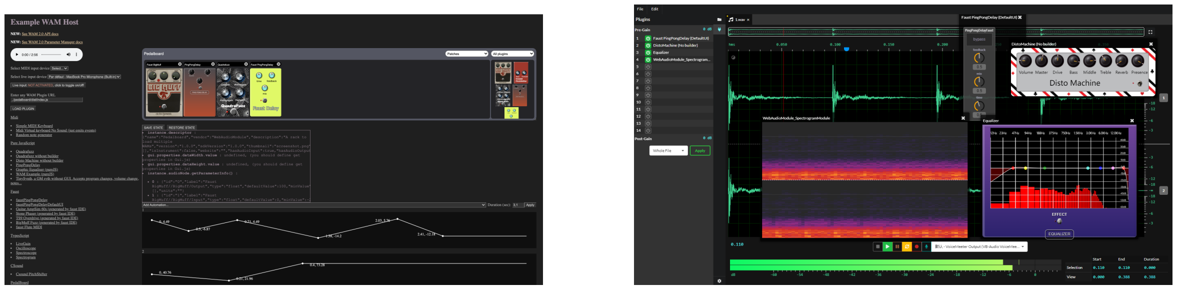

Figure 1: Examples of Web Audio Modules plugins loaded in a host application (left: host from WAM distribution, right: audio editor of the JSPatcher application)

ABSTRACT

A group of academic researchers and developers from the computer music industry have joined forces for over a year to propose a new version of Web Audio Modules, an open source framework facilitating the development of high-performance Web Audio plugins (instruments, realtime audio effects and MIDI processors). While JavaScript and Web standards are becoming increasingly flexible and powerful, C, C++, and domain-specific languages such as FAUST or Csound remain the prevailing languages used by professional developers of native plugins. Fortunately, it is now possible to compile them in WebAssembly, which means they can be integrated with the Web platform. Our work aims to create a continuum between native and browser based audio app development and to appeal to programmers from both worlds. This paper presents our proposal including guidelines and implementations for an open Web Audio plugin standard - essentially the infrastructure to support high level audio plugins for the browser.

CCS CONCEPTS

• **Software and its engineering → Abstraction, modeling and modularity**.

KEYWORDS

Web Audio, audio effects and instruments, plugin architecture, Web standards

ACM Reference Format:
Michel Buffa, Shihong Ren, Owen Campbell, Tom Burns, Steven Yi, Jari Kleimola, and Oliver Larkin. 2022. Web Audio Modules 2.0: An Open Web Audio Plugin Standard. In *Companion Proceedings of the Web Conference 2022 (WWW '22 Companion), April 25–29, 2022, Virtual Event, Lyon, France.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3487553.3524225

1 INTRODUCTION

In 2015 Jari Kleimola and Oliver Larkin created Web Audio Modules (WAM) [7], a standard for creating reusable and interoperable plugins implemented as WebComponents, the core of which was written in C/C++. This initiative was aimed primarily at developers of native plugins (i.e. VST plugins). In 2018, they joined forces with other groups of people working on interoperable Web Audio plugins and plugin hosts (including authors of this paper, some of whom are members of the W3C Web Audio WG) to synchronize their efforts toward the beginnings of an open standard called Web Audio Plugins (WAP) [1, 2], covering a wider range of use cases. Since 2018 W3C Web standards have matured: the appearance of WebAssembly, stabilization of WebComponents, support for AudioWorklets [4] in the Web Audio API, and continued evolution of JavaScript have all helped make professional-grade, Web-based audio production a reality. In addition, commercial companies now offer digital audio workstations (DAW) on the Web which act as host Web applications and support plugins [3].

Taking into account these developments and the feedback received from developers over the past few years, our group has further expanded and started working on a new version at the end of 2020. The project, called "Web Audio Modules 2.0" (WAM2), is open source and distributed as GitHub repositories and as npm modules.

