From Faust to Flocking: Declarative Control of the Faust Synthesis Language in the Browser.

Myles Borins[1] *

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

Abstract. The Flocking audio synthesis toolkit, developed by Colin Clark, offers a unique declarative model for doing signal processing within the browser. It's JSON-based syntax combined with the interpreted nature of the JavaScript language make it an ideal candidate for creating interactive sound synthesis environments. At the current time the toolkit has very few unit generators when compared to more traditional environments. The Faust project, developed at Grame CNCM, offers a unique solution to this problem. Faust consists of both a language and a compiler, allowing individuals to deploy a signal processor to various industry standard environments (max/msp, supercollider) and formats (vst, au, jack). This paper examines a technology stack that allows for Faust to be compiled to highly optimized Flocking unit generators that synthesize sound using the web audio api.

Keywords: Faust, Flocking, JavaScript, asm.js, emscripten

1 Introduction

The Flocking audio synthesis toolkit offers a unique declarative model for doing signal processing within the browser. It's JSON-based syntax combined with the interpreted nature of the JavaScript language make it an ideal candidate for creating interactive sound synthesis environments.[1] The toolkit is built on top of the Web Audio Api[2] offering support out of the box on almost every web-capable device. Deploying to the browser also enables one to leverage a wide variety of web technologies ranging from User Interface components to large scale development frameworks.

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, Yann Orley, and Colin Clark

[1] http://flockingjs.org/

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

The Faust project offers a unique solution to the first 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." [1] The project is both a language and a compiler, offering the ability to write code once and deploy to many different signal processing environments.

This section will continue to describe the current state of Faust -¿ Web audio compilation, specifically the problems with the current process which is inline with the above mentioned problems (efficiency, language limitations). It will then introduce Emscripten[3] a cross compiling tool designed to compile C/C++ into efficient asm.js code. Asm.js [4] will also be introduced as a subset of javascript that "provides a model closer to C/C++ by eliminating dynamic type guards, boxed values, and garbage collection."[2].

2 A Simple Approach

A first approach to automating the compilation process from Faust to Flocking involves manually implementing each step. The Faust code needs to be compiled to C++, from there it needs to compiled to LLVM bytecode by either Faust2 or emscripten. Once compiled to bytecode emscripten needs to be used to compile to asm.js. The asm.js needs to have binding functions written to get access to the compiled code within the javascript environment. These functions then need to be hooked into the web audio api. Finally bindings need to be written between these functions and the Flocking toolkit.

Making Noise: A first test was to attempt to compile the example noise.dsp that comes shipped with Faust. Noise was chosen as it is currently broken in the current iteration of Faust -¿ Web Audio Api. Due to the integer specific calculations required for the Faust implementation of noise, a satisfactory result was unable to be accomplished using the current system.

The below sections will be used to describe the process used to manually implement noise in the browser starting from a faust dsp file, and ending with a working Flocking Synthdef.

2.1 Faust

The noise unit generator starts as a Faust dsp file. It's code is fairly straight forward.

```
random  = +(12345)~*(1103515245);
noise   = random/2147483647.0;
process = noise * vslider("Volume[style:knob]", 0, 0, 1, 0.1);
```

[3] https://github.com/kripken/emscripten
[4] http://asmjs.org/

Faust is able to compile this to optimized C++ out of the box. Currently if you compile without selecting an architecture file then a number of things are missing from the C++, specifically the include files as well as the definition for the dsp object (which it appears is dependent on audio architecture specified). Is it possible to get the appropriate headers and class definitions compiled into the C++ without selecting an architecture file? If not what type of information should be generated for the web audio architecture

Another thing worth exploring is wether or not the faust code should be compiled to LLVM bytecode rather than C++, this will require a conversation with Stephane and Yann to figure out.

2.2 Emscripten & asm.js

Once you have the optimized C++ properly compiled (or llvm bytecode) it can then be compiled to asm.js using Emscripten. There are a handful of features of C++ that are not supported by Emscripten included vectorized optimizations, so it might necessary to do a number of tests before being able to properly compile asm.js

2.3 Web Audio Api

Once the asm.js code has been compiled a JavaScript file needs to be written in order to both 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.

2.4 Flocking

Once the web audio unit generators have been properly implemented, the final stage involves wrapping the web audio unit generator to be controlled by the Flocking toolkit. This process may or may not be trivial depending on how the above web audio code ends up being generated.

3 Automation

All the above steps should be able to be automated, if time permits this section will examine how an architecture file can be written in order to compile any faust dsp file to the web audio api and Flocking.

4 Results

This will be a quick look at the resulting generated signal, if it is indeed an accurate instance of noise, and how the signal compares to faust compiled noise generated in other signal processing environments. It may also be worth examining performance benchmarks, if a good model can be found to do so.

5 Looking Forward

This section will discuss current implementations of the above used technologies, and what type of improvements can be expected over time. Asm.js is getting both AOT and JIT optimizations built directly in the browser, currently only in Firefox, but when implemented in Chrome there will be some great improvements to efficiency.

Web audio api is currently only supported in Chrome, but work has begun to implement it in firefox. This creates an interesting problem where the most efficient browser to run the compiled code is unable to actually do anything with it due to not supporting web audio api yet.

Talk about How other web based frameworks can bootstrap on this idea in order to get use of the various unit generators that can be generated from this process. Once the compiled code is hooked into the web-audio api bindings can be written for any framework.

Finally discuss a conclusion about findings of the project

References

1. Y. Orlarey, D. Fober, and S. Letz. *FAUST : an Efficient Functional Approach to DSP Programming*, page 6596. 2009.
2. asm.js - frequently asked questions. http://asmjs.org/faq.html.