Web Audio Modules

Jari Kleimola
Dept. of Computer Science
Aalto University
Espoo, Finland
jari.kleimola@alumni.aalto.fi

Oliver Larkin
Music Department
University of York
York, UK
oliver.larkin@york.ac.uk

ABSTRACT

This paper introduces Web Audio Modules (WAMs), which are high-level audio processing/synthesis units that represent the equivalent of Digital Audio Workstation (DAW) plug-ins in the browser. Unlike traditional browser plugins WAMs load from the open web with the rest of the page content without manual installation. We propose the WAM API – which integrates into the existing Web Audio API – and provide its implementation for JavaScript and C++ bindings. Two proof-of-concept WAM virtual instruments were implemented in Emscripten, and evaluated in terms of latency and performance. We found that the performance is sufficient for reasonable polyphony, depending on the complexity of the processing algorithms. Latency is higher than in native DAW environments, but we expect that the forthcoming W3C standard AudioWorkerNode as well as browser developments will reduce it.

1. INTRODUCTION

Digital Audio Workstations (DAWs) have evolved from simple MIDI sequencers into professional quality music production environments. Contemporary DAWs equip home studios with multi-track audio and MIDI recording/editing capabilities, which enable musicians to turn their compositions into publication-ready master tracks. DAWs are standalone native applications, whose functionality may be extended using plug-ins. Plug-ins implement custom virtual instruments and effects processing devices, which co-operate inside the DAW host environment through host-specific application programming interfaces (APIs).

The primary application scope of a DAW is limited to music making in a local single user environment. In this sense, web browsers may be regarded as functional opposites of DAWs: browsers target a wide range of use cases, focus on networked connectivity between many users, and provide remote resource access in a global scope. Like DAWs, browsers have also matured in time from simple document viewing applications into rich interactive multimedia platforms. Moreover, the ever-increasing number of standardized web APIs and open source third party libraries continues to expand their scope of applicability. For instance, the recent Web Audio API extends the browser sandbox to fit in musical applications such as those presented in this work (see Figure 1). Browser functionality may be further increased with novel secure extension formats such as Emscripten and Portable Native Client (PNaCl) that are running close to native speeds, and without manual installation.

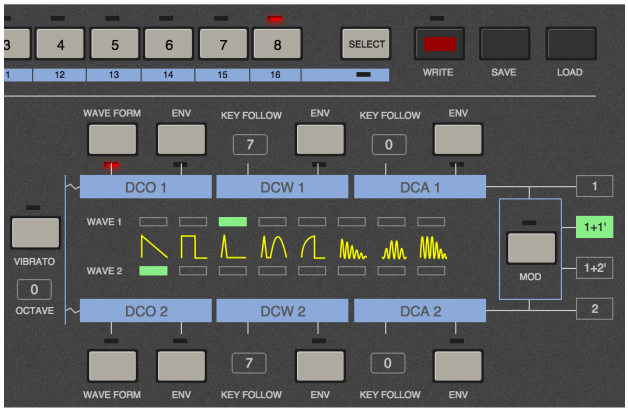

Figure 1. Detail of webCZ-101 user interface.

With these things in mind, we argue that enabling DAW-style virtual instruments and effects processors in web browsers – and integrating them with existing web APIs – introduces novel use cases that go beyond standalone DAW host scenarios. We give examples of four categories.

First, the Internet infrastructure may be utilized in direct distribution of software synthesizers, effects devices and their presets. Online DAW plug-ins may also be used in demoing native plug-ins without installation. More elaborate use cases include collaborative music making and live coding performances. *Second*, seamless integration with online web pages and strong support for multimedia suggests use cases for plug-in tutorials, interactive documentation, music theory lessons, online musical score rendering, audiovisual installations, and parameterized audio assets for online games. *Third*, wireless networking and support for various local communication protocols afford new interaction paradigms for software synthesizer control. Browsers also provide versatile tools for traditional graphical user interface (GUI) implementations. *Fourth*, the direct development approach – based on JavaScript (JS), HTML and CSS – encourages prototyping and exploration of novel audio synthesis and processing algorithms. *Finally*, when conforming to existing

web standards, these four categories are available in cross-platform and cross-device manner without plug-in host vendor control. We envision that many more scenarios will emerge.