2 BACKGROUND CONTEXT AND TERMS

2.1 State of the Art: Native Plugin Standard

Over the last 20 years there has been a big shift in the technology used for music production and performance. One of the most significant changes is that there are now far fewer hardware devices and many software-based studios where mixing consoles, effect units, and tape machines have all been replaced by virtual equivalents. One of the landmark moments for this technology was the introduction of VST (Virtual Studio Technology[1]) by Steinberg in 1996. This is a standard where modular audio software devices (plugins) are deployed as dynamic libraries loaded by a host application which would typically be a DAW. This has become one of the primary means by which musicians access new sounds, and in addition to a lively hobbyist community[2] there are many companies that focus on the development of these software instruments and effects. Several other manufacturers of DAWs and operating systems developed their own APIs in order to have more influence over the user experience of their platform (Apple's AudioUnits, Avid's AAX etc.), and the open source community has created similar formats[3] as well. These APIs all share common functionality, namely processing blocks of audio samples, handling parameter changes, handling MIDI, and managing state. Many third-party developers opt to use an intermediate C++ framework, such as JUCE[4]or iPlug[5], which facilitates compilation of a single codebase to multiple target APIs, saving the developer a lot of time.

2.2 The Web Audio API and the Lack of an Open Plugin Standard

The Web Audio API in its 1.0 version is now a W3C recommendation (a "frozen standard"). It proposes a set of unit generators called AudioNodes for graph-based realizations of audio algorithms and it is supported in the latest versions of most popular desktop and mobile browsers. The connection of these nodes in the browser via a JavaScript API allows for a range of different applications involving realtime audio processing. The API comes with a limited set of built-in nodes for common operations such as volume control, audio filtering, time delay/echo, reverberation, dynamics processing, spatialization, etc. The addition in 2018 of the AudioWorklet node provides a solution for implementing custom low level audio processing (supported by all major browsers in 2022), including DSP code compiled to WebAssembly, that will be executed in the browser's audio thread. Such nodes can be assembled into an "audio graph", which developers can use to write more complex audio effects or instruments.

Sometimes, when performing live, one needs to chain audio effects together (for example in the guitarist's pedal board) and when composing/producing music multiple effects and instruments are often used. These are use cases where the Web Audio API nodes are too low level, hence the need for a higher level unit in order to represent the equivalent of the "plugins" that are ubiquitous in native DAWs. Multiple proposals for WebAudio plugin standardization arose since 2013 [1, 5–7]. We published a state of the art [1–3] on this domain which is still relevant today.

2.3 What Makes the Web Platform Different?

Although we aim to introduce the functionality offered by native audio plugins to the Web, the distinct advantages and limitations of the environment require a different approach to API design and at least guidelines on how to package plugins for publication on a plugin repository/server and for their integration in a host. A Web-based API should be "Web aware" and use URIs as plugin identifiers. Plugins are just another kind of Web resource, like images, CSS or JS files and should be referenced by a relative/local or remote URL. Host web apps should be able to discover local/remote plugins by querying plugin folders/servers. Remote plugins should be usable without the need to download them manually, and the mixture of different JavaScript libraries and frameworks should not raise any naming conflicts or dependency problems. In the following sections we present WAM 2.0, a new evolution of the standards we have developed up to 2018, and emphasize the new features it offers.

```
1   const pluginURI = 'http://mypluginserver.com/MyFlanger/index.js';
2   const sdkURI = '../../sdk/';
3
4   (async () => {
5       // Init WamEnv
6     const { default: initializeWamHost } = await import(`${sdkURI}/initializeWamHost.js`);
7     const [hostGroupId] = await initializeWamHost(audioContext);
8
9     // Import WAM
10    const { default: WAM } = await import(pluginURI);
11    // Create a new instance of the plugin
12    // You can can optionnally give more options such as the initial state of the plugin
13    const instance = await WAM.createInstance(hostGroupId, audioContext);
14
15    // Connect the audionode to the existing audio graph
16    connectPlugin(instance.audioNode);
17
18    // Get the GUI as a DOM node if needed (headless plugin, pass {noGui:true})
19    const pluginDomNode = await instance.createGui();
20
21    // display the GUI
22    document.body.append(pluginDomNode);
23
24    // start
25    player.onplay = () => {
26      audioContext.resume();
27    };
28  })();
```

