Proceedings of the 2nd AES Workshop on Intelligent Music Production, London, UK, 13 September 2016

JSAP: INTELLIGENT AUDIO PLUGIN FORMAT FOR THE WEB AUDIO API

Nicholas Jillings and Ryan Stables

Digital Media Technology Lab
Birmingham City University
nicholas.jillings@mail.bcu.ac.uk, ryan.stables@bcu.ac.uk

ABSTRACT

Plugins are commonplace in audio production environments, however a common standard has not been developed for the web, utilising the Web Audio API. In this study we define a standard that can be deployed in web-based production environments by defining the plugin structure and host integration. The standard facilitates a novel method of cross-adaptive processing where features are transmitted between plugin instances instead of audio routing, saving on multiple calculations of features. The project will also enable the collection and delivery of semantic information to further the field of Intelligent Music Production.

1. INTRODUCTION

The Web Audio API [1] defines a cross-browser interface for real-time audio processing. The API is supported on all major desktop browsers and has led to a wide range of applications, including additive synthesisers [2] and full production suites[1]. Web Audio API plugin standards have been proposed before, such as Web Audio API eXtension (WAAX) [3] or Tuna[2]. These build audio effects nodes similarly to the Web Audio API's defined nodes, which are too restrictive for a full plugin standard. The Web Audio Modules (WAM) [4] define a processor / editor interface, however these assume the plugin processor is entirely custom DSP code and does not support the use of the streamlined audio nodes.

JSAP[3] (JavaScript Audio Plugin) is a new standard to build audio plugins for the web. It defines both the host interface (`PluginFactory`) and the plugin structure (`BasePlugin`) with all audio processing performed using the web audio API. The standard also defines a novel feature sharing method for building auto-/cross- adaptive effects as well as linking plugins with the session through semantic terms.

2. ARCHITECTURE

Audio processing is performed using the web audio API. Therefore each plugin instance holds an audio graph, called the 'sub-graph'. This is private to the plugin and cannot be controlled directly unless exposed by design. Each plugin instance defines a number of input and output connection points (which are web audio nodes). Each connection point can carry multiple channels depending on the configuration of the audio API stream passing through.

All parameters are defined within the plugin by building a custom parameter object. This object supports floating point numbers, text, boolean and event (button) style interface objects as well as ranges. The parameters are stored locally to the plugin are are accessed by calling getParameters() on the plugin instance, returing a JavaScript object holding the parameters and their respective values. Likewise the converse setParameters() accepts an object holding parameter name / value pairs, facilitating easy manipulation of multiple parameters at once.

Each plugin can also build a custom graphical user interface (GUI), returning a HTML tree for the host to display. If no GUI is generated in the plugin, or the host cannot show the desired GUI, the host must still display all the parameters. This can be styled as the host wishes.

The `PluginFactory` is a parent node defining the interface to all plugin instances and prototypes. The factory has a built-in asynchronous loader for downloading, parsing and storing prototypes from external JavaScript files. The plugin instances are loaded into SubFactories which hold a chain of plugins. The SubFactory, on construction, is passed two audio nodes defining the start and stop points of the chain. New plugin instances are inserted into this chain. Plugins can be moved around the chain, deleted or moved to other SubFactory instances.

3. FEATURES AND SEMANTICS

The PluginFactory can be provided links to data stores allowing plugins to connect to semantic networks or other functions without defining these connections themselves. The factory also handles inter-plugin feature sharing.

Cross-adaptive plugins are defined as plugins whose parameters are controlled by another audio channel's information [5, 6]. Early systems used external microphones and analog components [7], a style still used in live environments [8, 9]. Other effects perform all-channel computations [10, 11] which could be classed as a large auto-adaptive effect as no external channels are used.

For all cross-/auto- adaptive effects, the control signals are calculated based on audio features. If multiple plugins request the same feature from the same audio stream, traditional systems would waste resources re-calculating the same feature since the audio would be routed and feature extraction handled locally. With the factory, the requested

[1] Soundtrap uses the web audio API, available at `https://www.soundtrap.com/`

[2] Available at `https://github.com/Theodeus/tuna/`

[3] Available at: `http://www.semanticaudio.co.uk/jsap`

plugin calculates the desired features and sends them to the factory which dispatches the features to the correct plugin. This messaging system also saves handling potentially complex audio routing paths as well as saving on the complexity of internally building and managing a feature extraction. All plugin outputs are attached to a JS-Xtract feature extraction unit [12].

