FAUCK!! HYBRIDIZING THE FAUST AND CHUCK AUDIO PROGRAMMING LANGUAGES

Ge Wang, Romain Michon

FAUCK!! HYBRIDIZING THE FAUST AND CHUCK AUDIO PROGRAMMING LANGUAGES

Ge Wang
CCRMA, Stanford University
ge@ccrma.stanford.edu

Romain Michon
CCRMA, Stanford University
rmichon@ccrma.stanford.edu

ABSTRACT

This paper presents a hybrid audio programming environment, called FAUCK, which combines the powerful, succinct Functional AUdio STream (FAUST) language with the strongly-timed CHUCK audio programming language. FAUCK allows programmers to on-the-fly evaluate FAUST code directly from CHUCK code and control FAUST signal processors using CHUCK's sample-precise timing and concurrency mechanisms. The goal is to create an amalgam that plays to the strengths of each language, giving rise to new possibilities for rapid prototyping, interaction design and controller mapping, pedagogy, and new ways of working with both FAUST and CHUCK. We present our motivations, approach, implementation, and preliminary evaluation. FAUCK is open-source and freely available.

1. INTRODUCTION

A variety of computer music programming languages exist for the same reason there are many different types of tools: each is well-suited to different types of tasks, and speaks to different aesthetic and pragmatic preferences of the programmer. FAUST and CHUCK are two audio programming languages that effectively illustrate this point. FAUST (Functional AUdio STream) [1–4] embraces a declarative and functional paradigm, is succinct, tailored to expressively describe low-level digital signal processing (DSP) algorithm, and generates optimized, efficient synthesis modules. CHUCK, [5, 6], on the other hand, is imperative, designed around a notion of temporal determinism that includes sample-synchronous timing and concurrency (called *strongly-timed*), tailored for precise control, readability, and an on-the-fly rapid-prototyping mentality [7]. Yet, they share the general goal of sound synthesis for musical applications and both are text-based languages (e.g., not graphical patching).

What happens when one combines these two languages? More to the point, can these languages be combined in such a way to take advantage of the respective strengths of both? Furthermore, that the two languages seem vastly different in syntax, semantics, and personality is all the more reason to explore their intersections (i.e., why try to combine two things are already similar?). Might their profound differences give rise to something different from either alone? Such curiosities provide the primary motivation for our exploration in hybridizing FAUST and CHUCK.

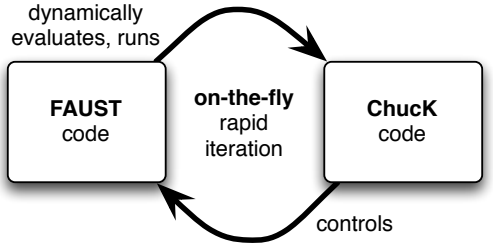

Figure 1. New FAUCK model.

FAUCK is a tight integration between FAUST and CHUCK that attempts to infuse their respective characteristics, reinforcing and even augmenting each language with aspects of the other (Figure 1). It is designed such that programmers can embed and evaluate FAUST code directly from within a CHUCK program, take full advantage of CHUCK's sample-synchronous time-based control and concurrency, and do all of this on-the-fly. At the same time, this makes available to CHUCK the entire existing body of FAUST programs, ready to be used for synthesis and interaction design. In combination, FAUCK provides a different hybridized way to rapidly prototype sample-precise audio synthesis code in FAUST and control it precisely using CHUCK.

2. RELATED WORK

2.1 Existing Paradigm

Thanks to its architecture system [8], FAUST can be used to easily build custom DSP modules that generate or process samples as part of a larger host. The traditional paradigm of incorporating FAUST into computer music systems involves a pipeline that 1) generates C++ code from FAUST code, 2) compiles into a plug-in, and 3) runs as part of a host software system [1] (e.g., *PureData* [2], *Max/MSP* [3], *SuperCollider* [4], etc.). In this regime, there is a new plug-in created from any given FAUST program (Figure 2). Such a system currently exists for compiling a given FAUST program into a CHUCK *chug-in* [9].

[1] http://faust.grame.fr/Documentation/ contains an exhaustive list of the FAUST architectures.
[2] https://puredata.info/.
[3] https://cycling74.com/products/max/.
[4] http://supercollider.github.io/.

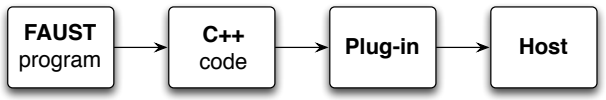

Figure 2. Traditional FAUST code to plug-in model.