Figure 2: Excerpt of code for importing and displaying a simple WAM plugin

Figure 2 shows code for a simple host loading a WAM, connecting it to the audio graph, and displaying its GUI. At line 10 a JavaScript dynamic import is used to load a WAM instance using its URI. The WAM instance "audioNode" property can be connected to the audio graph (line 16), while its "createGUI" method returns an HTML element that contains the plugin GUI (line 19), which can be in turn added to the DOM (line 22). The result of the execution of this

[1]https://en.wikipedia.org/wiki/Virtual_Studio_Technology
[2]http://www.kvraudio.com
[3]http://lv2plug.in/gmpi.html
[4]http://www.juce.com
[5]https://github.com/iPlug2

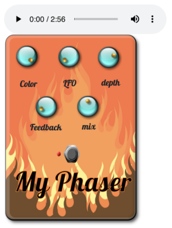

Figure 3: Host web page that corresponds to the execution of the code from Figure 2

code can be seen in Figure 3. In this example[6] the plugin is a phaser effect coded in Faust, compiled to a WebAssembly module residing in an AudioWorklet, and its GUI is a WebComponent.

3 WAM 2.0 ARCHITECTURE

3.1 Project Organization

Since the previous version of the WAM API, an organization at GitHub called WebAudioModules[7] was created to gather the related repositories and projects. The current work is grouped into 4 repositories, all under MIT license. They correspond to the WAM API (detailed in section 4), SDK and Param Manager SDK (section 5), and examples of plugins and hosts (section 6). The API and the SDKs are also published as NPM packages[8] under @webaudiomodules namespace (scope).

3.2 Realtime Audio Processing with AudioWorklet

Realtime audio processing is an example of a "hard realtime" task, meaning all computations must complete within a more or less fixed, finite amount of time determined by a number of factors tied to hardware capabilities and configurations, usually on the order of several milliseconds. If the processing cannot finish in the allotted time, audible glitches will corrupt the output. A critical component of such systems is a high-priority thread where short blocks of samples can be processed before being fed to the system's digital-to-analog converter and played over speakers or headphones. At present, AudioWorklet is the only entrypoint for developers to access the high priority audio thread that runs the Web Audio API's processing graph, and thus it is a central component of our proposed plugin architecture.

AudioWorklets are divided into two parts reflecting the multi-threaded nature of the environment: AudioWorkletNode and AudioWorkletProcessor. These APIs are extended to provide the main thread and audio thread components of WAM plugins: the WamNode and WamProcessor. While some WAMs may be implemented using a single node, others may be implemented using a subgraph of multiple nodes which may consist of an arbitrary mixture of built-in Web Audio nodes and custom AudioWorklets. All of this is abstracted by the WamNode and WamProcessor interfaces, which are presented to the host as a single Web Audio node and processor, regardless of underlying implementation. This abstraction is key to seamless interoperability between hosts and plugins on either thread.

[6]Available online: https://tinyurl.com/mr48npau
[7]https://github.com/webaudiomodules
[8]https://www.npmjs.com/search?q=keywords:webaudiomodules

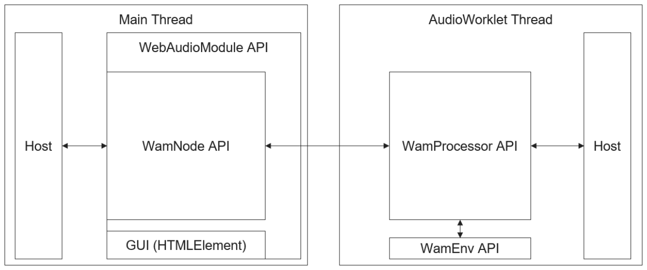

Figure 4: Each WAM is composed of a WamNode-WamProcessor pair.