This work introduces Web Audio Modules (WAMs), which are DAW-style plug-ins optimized for web browsers. Unlike most existing online synthesizer and audio effects implementations that build *on top* of Web Audio API, WAMs integrate *into* the Web Audio API via its script-based backend node: each WAM thus implements a full-blown software synthesizer or effects device inside a single Web Audio API node. The proposed WAM API strives to make these nodes reusable a) by standardizing how the enclosing web page loads and controls them, and b) by standardizing their DSP interface. The former enables development of novel application scenarios enlisted in the previous paragraph, while the latter enables audio algorithm development in JavaScript or cross-compiled C/C++. This, in turn, affords single code base for native and web audio plug-in implementation. However, in contrast to traditional web plugins, the cross-compiled WAMs load directly from the open web without manual installation (hence the term "module" instead of "plug-in"). The primary contribution of this work consists of:

- a proposal for a streamlined API that enables DAW-style virtual instruments and effects devices in web browsers.
- an implementation of the API in JS and C/C++.
- a minimal WAM example and two proof-of-concept WAM virtual instruments conforming to the API.
- a web service to aggregate WAMs and their presets.

This work also explores how WAMs integrate with existing and emerging APIs such as Web MIDI, WebGL, and Web Components.

The remainder of this paper is structured as follows. Section 2 reviews related work in native and web platforms. Section 3 details the API and its architecture, while section 4 explains how web pages and applications may embed WAMs, and describes their proof-of-concept implementations. Section 5 evaluates the implementations in terms of latency and performance, and finally, Section 6 concludes.

Source code, documentation, demos, and the web service are available via links at the accompanying website[1]

https://mediatech.aalto.fi/publications/webservices/wams

2. RELATED WORK

2.1 Native Plugin APIs

In 1996 Steinberg introduced Virtual Studio Technology (VST) and started a trend towards "in the box" audio production, where hardware effects and instruments could be replaced by native software equivalents running inside a DAW on personal computers. A publicly available C++ SDK[2] allowed developers to create their products as plug-ins – dynamic libraries conforming to a specific API, to be loaded by a "Host" application, which would typically be a DAW. A small industry had developed around the technology by the early-2000s with companies adopting the format along with a hobbyist community. Some host vendors, such as Apple and AVID created competing APIs allowing them a tighter control of the market and better integration with their platform. Although it represents a small fraction of musical instrument retail sales, the industry is still growing (at least in the USA) as can be seen in the NAMM 2014 global report[3].

Several plug-in APIs have prevailed and are used widely at the time of writing, including Steinberg's own VST2.4 and VST3, Apple's AudioUnit and AVID's AAX. In the open source community LADSPA and LV2 (LADSPA version 2) have been widely adopted. In the commercial arena the success and adoption of a particular API is often dictated by the host vendors and the market share they control, more than the merits of the API itself. VST plug-ins are supported by many hosts on the Mac and PC platforms, although the VST3 format, which is substantially different from VST2.4, has not yet seen widespread support outside of Steinberg's DAWs. The AudioUnit format only runs on the Mac platform and is the only format supported by Apple's popular Logic DAW. Access to the AAX SDK is controlled by AVID and only AVID can produce AAX hosting applications (such as ProTools). LV2 is platform agnostic and entirely open source with the most liberal license, but to date uptake has been mainly on Linux.

In 2003 a working group of the MIDI Manufacturer's Association (MMA) was set up to develop a non vendor controlled plug-in API Generalized Music Plug-in Interface (GMPI). Although this API never materialized, a draft of a list of requirements was produced based on the members' discussions[4]. This has informed the LV2 plug-in specification[5] and serves as a useful reference for the design of audio plug-in APIs.

Audio plug-in developers who want to make their software compatible with a variety of hosts and platforms and reach a wide market have to support multiple APIs and the complexity is increased by the need to provide cross platform GUI and file system features. For this reason many developers use an intermediate C++ framework such as JUCE[6] or IPlug[7] (or a proprietary solution) in order to develop an abstracted version of the plug-in which can then be compiled to multiple formats, platforms and architectures, saving development time.

Although we aim to introduce the functionality offered by the concept of native audio plug-ins to the web, the differences of the environment require a different approach to the API design, and the development of a new

[1] https://mediatech.aalto.fi/publications/webservices/wams

[2] http://www.steinberg.net/en/company/developers.html

[3] https://www.namm.org/files/ihdp-viewer/global-report-2014/

[4] http://retropaganda.info/archives/gmpi/gmpi-requirements-2005-04-05-final-draft.html

[5] http://lv2plug.in/gmpi.html

[6] http://www.juce.com/

[7] https://github.com/olilarkin/wdl-ol

API for WAMs provides an opportunity to improve upon some aspects of native APIs. Criticisms of existing audio plug-in APIs would include single vendor-control, unnecessary complexity/verbosity, ambiguity of operation (which thread calls which method and when), synchronization of user interface and DSP processing state, and multifarious preset formats which lead many plug-in developers to create their own preset format, thus increasing the problem.

