From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten to make noise in the browser.

Myles Borins *

Center For Computer Research in Music and Acoustics
mborins@ccrma.stanford.edu

Abstract. The Web Audio API is a platform for doing audio synthesis in the browser. Currently it has a number of natively compiled audio nodes capable of doing advanced synthesis. One of the available nodes the "JavaScriptNode" allows individuals to create their own custom unit generators in pure JavaScript. While a number of people have been writing custom nodes, they are rewriting code that has been perfected multiple times in various languages. The Faust project, developed at Grame CNCM, offers a unique solution to this problem. Faust consists of both a language and a compiler, allows individuals to deploy a signal processor to various architectures including Max/MSP, supercollider, and coreaudio. This paper examines a technology stack that allows for Faust to be compiled to highly optimized JavaScript unit generators that synthesize sound using the Web Audio API.

Keywords: Faust, Flocking, WAAX, JavaScript, asm.js, Emscripten

1 Introduction

The Web Audio API, released in 2011, is "a high-level JavaScript API for processing and synthesizing audio in web applications."[1] Currently there are a number of natively compiled audio nodes within the API capable of doing various forms of synthesis and digital signal processing. One of the available nodes, the "JavaScriptNode", allows individuals to create their own custom unit generators in pure JavaScript, extending the Web Audio API.

While the concept of making interactive sound synthesis environments in the browser is quite exciting, there are two primary factors stopping individuals from investing time into the Web Audio platform; There has not yet been enough Signal Processing related JavaScript code written yet, and some signal processing concepts prove difficult to implement efficiently in a loosely typed language with no memory management.

* Julius O. Smith, Stéphane Letz, Yann Orley, and Colin Clark
[1] https://dvcs.w3.org/hg/audio/raw-file/tip/WebAudio/specification.html

2 WAAX and Flocking

There are a number of projects that are in development abstracting over top of the Web Audio API in order to extend its capabilities, create more complicated unit generators, and allow for a more intuitive syntax. Projects such as WAAX (Web Audio API eXtension)[2] by Hongchan Choi do so while using only the natively compiled nodes in order to ensure optimum efficiency.[1]

While these projects offer a wide variety of unit generators and synthesis modules, they cannot be used to implement all cutting edge techniques. For example the delay node interface does not offer a tap in or tap out function, making wave guide models impossible to implement.[3]

The Flocking audio synthesis toolkit[4] by Colin Clark offers a unique declarative model for doing signal processing within the browser. Unlike WAAX, Flocking has opted to use the "JavaScriptNode" for all unit generators giving users access to a number of signal processing algorithms that can not be achieved using only native nodes.

WAAX and Flocking offer two very different approaches to Web Audio. WAAX offers efficiency, whereas Flocking offers bleeding edge unit generators and declarative syntax. That being said both projects suffer from the same problem, a lack of man hours. There are only so many individuals who have the time and domain specific knowledge necessary to contribute to their development.

3 Faust

The Faust project offers a unique solution to this problem; rather than write code, generate it. Faust, developed at Grame CNCM,"is a programming language that provides a purely functional approach to signal processing while offering a high level of performance." [2] The project is both a language and a compiler, offering the ability to write code once and deploy to many different signal processing environments.

Faust also has a community of scientists and developers who have contributed a large amount of code waiting to be compiled to other platforms. For example Julius Smith has done a substantial amount of research using Faust to implement wave guide synthesis models[3] and Romain Michon has ported the entire STK to Faust.[4]

Creating an efficient compile path from Faust to the Web Audio API would allow for all of the available Faust code to immediately be able to run in the browser. Further, using the architecture compilation model that Faust is famous for we would be able to wrap the compiled Web Audio code to be compatible with all current libraries and frameworks such as WAAX and Flocking.