3.3 Facilitating Interaction on the Audio Thread

Because the Web Audio API provides no "official" way to communicate with processors on the audio thread, the WAM framework relies on a singleton object attached to the audio thread's global scope called the WamEnv to facilitate interactions between hosts and processors. Processors are organized into WamGroups, managed by the WamEnv, where each WamGroup contains plugins created by a particular host or subhost (such as a pedalboard plugin). Much care was taken to minimize the API's assumptions about host implementation, and the WamEnv and WamGroup are the only objects in the WAM ecosystem which are expected to be provided by the host. Figure 4 illustrates the relationships between the host, WamNode, WamProcessor, and WamEnv, and how the API facilitates communication between these entities on each thread.

4 API OVERVIEW

4.1 Interoperability

The new WAM API[9] includes definition files for plugins and hosts written in TypeScript and abstract classes that conform to the API written in JavaScript. The API is designed for making web-based modular audio plugins and using them in compatible hosts. Relevant parts of the WAM API should be implemented by each WAM plugin or host. Plugins and hosts which conform to the WAM API are guaranteed to be compatible, regardless of their implementations.

4.2 Primary API Features

Getting the WAM's information by fetching a JSON file: Each WAM has a descriptor which contains metadata such as name, version, url for loading and running the plugin, etc.

Loading the WAM plugin constructor by fetching an ECMAScript Module file: WAMs must provide a primary index.js module file. By loading this file from the plugin URI, a host can create plugin instances.

Getting a WebAudio AudioNode-compatible processor that can be inserted into an existing audio graph: WAMs appear to hosts to be single Web Audio nodes, even if they are composed

[9]https://github.com/webaudiomodules/api

of multiple Web Audio nodes. The SDK comes with a CompositeAudioNode class that WAMs can inherit from which can wrap a subgraph of multiple nodes such that it behaves like a single node.

Saving and restoring the plugin's state: WAMs can have many different parameters and thus many possible states. If a WAM is loaded into a DAW and its settings are manipulated by the user, it is mandatory to be able to recall and reapply those settings (i.e values of the different parameters) when the user saves / loads a project in the host application.

Agnostic of host / plugin implementation: The standard is designed to provide full support for hosts which run in JavaScript's main thread or on the audio thread (via AudioWorklet). Furthermore, care has been taken to ensure that the interoperability guaranteed by the API makes as few assumptions as possible about the underlying implementation of both hosts and plugins. While the SDK provides concrete implementations, there is no requirement for developers to use it in order to build and distribute a valid WAM if they wish to implement some or all of the API themselves. Since WAM hosts and plugins can conceivably be designed using standard WebAudio nodes (in situations where access to the audio thread is not desired) or using AudioWorklet (in situations where running custom code and interacting with WAM processors synchronously on the audio thread is desired), it is necessary for WAM plugin developers to implement a subset of the API on both the main thread and audio thread. The SDK provides a number of convenience classes to help make it easier for developers to meet this requirement.

Getting parameter information from the main thread: Hosts can access each WAM's specific set of parameters (names, values, etc.) from either thread.

Scheduling automation events of audio parameters from both threads: Hosts can schedule automation of WAM plugin parameters from either thread.

Scheduling transport, MIDI and OSC events with the host from both threads: Hosts can schedule control events using protocols such as MIDI or OSC from either thread.

Passing events between WAM plugins on the audio thread: Hosts can make (non-audio) connections between WAMs for broadcasting / receiving events. Like audio connections between nodes, these event connections are managed from the main thread. However, the events are actually transmitted on the audio thread, a task facilitated by the WamEnv / WamGroup. Such connections can only be made between plugins residing in the same WamGroup.

Cleaning up when a plugin instance is destroyed: The API accounts for the full lifecycle of a WAM, ensuring that a plugin's resources can be released when it is no longer needed.

4.3 Key API Constructs

4.3.1 Interfaces and abstract classes. The API is composed of abstract interfaces and class definitions which fully define the expected behavior of a plugin and the environment it operates in. The primary interfaces are listed in Table 1.