2.2 Web Audio

The Web Audio API [1] is a W3C standard for enabling realtime audio synthesis and processing in web browsers. The API models audio algorithms as interconnected node graphs. The current node set includes 18 native nodes as general building blocks (e.g., classic waveform oscillators and filters), and a generic script node that enables arbitrary DSP algorithm implementations using JS. The Web Audio API is still in development, and the current ScriptProcessorNode (SPN) – which resides entirely in the main thread – will eventually be deprecated in favor of AudioWorkerNode (AWN). AWN splits its functionality between main and audio threads for reduced latency and increased performance. Web MIDI API [2] complements Web Audio API by offering access to local MIDI devices for control-oriented tasks.

The Web Audio API extension framework (WAAX) [3] abstracts Web Audio API node graphs as units, which may be parameterized and interconnected with other WAAX units and Web Audio API nodes. Its latest version turns units into more functional plug-ins, and provides two-way data binding between plug-in parameters and GUI elements. Plug-in parameters abstract Web Audio API AudioParams, and are therefore sample accurate and may be modulated at audio rates. WAAX targets only native Web Audio API nodes, and does not support scripted DSP algorithms. The WAM concept introduced in this work thus complements WAAX.

Web Audio Components[8] (WACs) are interoperable and reusable custom DSP units similar to WAAX plug-ins. WACs define a JavaScript Object Notation (JSON) manifest and publish themselves in a centralized registry. Each WAC also implements a constructor, metadata for parameter space description, and set of instance properties for interconnecting with other nodes. The current WAC registry has a RESTful[9] API, and contains eight components that operate as building blocks of larger DSP pipelines.

WebMidiLink[10] defines a simple textual language for transmitting MIDI and patch dump messages between a hosting web application and a conforming web synthesizer. The service maintains a list of synthesizer descriptors in JSONP format. A conforming synthesizer is loaded from the URL into an `iframe`, which enables cross-domain control using `window.postMessage()` function calls. At the time of writing, WebMidiLink registry contains 16 conforming web synthesizers. WAMs do not require an `iframe` container, but a simple wrapper can make them WebMidiLink conformant.

Emscripten [4] is a toolchain and virtual machine that enables cross-compilation of C/C++ code into high performance JS subset called asm.js [5]. Since asm.js *is* JavaScript, Emscripten modules (such as WAMs) work in all modern browsers without manual installation. PNaCl [6] loads LLVM bitcode (which is cross-compiled from C/C++) into the browser, and compiles that into native sandboxed code ahead of runtime. Recently in [7], native DAW plug-ins were ported to web environments as Emscripten and PNaCl modules. The work concluded that porting is feasible, and that web browsers are capable of running ported plug-ins without audible artifacts. The latencies were found to be higher than in native implementations, but expected to improve with AWNs.

As stated above, conforming to native DAW plug-in format such as VST carries unnecessary complexity that is irrelevant in web platform. Instead of porting native plug-in formats, the present work proposes a streamlined API that is optimized for web browsers and the AWN node. However, since AWNs are not yet supported by browser engines, the current WAM version uses SPN to emulate the AWN approach.

We also address a few Web Audio API shortcomings. Although the current node set exposes common and often used building blocks for various DSP implementations, the number of native node types (18) is insufficient to cover general DAW-style virtual instrument realizations. Parameterization and granularity of nodes raises further issues: for instance, Web Audio API oscillator nodes do not expose phase signals for external manipulation, and the filter nodes are simply textbook biquads. Single sample feedback connections between nodes are also problematic. The WAM API proposal relies therefore on script nodes that do not pose similar restrictions. The following section describes the proposed WAM architecture and API in detail.

3. PROPOSED API

3.1 Goals and Restrictions

The goal of the WAM proposal is to specify a streamlined API that enables DAW-style virtual instruments and effects processors in web browsers. The API needs to be simple, extensible, and strive for minimal latency and maximum performance. WAMs should load straight from the open web without manual installation, and they should integrate seamlessly with existing W3C APIs. WAMs may be developed in vanilla JavaScript, or cross-compiled C/C++ using the Emscripten or PNaCl toolchains.

The browser sandbox and W3C APIs pose specific restrictions to WAM implementations. The most prominent are: A) access to native operating system services and resources such as the local file system is restricted, and

[8] http://component.fm/#about

[9] https://en.wikipedia.org/wiki/Representational_state_transfer

[10] http://www.g200kg.com/en/docs/webmidilink/

B) custom DSP needs to run in a separate audio thread, while the rest of the WAM (e.g., GUI) resides in the main thread. Inter-thread communication is asynchronous.

3.2 Architecture

A WAM consists of Controller and Processor parts as shown in Figure 2. The **Controller** exposes the JS developer API, interfaces with other web APIs, and optionally provides the GUI. The **Processor** implements signal processing algorithms in JS or cross-compiled C/C++. Controller and Processor run in separate threads, and communicate through a "datachannel" using asynchronous events. In most cases, the events flow in a single direction (from Controller to the Processor), and asynchronous request-reply communication is only required during the initialization phase. The events are parsed and translated into method invocations at the Processor side in the **wrapper API**, which is exposed as a JS prototype or C/C++ header file. There is no traditional plug-in host concept in the API. Instead, the Controller hosts the Processor directly, and all interaction with the WAM and the web application code happens through Controller. This resolves the ambiguity of operation and synchronization issues present in some native plug-in APIs.