The system supports several ontologies allowing data to be collected using the standard and stored in a linked database. [13] show how using plugins with a semantic store can benefit audio production, where producers enter terms describing the desired sound and the plugin sets the parameters to match.

The PluginFactory can be fed global information such as tempo, audio sample rate, user information and certain events. The SubFactories are then fed track specific terms such as track name, instrument, group name and audio event locations. Most of these can be described using the studio, event and timeline ontologies. Each plugin itself is given these semantic descriptions enabling it to understand its location. Each plugin also constructs its own semantic information either in definition (name, plugin type etc.) or through use (paramter movement events, effect transforms etc.).

4. DEPLOYMENT

The code is in a single JavaScript file available from `http://www.semanticaudio.co.uk/jsap/` as well as examples and documentation.

A first use-case of the standard is online[4] where the SAFE plugins [13] have been converted into JSAP plugins. These plugins will be used to extend the SAFE dataset through more targeted collection of terms.

5. CONCLUSION

This paper introduces the JSAP standard for building intelligent audio plugins for the web audio API. The standard introduces the two main components that developers will have to use to host and build plugins. The novel feature sharing should enable more complex effects to be built along with the power of the semantic web to drive the next generation of audio effects.

6. REFERENCES

[1] P. Adenot and C. Wilson, "Web Audio API," 2013.

[2] L. Teaford, "Designing synthesizers with web audio," in *Proceedings of the 2nd Web Audio Conference (WAC-2016)* (J. Freeman, A. Lerch, and M. Paradis, eds.), (Atlanta, GA, USA), April 2016.

[3] H. Choi and J. Berger, "WAAX: Web audio api extension.," in *NIME*, pp. 499–502, 2013.

[4] J. Kleimola and O. Larkin, "Web audio modules," in *Proceedings of the Sound and Music Computing 2015*, 2015.

[5] J. D. Reiss, "Intelligent systems for mixing multichannel audio," in *2011 17th International Conference on Digital Signal Processing (DSP)*, pp. 1–6, IEEE, 2011.

[6] V. Verfaille, U. Zolzer, and D. Arfib, "Adaptive digital audio effects (A-DAFx): A new class of sound transformations," *IEEE Transactions on audio, speech, and language processing*, vol. 14, no. 5, pp. 1817–1831, 2006.

[7] D. Dugan, "Automatic microphone mixing," *Journal of the Audio Engineering Society*, vol. 23, no. 6, pp. 442–449, 1975.

[8] E. Perez-Gonzalez and J. Reiss, "Automatic gain and fader control for live mixing," in *IEEE Workshop on applications of signal processing to audio and acoustics*, (New Paltz, NY, USA), pp. 1–4, October 2009.

[9] E. Perez-Gonzalez and J. Reiss, "Automatic mixing: live downmixing stereo panner," in *Proceedings of the 7th International Conference on Digital Audio Effects (DAFx07)*, (Bordeux, France), pp. 63–68, September 2007.

[10] A. Clifford and J. Reiss, "Calculating time delays of multiple active sources in live sound," in *Audio Engineering Society Convention 129*, Audio Engineering Society, 2010.

[11] J. A. Maddams, S. Finn, and J. D. Reiss, "An autonomous method for multi-track dynamic range compression," in *Proceedings of the 15th International Conference on Digital Audio Effects (DAFx-12)*, 2012.

[12] N. Jillings, J. Bullock, and R. Stables, "JS-Xtract: A Realtime audio feature extraction library for the web," in *17th International Society for Music Information Retrieval Conference (ISMIR 2016)*, August 2016.

[13] R. Stables, S. Enderby, B. De Man, G. Fazekas, and J. Reiss, "SAFE: A system for the extraction and retrieval of semantic audio descriptors," in *15th International Society for Music Information Retrieval Conference (ISMIR 2014)*, 2014.

[4] `http://dmtlab.bcu.ac.uk/nickjillings/safe-js/`