Table 1: Important interfaces in the WAM API

Interface	Description
WebAudioModule	main entry point of a WAM plugin instance
WamDescriptor	contains general information about the plugin
WamNode	extends Web Audio's AudioNode. Can be inserted into the host's audio graph
WamProcessor	extends Web Audio's AudioWorkletProcessor, processes signals in the audio thread
WamParameterInfo	provides parameter metadata and convenience methods
WamParameter	maintains parameter state information
WamEvent	provides information for scheduling or emitting WAM events (automation, MIDI messages)
WamGroup	maintains event graph information for hosts and subhosts on the audio thread
WamEnv	manages WamGroups, registers WamProcessors, and stores plugin dependencies on the audio thread

4.3.2 Plugin Parameters. Since the WAM standard aims to allow maximum flexibility for developers, we have decided that Web Audio's AudioParams should not appear explicitly in the API. Some WAM design patterns do not make use of any stock Web Audio nodes, and while it is possible to define custom Web Audio AudioParams for AudioWorklet, in many cases it would be too heavy and cumbersome to expose the potentially hundreds of parameters residing in WebAssembly code via that API. Furthermore, the parts of the Web Audio API having to do with AudioParams were conceived before developers had any direct access to the audio thread, forcing parameter updates to be scheduled asynchronously from the main thread far enough in advance to account for crossing the thread barrier. This aspect of the Web Audio API is not compatible with our goal to support synchronous, "just in time" interaction between hosts and plugins on the audio thread as in native plugin environments. To get around these limitations we have chosen to create our own WamParameter API to handle parameter updates, designed to facilitate many different WAM design patterns and interaction between hosts and plugins on either thread.

4.3.3 Event Scheduling. Sample-accurate event scheduling is a critical requirement for professional audio applications. The Web Audio API does allow for sample-accurate scheduling of AudioParams, and this capability is leveraged in the WAM SDK to facilitate WAM designs that incorporate one or more built-in Web Audio nodes. However, as mentioned previously the automation API for AudioParams is not useful for many WAM design patterns, and we also wish to support sample-accurate processing of other kinds of events such as MIDI messages. The WAM API facilitates event scheduling through a unified interface that is mirrored on both the main thread and audio thread. Hosts operating entirely on the main thread will still be required to schedule events with some lookahead to ensure that they are processed at the intended time, as these messages must still cross the thread barrier. However, hosts with a presence on the audio thread can schedule events at the beginning of the rendering block in which the events should occur.

4.4 SDK Overview

The SDK provides reference implementations covering the full functionality of the WAM API, as well as some utility classes to help developers build WAMs using several different design patterns. However, if developers wish to use their own implementations there is no need to use the SDK as long as the alternative implementation conforms to the API.

Parameter Manager and CompositeNode

The WAM SDK's Parameter Manager (ParamMgr) and CompositeNode[10] classes provide scaffolding for plugins that use multiple audio nodes. ParamMgr automatically wraps AudioParams as WamParameters, while CompositeNode wraps a subgraph such that it can be treated as a single audio node. In addition, non-audio parameters or multiple AudioParams can also be controlled through these WamParameters by specifying callbacks in the ParamMgr.