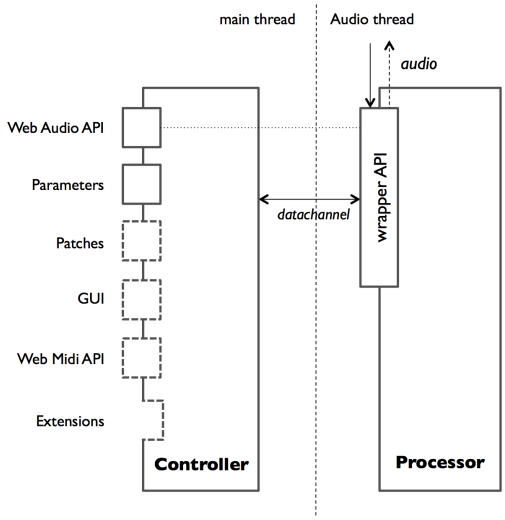

Figure 2. WAM architecture. Solid squares denote mandatory functionality that all WAMs need to implement, dashed ones are optional.

The division of functionality between the two WAM parts is as follows. The Controller holds the state (e.g., parameter values, loading and saving them from/into patches), while Processor implements the DSP (reflecting the parameter values as properly scaled synthesis parameters). Audio buffers are passed directly from/to the audio rendering pipeline, and they are thus not transferred between Processor and Controller. The parameter space and audio/event I/O configuration is declared as a JSON descriptor during initialization time, either at the Controller side or the Processor side (latter preferred). GUIs are outside the scope of this proposal, although they attach to the Controller using the functions defined in the API. Handling of the remaining optional functionality, i.e., Web MIDI API integration and patch handling is at the discretion of each individual WAM implementation.

3.3 Controller API

The Controller is implemented in vanilla JS and it runs in the web application's main thread. The mandatory functionality consists of lifecycle management (discussed later), Web Audio API integration (AWN-based implementation will move this to the Processor side), and parameter handling. The full Controller prototype, including optional functionality, is shown in Listing 1. Custom Controllers are derived from `WAM.Controller` using prototypal inheritance, and they decide which optional functionality to support.

```
WAM.Controller = function () { … }
WAM.Controller.prototype = {
  setup: function (actx,bufsize,desc,proc) { … },
  terminate: function () { … },
  connect: function (destnode, port) { … },
  disconnect: function (destnode, port) { … },
  getParam: function (id) { … },
  setParam: function (id, value) { … },
  setPatch: function (data) { … },
  postMidi: function (msg) { … },
  postMessage: function (verb,resource,data) { … },
  onMessage: function (verb,resource,data) { … } };
```

Listing 1. Controller prototype.

A WAM is exposed as a virtual Web Audio API AudioNode instance, which may be inserted into the node graph like any other real AudioNode. The JSON descriptor, which is thus formed either in Controller or Processor side during initialization time, serves as a contract between control and patch handling and the DSP implementation. The descriptor contains audio, MIDI, and data I/O configuration, as well as parameter space definition. WAM may contain any number of *audio* input and output buses, each with variable number of channels (thus enabling side-chaining). *MIDI ports* are bidirectional and *data ports* are provided for non-MIDI control streams, such as Open Sound Control (OSC). Finally, *parameter* definitions are optionally organized into a tree-like structure, which permits URL-like parameter addresses familiar from OSC. Each parameter is defined with id, name, datatype, min/max/default/step values, and modulation rate (control or audio). Parameter types include int32, double, enum, string, bool, and opaque chunk (void* + length).

3.4 Processor API

The Processor implements the realtime DSP algorithms conforming to the wrapper API, which operates as a bridge between the Controller and the Processor (see Figure 2). The wrapper API has bindings for JS and C/C++. Custom Processor implementations inherit `WAM::Processor` class, whose C++ interface is shown in Listing 2.

```
class Processor {
// -- lifecycle
public:
Processor() {}
virtual const char* init(uint32_t bufsize, uint32_t sr,
   char* descriptor);
virtual void terminate() {}
// -- audio and data streams + patches
virtual void onProcess(AudioBus* audio, void* data) = 0;
virtual void onParam(uint32_t idparam, double value) {}
virtual void onMidi(byte* msg, uint32_t size) {}
virtual void onMessage(char* verb, char* res, void* data,
   uint32_t size) {}
virtual void onPatch(void* data, uint32_t size) {}
// -- controller interface
protected:
void postMessage(const char* verb, const char* resource,
   void* data, uint32_t size) {}
uint32_t m_bufsize,m_sr; };
```

Listing 2. Processor interface.

