From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten

Myles Borins

▶ To cite this version:

Myles Borins. From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten. Linux Audio Conference (LAC-14), May 2014, Karlsruhe, Germany. hal-03162964

HAL Id: hal-03162964
https://hal.science/hal-03162964

Submitted on 8 Mar 2021

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

From Faust to Web Audio: Compiling Faust to JavaScript using Emscripten

Myles Borins
Center For Computer Research in Music and Acoustics
Stanford University
Stanford, California
United States,
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 "ScriptProcessorNode" allows individuals to create their own custom unit generators in pure JavaScript. The Faust project, developed at Grame CNCM, consists of both a language and a compiler and allows individuals to deploy a signal processor to various languages and platforms. 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

WebAudio, Faust, Emscripten, JavaScript, asm.js

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 "ScriptProcessorNode", 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, many factors stop individuals from investing time into the Web Audio platform. Ignoring the constraints of a single threaded environment there appear to be two primary limitations when working with web audio: 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.

[1] https://dvcs.w3.org/hg/audio/raw-file/tip/WebAudio/specification.html

2 Some Context

2.1 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.[H. Choi and J.Berger, 2013]

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 internally manage all signal generation and using a single "ScriptProcessorNode" to hand off precomputed buffers of samples.

WAAX and Flocking offer two very different approaches to Web Audio. WAAX offers efficiency, whereas Flocking offers an extensible architecture and declarative syntax in which web developers can write their own first-class custom unit generators. 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.

2.2 Introduction to Faust

The Faust project offers a unique solution to this problem; rather than write code, generate

[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/

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." [Orlarey et al., 2009] 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[Smith et al., 2010] and Romain Michon has ported the entire STK to Faust.[Michon and Smith, 2011]

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.

2.3 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]. While the implementation is elegant, any algorithms relying on integer arithmetic are currently broken due to JavaScript representing all Numbers as 32-bit floating point at a binary level.

2.4 Introduction to asm.js

One way to do integer arithmetic with cross-browser support is asm.js. The asm.js specification[6] outlines a 'strict subset' of JavaScript that offers a unique programming model. Through the use of typed arrays[7] it is possible to do integer and floating-point arithmetic. This is done with a virtual machine that gives developers access to a heap and functions to be used to manage memory and perform arithmetic operations.

While it would have been possible to use Stéphane Letz's work as a starting point and extend the current WebAudio architecture to utilize the asm.js susbset, it would require quite a bit of overhead. Not only would integer and floating point specific interpretation need to be implemented, but a functional virtual machine would need to have been developed in order to take advantage of asm.js.

Further, we would not see any of the optimization benefits that one would get from a modern compiler such as gcc or clang. In the spirit of this project, a search was done to find a way to automate away the need to worry about all of these complications.

2.5 Introduction to Emscripten

Emscripten is a project started by Alon Zakai from Mozilla that compiles LLVM(Low Level Virtual Machine) assembler to JavaScript, specifically asm.js.[Zakai, 2011] The platform is both a compiler and a virtual machine capable of running C and C++ code in the browser.

Emscripten gives you an interface to break out C functions so that they can be called using JavaScript. It also provides functions for managing memory in the virtual machine your C code is running. These functions allow you to allocate new memory to be operated on (in the case of sound buffers), and the ability to manipulate memory in the heap (in order to change parameters).

Currently Faust is able to compile to a 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 offers another potential compilation path.

3 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 functions 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 provide an intuitive JavaScript interface that will operate on the dsp object running in the Emscripten virtual machine. Finally an interface between the Emscripten virtual machine and WebAudio needs to be made to hand off samples that need to be sonified.

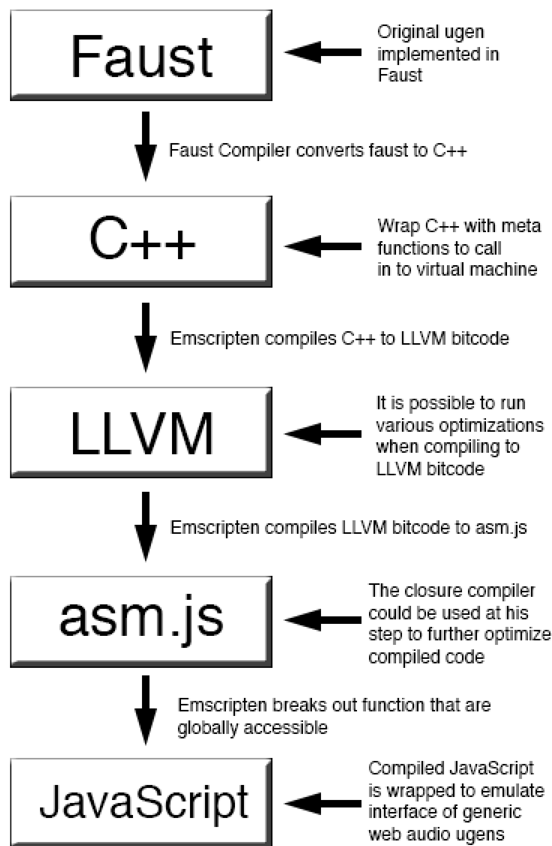
[5] http://faust.grame.fr/index.php/7-news/73-faust-web-art

[6] http://asmjs.org/spec/latest/

[7] https://developer.mozilla.org/en-US/docs/Web/JavaScript/Typed_arrays?redirectlocale=en-US&redirectslug=JavaScript/Typed_arrays

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.