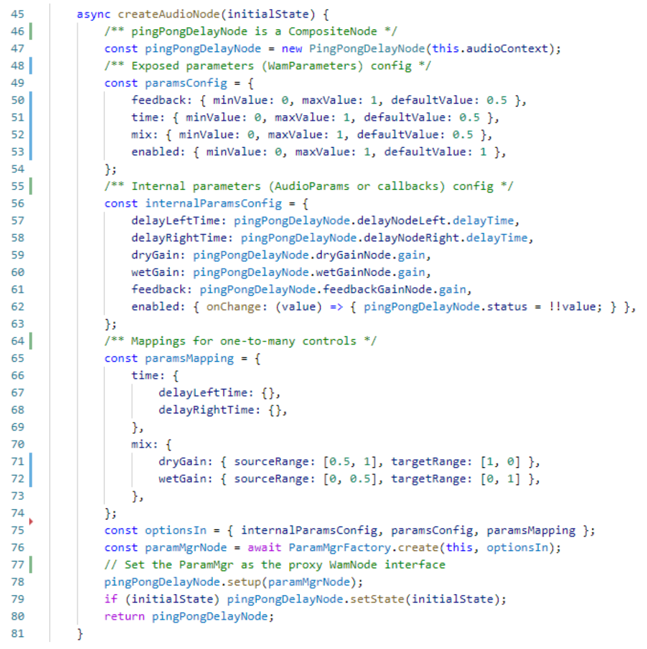
```
async createAudioNode(initialState) {
    /** pingPongDelayNode is a CompositeNode */
    const pingPongDelayNode = new PingPongDelayNode(this.audioContext);
    /** Exposed parameters (WamParameters) config */
    const paramsConfig = {
        feedback: { minValue: 0, maxValue: 1, defaultValue: 0.5 },
        time: { minValue: 0, maxValue: 1, defaultValue: 0.5 },
        mix: { minValue: 0, maxValue: 1, defaultValue: 0.5 },
        enabled: { minValue: 0, maxValue: 1, defaultValue: 1 },
    };
    /** Internal parameters (AudioParams or callbacks) config */
    const internalParamsConfig = {
        delayLeftTime: pingPongDelayNode.delayNodeLeft.delayTime,
        delayRightTime: pingPongDelayNode.delayNodeRight.delayTime,
        dryGain: pingPongDelayNode.dryGainNode.gain,
        wetGain: pingPongDelayNode.wetGainNode.gain,
        feedback: pingPongDelayNode.feedbackGainNode.gain,
        enabled: { onChange: (value) => { pingPongDelayNode.status = !!value; } },
    };
    /** Mappings for one-to-many controls */
    const paramsMapping = {
        time: {
            delayLeftTime: {},
            delayRightTime: {},
        },
        mix: {
            dryGain: { sourceRange: [0.5, 1], targetRange: [1, 0] },
            wetGain: { sourceRange: [0, 0.5], targetRange: [0, 1] },
        },
    };
    const optionsIn = { internalParamsConfig, paramsConfig, paramsMapping };
    const paramMgrNode = await ParamMgrFactory.create(this, optionsIn);
    // Set the ParamMgr as the proxy WamNode interface
    pingPongDelayNode.setup(paramMgrNode);
    if (initialState) pingPongDelayNode.setState(initialState);
    return pingPongDelayNode;
}
```

Figure 5: Code example of CompositeNode and parameter mapping with ParamMgr

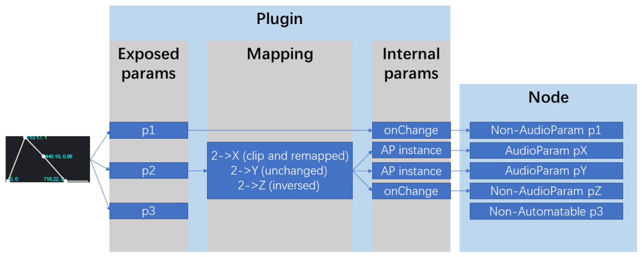

Figure 6: Use parameter mappings in the ParamMgr

The developer should declare and configure via WamParameterInfo every parameter that is controllable and automatable by the host application, and make them accessible via WamNode's methods such as getParameterInfo. In the ParamMgr, we consider these parameters the WAM's "exposed parameters".

By automating or controlling these exposed parameters the host can indirectly change the WAM's internal state. The variables to be changed in the internal state, which we call "internal parameters", can be an AudioParam or an event handler that will be called when the values change. In some use cases the plugin needs to control multiple internal parameters via a single exposed parameter, and with different value scalings or mappings for each internal parameter. (Figure 5 and 6).

5 WAM EXAMPLES

5.1 WAM Plugin Example Repository

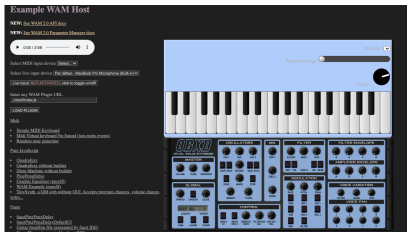

Figure 7: MIDI controller and Oberheim OB-Xd synthesizer WAM plugins loaded in the host.