3.5 WAM Lifecycle

Figure 3 shows WAM lifecycle as a sequence diagram. `WAM.Controller.setup()` first loads the processor script (which contains the implementation of the DSP code in vanilla JS or in Emscripten/asm.js), and calls the `createProcessor()` entry point at Processor side to create a new custom Processor instance. The Controller then initializes the Processor by passing buffer size, sample rate, and an optional descriptor as parameters. The Processor may choose to return a new descriptor as a JSON string instead of conforming to the Controller suggested parameter space (if any). WAM then enters runtime stage, which is aborted by invoking `terminate()`. Terminate disconnects the virtual AudioNode from the Web Audio API node graph, disconnects MIDI ports, and blocks the datachannel between Controller and Processor. Controller and Processor are eventually disposed by garbage collection.

Runtime control is available via `set/getParam`, `setPatch` `postMidi`, and `postMessage` functions, which are routed to the Processor side `onParam`, `onPatch`, `onMidi` and `onMessage` handlers. `Processor.postMessage()` is routed to the opposite direction.

During runtime, Web Audio API requests periodically a new block of samples. The request is dispatched to the `Processor.onProcess()` function, passing audio input and output buffers, as well as AudioParams (containing parameter automation and audiorate parameter modulation signals) in the first argument. The sample size is 32 bit float, normalized to unity range [-1,1]. Audio is non-interleaved, i.e., there is one buffer per channel. Custom Processor implementations may perform internal processing using double precision, although the Web Audio API input and output buffers are restricted to 32 bit floats.

Sample-accurate MIDI and data events are passed in the second argument, which holds a pointer to an ordered event queue. The queue entries are timestamped with sample offsets from the start of the current audio buffer. Since processing sample-accurate events produces overhead, the Processor needs to request them in the JSON descriptor. A detailed description of the optional functionality for data, MIDI, patches, and GUIs is available at the accompanying website[1] of this paper.

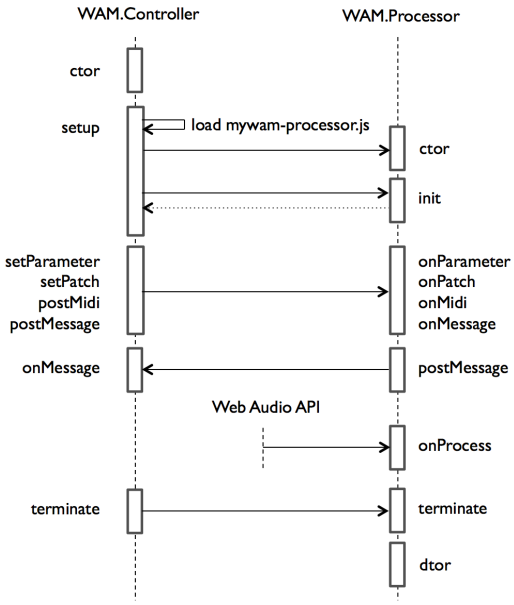

Figure 3. WAM lifecycle.

4. IMPLEMENTATIONS

4.1 WAM Usage

A Web page may embed a WAM by loading the supporting framework and the custom Controller implementation (lines 1-2 in Listing 3), and initializing the WAM in lines 4-9. The initialization script creates new Web Audio API AudioContext and the custom WAM in lines 4-5. Line 6 initializes the WAM instance by passing audio context and buffer size as arguments. The initialization function loads the Processor script asynchronously, and therefore returns a JS Promise that resolves in line eight. Line eight simply connects the custom WAM into the default AudioContext sink (i.e., the speakers). Another implementation might extend line eight into a more elaborate audio graph, for instance, by connecting the WAM into a convolution reverb node. Naturally, the audio graph may also chain WAMs together.

```
1 <script src="wam.min.js"></script>
2 <script src="sinsynth.js"></script>
3 <script>
4    var actx = new AudioContext();
5    var sinsyn = new SinSynth();
6    sinsyn.init(actx, 256).then(function ()
7    {
8       sinsyn.connect(actx.destination);
9    });
A </script>
```

Listing 3. WAM usage.

As stated previously, GUIs are outside the scope of WAM proposal. However, like WAAX [3], we have found Web Components and Polymer[12] useful for GUI implementation. Listing 4 shows an example. Line 1 loads the Polymer framework, while line two uses HTML imports to include the custom GUI. Line four

inserts the GUI into the web page, which may of course contain other HTML5 and WAM GUIs as appropriate. The `controller` attribute links the GUI with the custom WAM embedded in Listing 3.

```
1 <script src="polymer.min.js"></script>
2 <link rel="import" href="wam-sinsynth.html">
3 <body>
4   <wam-sinsynth controller="sinsyn"/>
5 </body>
```

Listing 4. Embedding WAM GUI into a webpage.