[2] `https://github.com/hoch/waax`

[3] `https://dvcs.w3.org/hg/audio/raw-file/tip/WebAudio/specification.html#DelayNode-section`

[4] `http://flockingjs.org/`

4 Current Web Audio Implementation

Currently there is an implementation done by Stéphane Letz to compile Faust to Web Audio directly from the Faust Intermediate Representation[5]. This implementation currently has a number of problems. While the implementation is elegant, any algorithms relying on integer arithmetic are currently broken due to JavaScript representing all variables as 32-bit floating point at a lower level.

5 And now for something completely different

5.1 asm.js

A potential solution to the problem of lower level variable represetation is asm.js. Asm.js "is a strict subset of JavaScript that can be used as a low-level, efficient target language for compilers. The asm.js language provides an abstraction similar to the C/C++ virtual machine: a large binary heap with efficient loads and stores, integer and floating-point arithmetic, first-order function definitions, and function pointers."[6]

5.2 Emscripten

As mentioned above asm.js is designed to be a target language for compilers. While it would be possible to work with the current implementation of Faust to Web Audio to utilize asm.js in order to get specific types there are a number of project that currently exist that have development teams working to make the process as efficient as possible.

Emscripten is a project started by Alon Zakai from Mozilla that compiles LLVM(Low Level Virtual Machine) assembler to JavaScript, specifcally asm.js.[5] Currently Faust is able to compile to a barebones C++ file using the minimal.cpp architecture file, the resulting file can painlessly be compiled to asm.js with Emscripten. The upstream Faust2 branch can compile Faust to LLVM byte-code which could potentially be another compilation path.

6 Making Noise

A first approach to automating the compilation process from Faust to Web Audio involves manually implementing each step. The Faust code needs to be compiled to C++ and have the resulting dsp class wrapped in order to allow internal data and member function to be accessed once compiled to JavaScript. The resulting C++ file then needs to be compiled by Emscripten to asm.js. The asm.js needs to once again be wrapped in order to recreate the original C++ object as a JavaScript object. These functions then need to be hooked into the Web Audio API.

[5] `http://faust.grame.fr/index.php/7-news/73-faust-web-art`

As an initial proof of concept it was attempted to compile the example noise.dsp that comes shipped with Faust to JavaScript by way of Emscripten. Noise was a prime candidate for these initial tests due to the integer specific calculations used in its algorithm.

The below sections will describe the process used to manually implement noise in the browser starting from a faust dsp file, and ending with a working Web Audio API JavaScript Object.

6.1 Faust

The noise unit generator starts as a Faust dsp file.

```
random  = +(12345)~*(1103515245);
noise   = random/2147483647.0;
process = noise * 0.5;
```

In order to compile to C++ in a manner that will be compatible with Emscripten we must use the follow command.

```
faust -a minimal.cpp -i -uim -cn Noise  dsp/noise.dsp \
  -o cpp/faust-noise.cpp
```

The resulting file must then be wrapped in a way to allow its functions to be accessed once compiled to JavaScipt. The source for this wrapper can be found within the code repository on GitHub, a link for which is found at the end of this paper. The wrapper creates five functions, a constructor, a destructor, a function to get the number of inputs / outputs, and a compute function. These functions are all placed in an 'extern' in order to avoid name space obfuscation.

6.2 Emscripten & asm.js

Once the wrapper has been concatenated with the Faust compiled C++ it can then be compiled by Emscripten to asm.js. This is done with the following command

```
emcc cpp/faust-noise.cpp -o js/faust-noise-temp.js \
  -s EXPORTED_FUNCTIONS="['_NOISE_constructor', \
  '_NOISE_destructor', '_NOISE_compute', \
  '_NOISE_getNumInputs', '_NOISE_getNumOutputs']"
```

Note the exported functions, which are referencing the five wrapper functions mentioned in the previous step. This is required to stop Emscripten from obfuscating the names of the functions when certain optimization flags are thrown during compilation

6.3 Web Audio Api

Once the asm.js code has been compiled a JavaScript wrapper is used break out the functionality of the code into JavaScript functions. As well, the correct context for generating audio in the browser needs to be set up within the Web Audio API, connecting the generated data from the Faust generated functions to the correct Web Audio API functions in order to generate sound. Again this wrapper can be found in the source repository on GitHub

7 Results

Using the above methods a Faust compiled WebAudio noise unit generator was successfully created. The result can be found at:

`https://ccrma.stanford.edu/~mborins/420b/demo/`

8 Limitations

There were a few limitations that were found with the current compilation method. First, we can currently only use Faust code that requires no input and only a single output channel. This is due to the fact that managing memory in C++ while maintaing persistence when compiled to JavaScript can be tricky. Specifically because Faust represents inputs and outputs as a float **. More research needs to be done in to JavaScript pointer arithmetic before this problem will be able to be solved.