2.2 Dynamic FAUST Extensions

Modules created using the technique described in §2.1 are static and cannot be modified once they are compiled. `libfaust` [10] which is part of the FAUST2 distribution [5] allows to embed the FAUST compiler in any program written in C++ to dynamically generate DSP modules on the fly (see §4).

Since its introduction in 2012 `libfaust` has been used in various projects to integrate the FAUST compiler to existing platforms or to create new one. The FAUST backend [11] of the *Pure* programming language [12] that predates the creation of `libfaust` was used as the basis for the design of this tool.

`libfaust` was used for the first time in *PureData* within a dynamic *external* that can be turned into any DSP object by modifying the FAUST code linked to it [13]. Similarly `faustgen` [14] is a dynamic *MaxMSP* external borrowing the paradigm used in `gen` [15] to create custom unit generators. *CSound* [6] now hosts a dynamic *opcode* functioning the same way than the two previous examples [16]. `libfaust` has also been used under different forms within the *Web Audio API* [17] to create custom dynamic *nodes* [18, 19]. More recently, FAUST has been integrated to the *JUCE* [7] platform [20] and the *Processing* [8] programming language.

Finally, `libfaust` is at the core of the FAUSTLIVE "just in time" FAUST compiler [21] allowing to write FAUST code in a text file and generating the corresponding standalone audio application almost instantly.

3. THE FAUCK APPROACH

3.1 Using FAUCK

FAUST objects can be used easily in any CHUCK code through a *chugin* called `Faust`. For example, a new FAUST unit generator (e.g., an audio DSP effect that takes an input from CHUCK) can be declared as follow:

```
adc => Faust foo => dac;
```

In the case where `foo` would be a synthesizer, the `adc` would be ignored and we could simply write:

```
Faust foo => dac;
```

Any FAUST program can be associated with `foo` and dynamically evaluated by calling the `eval` method.

```
foo.eval(`process=osc(440);`);
```

For brevity and convenience, several common libraries (`music.lib`, `filter.lib`, `oscillator.lib`, `effect.lib`, `math.lib`) are, by default, automatically imported by FAUCK. Furthermore, note the use of the backtick (`` ` ``) to delineate the inline FAUST code – this removes the need to manually escape single and double quotation marks used in the FAUST code.

Alternately, the same object can load a FAUST program from the file system by invoking `compile` and providing a path to a FAUST `.dsp` file:

```
foo.compile("osc.dsp");
```

Next, the `v` method can be called at anytime to change the value of a specific parameter defined on the FAUST object that is specified by its path (`v` stands for "value"; we chose this abbreviation in anticipation that most program will invoke this method often). For example, here we create a sine wave oscillator whose only parameter is its frequency (`freq`) and we set it to 440Hz:

```
foo.eval(`
    frequency = nentry("freq",
        200,50,1000,0.01);
    process = osc(frequency);
`);
foo.v("freq",440);
```

Finally, the `dump` method can be called at any time to print a list of the parameters of the FAUST object as well as their current value. This is useful to observe large FAUST programs that have a large number of parameters in complex grouping paths. Programmers can also directly copy the path of any parameter to control for use with the `v` method.

3.2 Examples

3.2.1 A Simple Example

The following example puts together the different elements given in §3.1 by implementing a simple sine wave oscillator (specified in FAUST) whose frequency and gain are randomly changed every 100ms (controlled in CHUCK).

```
// connect a Faust object to ChucK dac
Faust foo => dac;
// evaluate
foo.eval(`
    frequency = nentry("freq",
        200,50,1000,0.01) : smooth(0.999);
    gain = nentry("gain",
        1,0,1,0.01) : smooth(0.999);
    process = osc(frequency)*gain;
`);
// ChucK time loop
while( true ){
    // control frequency
    foo.v("frequency", Math.random2f(50,800));
    // control gain
    foo.v("gain", Math.random2f(0,1));
    // advance time
    100::ms => now;
}
```

3.2.2 An Advanced Example

Making use of CHUCK's sample-precise timing and concurrency mechanisms, it is straightforward to mix CHUCK

[5] `https://sourceforge.net/p/faudiostream/code/ci/faust2/tree/`.
[6] `http://www.csounds.com/`.
[7] `https://www.juce.com/`.
[8] `https://processing.org/`.