In addition to the API, SDK, and ParamMgr repositories, we have created a repository[11] presenting a large number of example plugins (effects, instruments, MIDI controllers) using different development approaches (JavaScript, build systems, TypeScript, FAUST, Csound, front-end frameworks, etc.). These examples are testable with a simple host application, also provided in the repo (Figure 1 and 7). An audio player or a live input stream can be used at the root of the host's audio graph to try an audio effect plugin. In the case of plugins that are "MIDI controllers", i.e a piano keyboard, they can be loaded upstream in the processing chain to test audio synthesizers or any other plugins that accept MIDI input. For each loaded plugin, the host allows one to perform some simple tasks like testing automation of its parameters (Figure 1 shows some parameter automation curves).

5.2 Online IDE for FAUST / WebAssembly WAMs

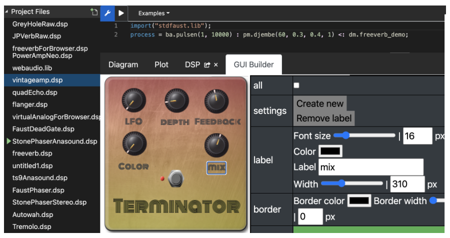
```
import("stdfaust.lib");
process = ba.pulsen(1, 10000) : pm.djembe(60, 0.3, 0.4, 1) <: dm.freeverb_demo;
```

Figure 8: The Faust online IDE can make AudioWorklet / WebAssembly based WAMs in minutes!

The FAUST online IDE[12] (Figure 8) can be used to rapidly create WAMs online [8], and has recently been updated to support WAM

[10] A CompositeNode extends the GainNode class from the Web Audio API. All Web Audio API nodes implement the AudioNode interface.

[11] https://github.com/webaudiomodules/wam-examples, the host can also be tried online: https://mainline.i3s.unice.fr/wam2new/packages/_/

[12] https://faustide.grame.fr/

2.0. An online tutorial[13] (also in video[14]) shows how you can edit, compile to WebAssembly executed within an AudioWorklet, test, analyze, and publish a WAM plugin.

The whole process takes a few minutes if you start from an existing Faust code (many ready to use examples are available from the IDE example menus or from various open source projects). The IDE can also generate a default GUI in SVG and comes with an editor for making WebComponent based GUIs. Plugins can be exported as zip archives or published directly to a WAM server.

5.3 JSPatcher

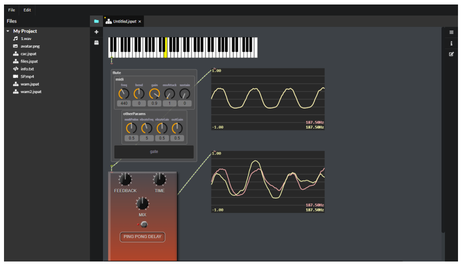

Figure 9: A graph of WAMs loaded in JSPatcher

JSPatcher [9] is an online visual programming language in the style of Max / PureData for interactive programming, audio processing, and realtime multimedia projects. It can be used as a graph editor for WebAudio nodes and WAMs. (Figure 9)

5.4 Sequencer.Party

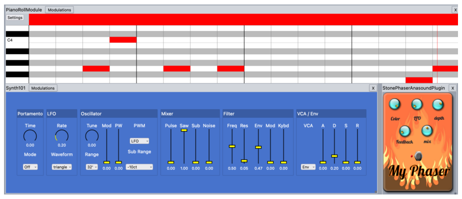

Figure 10: Built-in and remote WAMs loaded in sequencer.party

Sequencer.Party[15] is a realtime collaborative audio / visual platform built entirely out of WAMs. Users work together in real-time sessions, and can share WAM presets and projects publicly on the website. It comes with its own collection of open-source WAMs[16], and users may load remote WAMs by URL (such as the orange WAM plugin visible in Figure 10, created using the FAUST IDE).

5.5 Commercial DAW: Amped Studio

The developer's version of AmpedStudio.com supports WAMs natively, as shown in Figure 11. Communication with WAM plugins can be done entirely in the audio thread as the Amped Studio DAW uses AudioWorklet.