Listing 5 shows a minimal Controller implementation. The key to brevity is line five, which delegates most of the functionality to the WAM framework. The `setup` call in line three initializes the Processor. The third parameter denoting the descriptor is set to null, indicating that the Processor side is free to define its parameter space.

```
1 var SinSynth = function () {
2 self.init = function (ctx, bufsiz) {
3   return self.setup(ctx,bufsiz,null,"sinproc.js");
4   }};
5 SinSynth.prototype = new WAM.Controller();
```

Listing 5. Minimal WAM Controller (sinsynth.js).

Listing 6 shows the related minimal Processor implementation in vanilla JS (parts omitted for brevity). Line one is the entry point creating a new Processor instance. Line four returns a descriptor to define the number of audio input/output ports and parameter space. Lines 5-B implement the DSP algorithm for a simple monophonic sinusoidal synthesizer. Line six indicates silence, while line B indicates data in the output buffer. Line C receives MIDI input to update `voiceActive` and phase increment `phinc` member variables according to received status code and note number. Line D receives a parameter from the GUI to update the `gain` parameter, and finally, line E ties the implementation into the WAM framework.

```
1 function createProcessor() { return new SinProc; }
2 var SinProc = function () {
3 this.init = function (bufsize,sr,desc) {
4   return { … }}
5 this.onProcess = function (audio,data) {
6   if (!voiceActive) return false;
7   var out = audio.outputs.getChannelData(0);
8   for (var n=0; n<out.length; n++) {
9     out[n] = gain*Math.sin(phase*2*Math.PI);
A     phase = (phase + phinc) % 1; }
B   return true; }
C this.onMidi  = function (msg) { … }
D this.onParam = function (id,value) { … }};
E SinProc.prototype = new WAM.Processor();
```

Listing 6. Simple Processor implementation (sinproc.js)

WAM JS bindings enable rapid audio algorithm prototyping, since code changes are reflected by simply refreshing the browser window. WAM bindings also provide additional prototyping boost with a generic polyphonic synthesizer framework[1].

4.2 webCZ-101

webCZ-101 is an emulation of the Casio CZ101 Phase Distortion synthesizer, based on the DSP engine of Larkin's VirtualCZ plug-in[11] with a new user interface developed using Web Components/Polymer[12] (see Figure 1). VirtualCZ is implemented using the IPlug C++ framework, which the authors were able to extend to export the processor part of the WAM. This demonstrates how a closed-source plug-in, written in C++ can be ported to the WAM API, and that an existing cross platform plug-in framework can be adapted for WAMs. For some use cases (such as a web demo of a native plug-in, or interactive documentation) it would clearly be desirable to use the same C++ GUI code in the Web version, rather than rewriting it with a web-oriented GUI, but this is out of the scope of this work. In the current situation, where a different web GUI is necessary, porting a native plug-in is made much easier if the code for the DSP of the synthesiser or effect is clearly separated from the existing GUI code, which is something that is encouraged by modern plug-in APIs such as Steinberg's VST3. If GUI or native specific code is interleaved with the DSP, it can usually be easily excluded from compilation via the C preprocessor.

webCZ-101 implements five public methods from the WAM processor C++ interface.

- The init() method specifies the parameter space and I/O of the WAM as a JSON description as well as initializing the DSP with the sample rate and block size.
- The onProcess() method simply calls the DSP's block process method.
- The onMidi() method adds incoming MIDI messages to the DSP's internal MIDI message queue.
- The onPatch() method handles an opaque data chunk that is delivered from the controller after a patch is loaded in the GUI. The chunk is parsed and DSP parameters are updated.
- The onParam() method is called whenever a parameter change occurs in the GUI, and the DSP is updated accordingly.

The webCZ-101 controller side is written entirely in JS and handles the loading of CZ System Exclusive (sysex) files and GUI interaction. Since the source code for the processor part of the WAM is compiled to JS via Emscripten, the code is obfuscated to a degree and cannot be easily reverse engineered, however the JS and supporting files could potentially be used elsewhere, much like any other elements of a web page can be extracted.

4.3 webDX7

The Yamaha DX7 was the first affordable digital synthesizer, and it still remains the most sold hardware synthesizer to date. Since the theory of FM synthesis is well known, several virtual DX7 implementations exist both in open and closed source form. webDX7 uses the open source msfa[13] synthesis engine, which was initially de-

[11] http://www.olilarkin.co.uk/index.php?p=virtualcz

[12] https://www.polymer-project.org

[13] https://code.google.com/p/music-synthesizer-for-android/

veloped for Android OS and later encapsulated as a native VST plug-in[14] and its PNaCl port [7]. In the present work, the msfa DSP engine was wrapped inside the `WAM.Processor` class, and cross-compiled into an Emscripten module. The C++ implementation was then interfaced with a basic JS Controller class.