A second problem is that Faust relies on UI as a control interface to make any changes to the state of a unit generator. As such current code that relies on this would need to be modified prior to working with the current tool chain. Rather than requiring individuals to manually augment all previously written code, a solution needs to be found to allow current UI elements to be interpreted in a way that changes can be made through events in JavaScript.

Finally the current compilation path is hard coded, there is no way to compile other unit generators without going through the entire process again from scratch. A solution to this problem will most likely be simple, but implementing it without polluting the global name space of JavaScript could prove challenging.

9 Looking Forward

While the above mentioned limitations do needs to be worked on, benchmarks should be performed on the currently compiling code to ensure that this compilation method is in fact a good direction. Colin Clark has offered to implement some benchmark tests against his Flocking library as well as a number of other WebAudio based implementations if he can be provided a working version of freeverb. Due to this generous offer, creating a working freeverb will most likely be the next logical step in for this research.

As well, Stéphane Letz and Yann Orley have expressed a desire to approach this problem using their original method of going directly from the Faust Intermediate Representation to JavaScript. This would avoid moving from a functional language to an object oriented language back to a function language, which has proven somewhat inelegant. It may prove appropriate once the Emscripten method can be benchmarked to put time in to developing this more direct compilation path so that the results from the two methods can be compared.

10 Conclusion

The results of this research have shown that it is indeed possible to get compiled Faust code running properly in the browser. This is very exciting, as if the benchmarks are encouraging we will be able to use the resulting code to greatly expand the ecosystem for digital signal processing in the browser.

One of the most exciting parts of the results are that if this process can be perfected we will continue to see improvements in efficiency as the various technologies we are relying on continue to improve. As JavaScript becomes more efficient, so does the compiled code. As WebAudio becomes more stable, so does the compiled code. As asm.js optimizations improve in the browser, we get the optimizations for free. Simply put, even if the resulting benchmarks prove to not be competitive with current hand written JavaScript, it will only get better with time while requiring minimal time maintaing the project.

11 Code Repository

The entire source code can be found online on GitHub at:
`https://github.com/TheAlphaNerd/faust2webaudio`

References

1. H. Choi, J.Berger., "WAAX: Web Audio API eXtension" *Proceedings of the Thirteenth New Interfaces for Musical Expression Conference*, 2013.
2. Y. Orlarey, D. Fober, and S. Letz., "FAUST : an Efficient Functional Approach to DSP Programming," *New Computational Paradigms for Computer Music*, pp. 65-96, Editions DELATOUR FRANCE, 2009.
3. J. Smith, J. Kuroda, J. Perng, K. V. Heusen, and J. Abel, "Efficient computational modeling of piano strings for real-time synthesis using mass-spring chains, coupled finite differences, and digital waveguide sections" *Acoustical Society of America, Program of the 2nd Pan-American/Iberian Meeting on Acoustics (abstract and presentation)*, Cancun, Mexico, Nov. 1519, 2010, invited presentation. Presentation overheads: https://ccrma.stanford.edu/ jos/pdf/ASA-2010-jos.pdf
4. R. Michon and J. O. Smith, "Faust-STK: A set of linear and nonlinear physical models for the Faust programming language", in *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11),* Paris, France, September 1923, 2011

5. A. Zakai. “Emscripten: an llvm-to-javascript compiler.” *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion.* ACM, 2011.
6. *asm.js - frequently asked questions.* `http://asmjs.org/faq.html`.