[13]https://tinyurl.com/yf3hrxvw
[14]https://youtu.be/svMnKQDnipo
[15]https://sequencer.party, video: https://youtu.be/8G3we8dikq8?t=10528
[16]https://github.com/boourns/burns-audio-wam

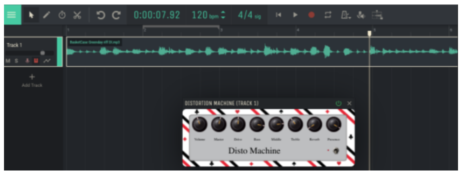

Figure 11: WAM 2.0 plugin running in the commercial DAW Amped Studio

6 CONCLUSION / DISCUSSION

The new WAM ecosystem covers many use cases for developing plugins, from the amateur developer writing simple plugins using only JavaScript / HTML / CSS to the professional developer looking for maximum optimization, using multiple languages and compiling to WebAssembly. It was designed by people from the academic research world and by developers who are experts in Web Audio and have experience developing professional computer music applications. In its current state, the open source WAM 2.0 standard is still considered a "beta version", but in a stable state. The framework provides most of the best features found in native plugin standards, adapted to the Web. We regularly add new plugins to the wam-examples GitHub repository, but there are also dozens of WAMs developed by the community, such as the set of plugins created by the author of sequencer.party, who has open sourced them in their entirety. We invite Web Audio developers and native audio developers interested in plugin development for the Web to have a look at WAM 2.0. This work has been presented in December 2021 to the W3C WebAudio Working Group and Community Group, and in a workshop at the Web Audio Conference 2021[17], as it was made possible by an extensive use of recent results from W3C standardization processes (WebAssembly, WebAudio, etc.)

REFERENCES

[1] Michel Buffa, Jerome Lebrun, Jari Kleimola, Oliver Larkin, and Stephane Letz. 2018. Towards an open Web Audio plugin standard. In *Companion Proceedings of the The Web Conference 2018*. Lyon, France, 759–766.
[2] Michel Buffa, Jerome Lebrun, Jari Kleimola, Oliver Larkin, Guillaume Pellerin, and Stéphane Letz. 2018. WAP: Ideas for a Web Audio Plug-in Standard. In *Proceedings of the Web Audio Conference*. Berlin, France.
[3] Michel Buffa, Jerome Lebrun, Shihong Ren, Stéphane Letz, Yann Orlarey, Romain Michon, and Dominique Fober. 2020. Emerging W3C APIs opened up commercial opportunities for computer music applications. In *The Web Conference 2020 Developers' Track*. Taipei.
[4] Hongchan Choi. 2018. Audioworklet: the Future of Web Audio.. In *Proceedings of the International Computer Music Conference*. Daegu, South Korea, 110–116.
[5] Hongchan Choi and Jonathan Berger. 2013. WAAX: Web Audio API eXtension. In *Proceedings of the International Conference on New Interfaces for Musical Expression*. Daejeon, South Korea.
[6] Nicholas Jillings, Yonghao Wang, Ryan Stables, and Joshua Reiss. 2017. Intelligent audio plugin framework for the Web Audio API. In *Proceedings of the Web Audio Conference*. Queen Mary University of London, London, United Kingdom.
[7] Jari Kleimola and Oliver Larkin. 2015. Web Audio Modules. In *Proceedings of the Sound and Music Computing Conference*.
[8] Shihong Ren, Stéphane Letz, Yann Orlarey, Romain Michon, Dominique Fober, Michel Buffa, Elmehdi Ammari, and Jerome Lebrun. 2019. FAUST online IDE: dynamically compile and publish FAUST code as WebAudio Plugins. In *Proceedings of the Web Audio Conference*. Trondheim, Norway.
[9] Shihong Ren, Laurent Pottier, and Michel Buffa. 2021. Build WebAudio and JavaScript Web Applications using JSPatcher: A Web-based Visual Programming Editor. In *Proceedings of the Web Audio Conference (WAC '21)*. UPF, Barcelona, Spain.

[17]https://www.youtube.com/watch?v=8G3we8dikq8