Although DX7 sounds were notoriously difficult to program, a large amount of patches are available on the Internet due to its commercial success. For instance, the collection at Kronos site[15] contains more than 200,000 patches gathered from the web. The collection contains many duplicates, but still offers a large corpus of presets that are usable with the WAM metadata preset format specification. We started exploring the metadata concept using *set4* from the Kronos collection. After removing duplicates, set4 contained 10236 unique patches pre-organized into instrument categories and their subcategories. This provided a base for keyword-based classification. Each patch's data was then analyzed into a set of continuous-range perceptual features such as attack speed, duration, and DX7 algorithm. The analysis phase takes less than 100 ms for the 10236 patch corpus, which itself takes 437 kB when compressed.

We then explored their visualization to find out how WAMs integrate with other web APIs such as webGL. Each white particle in Figure 4 represents a DX7 patch. The initial screen (Figure 4a) shows a disorganized set of patches, in which particles stray across the screen space with random velocities and bounce from the screen bounds. The user is able to position a "magnet" over the patch cloud and move it around. Depending on the attributes of the magnet, it either attracts or repels patch particles based on their qualities. It is also possible to use multiple magnets with different attributes simultaneously. Particles are either orbiting a single magnet, or moving along a path between several of them. The PatchCloud implementation is based on a force-driven physical model [8] and implemented in three.js[16].

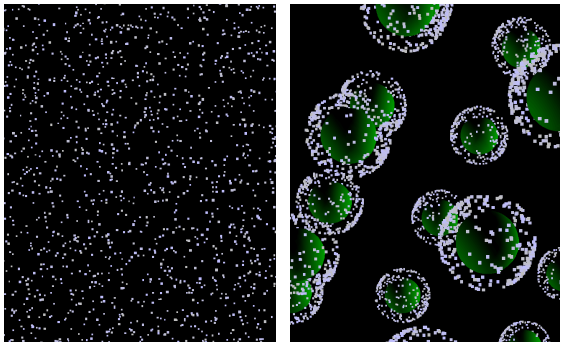

Figure 4. PatchCloud. (a) disorganized set, (b) arranged by DX7 algorithm.

Once a desired magnet constellation has been set up, the user may switch into patch audition mode. A cursor picks a single particle to send the associated patch to the webDX7 instance for audio rendering. Figure 4b shows a snapshot where patches have been organized into an evolving 3D setup based on their algorithms.

4.4 Web Service

We are aggregating WAM implementations and their patches in a public web service. The service maintains a central WAM registry, and exposes a RESTful API for querying and accessing the WAMs. It will thus operate as a distributed cloud VST folder for web applications, such as web DAW hosts. The web service has endpoints for headless WAMs (e.g., sinsynth.js in Listing 3), optional GUI implementations (wam-sinsynth.html in Listing 4), and standalone versions that are embedded inside an `iframe` using a WebMidiLink[10] manifest. For example, a web application may issue a request `"GET /synths/subtractive"` to get a list of all virtual analog WAM synthesizers in the registry, or access one directly by issuing `"GET /synths/subtractive/mysynth.js"`. A similar patch URL enables preset download and upload. Link to the service is available at the accompanying website[1].

5. EVALUATION AND DISCUSSION

5.1 Latency

WAM implementations were evaluated in terms of latency and performance (OSX Mavericks, MPB 2.2 GHz Intel Core i7, 256 sample buffer size, 44.1 kHz sample rate, Chrome v43, Emscripten optimization level –O2). The end-to-end latency was measured by connecting an external MIDI keyboard to a laptop via USB, and using the embedded microphone to capture the mechanical MIDI key click and the output sound of the WAM.

The latency measured 40-48 ms in all implementations as shown in column 2 of Table 1. On the average, this is ~32 ms higher than the theoretical 2×256/44100 = 11.6 ms SPN latency. To find out the cause for the increase, we implemented the baseline algorithm of Listing 6 directly in Web Audio API's `SPN.onaudioprocess()` handler, which gave 39 ms latency on the average. Comparing this to baseline WAM SinSynth, we note that WAM framework overhead is only 1 ms. Latency must therefore be related to the browser, operating system, and mechanical delay in the external MIDI keyboard.

In Chrome, browser-induced latency may be reduced by defining its buffer size with a command-line parameter. We found that buffer size of 32 samples gave lowest latency, as listed in column 3 of Table 1. Considering that the AWN node will remove the 11.6 ms SPN overhead, the latencies have potential to drop below 20 ms.

WAM	default	buffer = 32
SinSynth SPN	39	24
SinSynth WAM	40	28
webCZ-101	48	33
webDX7	45	31

Table 1. Latency in milliseconds.