3.1 Faust Source

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
```

This tells faust to compile the above code using the *minimal.cpp* architecture file, to call the object being created Noise, and to include all necessary header files and dependencies.

The resulting C++ code need to be wrapped with a series of meta functions that can be called to operate on objects living in the virtual machine. A constructor and destructor are implemented in order to create objects and properly clean them up, and a compute function is then used to grab the latest frame of samples from the unit generator. In order to change the state of the unit generator after its instantiated in the heap a number of other functions are available to create a map of the ugen's parameters, and get / set values.

3.2 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',\
  '_NOISE_getNumParams',\
  '_NOISE_getNextParam']"
```

Note the exported functions, which are referencing the seven 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, and to make access to them available in the global namespace of JavaScript.

3.3 Web Audio Api

Once the asm.js code has been compiled a JavaScript wrapper is used to 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

4 Results

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

`http://thealphanerd.io/examples/faust2webaudio/`

4.1 Other Examples

This process has been repeated for a number of other unit generators including a sine oscillator, freeverb, and a 16th order FDN reverb (included in the Faust distribution as Reverb Designer). All three examples work in the browser, although the 16th order FDN takes a few seconds to get going. Once the unit generators have been compiled to JavaScript it is quite easy to connect them to each, and other web audio components.

Below is an example of how to create a noise object, and apply freeverb to its output. Both of these objects have been compiled using Faust2WebAudio. This example can be found online at `http://thealphanerd.io/examples/faust2webaudio/freeverb.html`

```
var noise = faust.noise();
var freeverb = faust.freeverb();
noise.connect(freeverb);
noise.update("Volume", 0.1);
freeverb.update("Damp", 0.75);
freeverb.update("RoomSiz", 0.75);
freeverb.update("Wet", 0.75);
freeverb.play();
```

5 Limitations

Currently the automation layer has not yet been completed. While the wrapper scripts have all been generically written, hand written bash scripts utilizing tools such as sed are currently being used to compile individual unit generators. A next step would involve moving the generic wrappers in to their own architecture file and relying on the Faust build system to handle generic compilation.

Another major limitation is that I am currently utilizing a separate instance of the Emscripten virtual machine for each unique unit generator. This is an unfortunate side effect of the current compilation method. Emscripten includes the virtual machine at the head of every compiled js file. There is an option to statically link a number of compiled js files to a single optimized file with redundancies removed, but I am concerned about the implications of that workflow.

A developer would be required to supply all of the faust objects at once, and not have the ability to swap in and out files at their leisure. Unless their is an intuitive and fast way to compile this final file, it will make it difficult for individuals to add new unit generators on the fly as they are composing in the browser.

One solution is to utilize a JavaScript task runner such as grunt to watch for changes in specific directories / files and to properly compile and statically link multiple files on the fly.

6 Looking Forward

While the above mentioned limitations do need to be worked on, benchmarks should be performed on the currently compiled code to ensure that this compilation method is in fact a good direction.

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.

7 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.

8 Code Repository

Find the source online at:
`https://github.com/TheAlphaNerd/faust2webaudio`

9 Acknowledgements

I would like to thank Julius O. Smith, Stéphane Letz, Yann Orley, and Colin Clark for their guidance and support.

References

H. Choi and J.Berger. 2013. Waax: Web audio api extension. In *Proceedings of the Thirteenth New Interfaces for Musical Expression Conference.*

R. Michon and J. O. Smith. 2011. 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)*, pages 65–96, September.

Y. Orlarey, D. Fober, and S. Letz. 2009. Faust : an efficient functional approach to dsp programming. In *New Computational Paradigms for Computer Music*, pages 65–96. Editions DELATOUR FRANCE.

J. Smith, J. Kuroda abd J. Perng abd K. V. Heusen, and J. Abel. 2010. Efficient computational modeling of piano strings for real-time synthesis using mass-spring chains, coupled finite differences, and digital waveguide sections. In *Acoustical Society of America, Program of the 2nd Pan-American/Iberian Meeting on Acoustics (abstract and presentation)*, pages 65–96, Nov.

A. Zakai. 2011. Emscripten: an llvm-to-javascript compiler. In *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion.* ACM.