[14] https://github.com/asb2m10/dexed
[15] http://korgpatches.com/patches/kronos/dx7_200k_collection
[16] http://threejs.org

5.2 Performance

Performance was evaluated in terms of polyphony, i.e. maximum number of simultaneous voices that still produce artifact free sound output. The results are shown in Table 2. The second column lists the number of voices in the WAM implementation, while the third column shows the performance in alternative implementations. webCZ-101 was compared against native standalone version (factory preset BRASS ENS. 1), and webDX7 against the Dexed PNaCl port from [7] (factory preset EPiano1). Baseline was provided by the minimal WAM synthesizer of Section 4.1 and its PNaCl version [7].

WAM	JavaScript	Native / PNaCl
webCZ-101	60	200
webDX7	17	128
SinSynth	280	350

Table 2. Performance in number of voices.

As expected, JS performance was lower than in native and PNaCl targets. webCZ-101 reached 30% of the native standalone VirtualCZ polyphony, which is acceptable. However, webDX7 achieved only 13.2% of the PNaCl polyphony, which suggests that its rather complex processing algorithm does not optimize well for JIT compilation. The performance can however be improved with larger buffer sizes.

5.3 Commercial Concerns

Although the web is based on open standards and web developers are accustomed to the fact that client-side code is easily viewable, there may be concerns relating to copy protection and monetization that could prevent companies from releasing their products as WAMs. The audio software industry is notoriously concerned with piracy, with many companies using hardware based copy protection systems for their products. It would be a significant challenge to reverse engineer the Emscripten-compiled asm.js into a readable and useable native form, but relatively easy to extract a WAM's entire code and use it elsewhere. Until pro audio on the web has matured and is seen to rival native platforms this is probably not a significant issue, and by that time attitudes may have changed and solutions may exist to protect and monetize products of this nature.

6. CONCLUSION

This paper introduced Web Audio Modules (WAMs), which are DAW-style virtual instruments and effects processors for web browsers. A streamlined API which is optimized for the forthcoming Web Audio API AudioWorkerNode was proposed, and two proof-of-concept WAMs were implemented. We found that it is trivial to add a degree of support for the WAM format to existing plug-in abstraction frameworks, and that JS/HTML/CSS provides a rapid prototyping environment for virtual instrument development. The implementations were evaluated in terms of latency and performance. The results show that although the default latency is relatively high, it has potential to fall below 20 ms with proper buffer size adjustments and the introduction of AWNs. Performance of SPN-backed JS modules is sufficient for multitimbral compositions, albeit not yet on par with corresponding native and PNaCl implementations.

We also explored WAM integration with other web APIs. Web Components were found useful in GUI implementation, while WebGL has clear potential in visualizing and browsing large preset libraries. The RESTful web service API for WAMs and their preset dissemination scales well for metadata-based patch queries and even accessing each preset with a unique URL.

Our future work will add support for PNaCl targets and AWN implementation once available. We shall also provide more WAM implementations and investigate how to allow a single code base to be used for both the web and native versions of an instrument or effect.

Finally, we would like to stress out that the API presented in this work is a proposal for community feedback. We welcome comments and contributions to make the API as usable as possible.

7. REFERENCES

[1] P. Adenot, C. Wilson, and C. Rogers, "Web Audio API," W3C Working Draft, Oct 10, 2013 and W3C Editor's Draft, April 03, 2015. Available online at http://www.w3.org/TR/webaudio/ and http://webaudio.github.io/web-audio-api/

[2] J. Kalliokoski and C. Wilson, "Web Midi API," W3C Working Draft, March 17, 2015 and W3C Editor's Draft, April 24, 2015. Available online at http://www.w3.org/TR/webmidi/ and http://webaudio.github.com/web-midi-api/

[3] H. Choi and J. Berger, "WAAX: Web Audio API eXtension," in Proc. Int. Conf. New Interfaces for Musical Expression (NIME'13), Daejeon, Korea, 2013, pp. 499–502.

[4] A. Zakai, "Emscripten: an LLVM-to-JavaScript compiler," In Proc. ACM Int. Conf. companion on Object oriented programming systems languages and applications (OOPSLA '11). New York, 2011, pp. 301-312.

[5] D. Herman, L. Wagner, and A. Zakai, "asm.js," Working Draft, Aug. 18, 2014. Available online at http://asmjs.org/spec/latest/.

[6] A. Donovan, R. Muth, B. Chen, and D. Sehr, "PNaCl: Portable Native Client Executables," White paper, Feb. 22, 2010.

[7] J. Kleimola, "DAW Plugins for Web Browsers," in Proc. 1st Web Audio Conference (WAC-15), Paris, 2015.

[8] D. Shiffman, The Nature of Code, 2012. Available online at http://natureofcode.com