unit generators with FAUST objects to create hybrid elements. In the following example, a string physical model implemented in an external FAUST file is filtered by a *crybaby* effect evaluated in the CHUCK file and declared in `effect.lib` (FAUST library). The `wah` parameter of the *crybaby* effect is modulated by an *LFO*[9] declared as a CHUCK object. The string physical model is controlled in concurrent CHUCK shreds, spawned through the `spork` operator.

```
// instantiate and connect 2 Faust modules
Faust string => Faust cryBaby => dac;
// LFO using ChucK UGen
SinOsc LFO => blackhole; 6 => LFO.freq;

// load FAUST program; map to Faust object
string.compile("string.dsp");
// evaluate code; crybaby from effect.lib
cryBaby.eval(`process = crybaby_demo;`);

// generates random notes
fun void notes(){
    while( true ){
        // new note
        string.v("gate",0);
        10::ms => now;
        string.v("gate",1);
        // with random frequency
        string.v("freq",
        Math.random2f(80,800) );
        100::ms => now;
    }
}
// modulates the cry baby with the LFO
fun void lfoWah(){
    while( true ){
        cryBaby.v(
            "/CRYBABY/Wah_parameter",
            (LFO.last()*0.5+0.5) );
        1::samp => now; // every sample!
    }
}

spork ~ notes();
spork ~ lfoWah();
while( true ){
    10::ms => now;
}
```

4. IMPLEMENTATION

FAUCK is implemented as a *chugin* (CHUCK plugin), simply named `Faust`. The chugin, when installed, shows up as the FAUST unit generator in CHUCK, and can be used in any number or configuration from CHUCK (as shown in the code example above). FAUCK internally manages the interface between CHUCK and the just-in-time FAUST compiler. Each instance of the `Faust` unit generator maintains a map of parameters indexed on the full parameter path, which enables real-time look-up and direct manipulation of the named parameters.

FAUCK is written in C++ and is made possible by `libfaust`, an embedded version of the FAUST compiler, capable of generating *LLVM* bitcode (*LLVM IR*) instead of C++. `libfaust` invokes the *LLVM* compiler with the bitcode to emit into an efficient binary format that can be run dynamically [10].

[9] *Low Frequency Oscillator*.

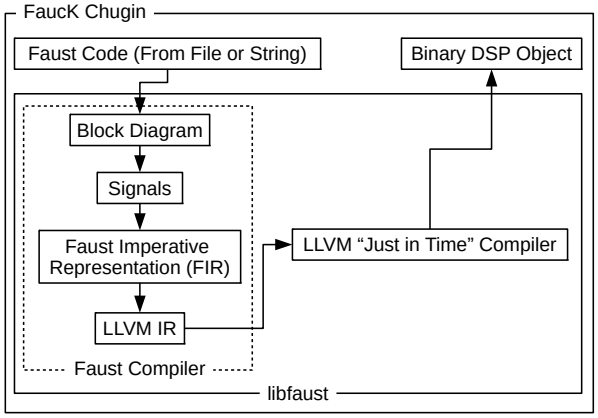

Figure 3. Overview of the FAUCK *chugin*.

Real-time performances seem promising. The advanced example in the previous section runs below 7% CPU utilization on a MacBook Pro from 2012 (this includes the baseline overhead of the CHUCK virtual machine). For comparison, a similar example, missing the `notes()` function, fully implemented in FAUST, and compiled as a standalone *CoreAudio* application, runs at approximately 5% on the same computer.

5. CONCLUSIONS AND FUTURE WORK

As it turns out (and as we had hoped), the pronounced differences between FAUST and CHUCK actually make it easier to articulate a useful intersection between the two languages. FAUST code tends to operate at the within-unit generator DSP level, whereas CHUCK's unique strength lies in the strongly-timed, concurrent control of unit generators. This makes for easy division of labor in FAUCK. Furthermore, it is straightforward in this model to mix FAUST modules with existing CHUCK unit generators.

More generally, FAUCK provides a few unique benefits:

- FAUCK combines the control capabilities of CHUCK to the efficiency and concise, expressive DSP programming of FAUST.
- FAUST has no scheduler/control system — something CHUCK was specifically designed for (making for example, polyphonic FAUST objects is now easy with FAUCK).
- The `Faust` modules provides seamless integration with CHUCK unit generators, enabling a new type of rapid prototyping and experimentation in FAUST, CHUCK, or in tandem.
- Overall, FAUCK provides a new way to work with FAUST, while expanding CHUCK's synthesis capabilities to include the large and growing body of FAUST code. FAUCK presents a clear deterministic all-in-one place delineation of both FAUST and CHUCK code, which can potential benefit both research and classroom settings.

For future work, we'd like to continue experimenting with features in FAUCK to further facilitate this hybridization. For example, while it's possible for CHUCK code to control all parameters in a FAUST program, regardless of the type of UI defined for the parameter, it would be convenient if FAUCK can provide functionality to auto-generate miniAudicle user interfaces (MAUI) [22] from any FAUST code. Also, since both FAUST and CHUCK are text-based, it would be intriguing to further deepen the intersection with dynamically- or self-generating FAUST code from within CHUCK. Also, we are beginning to apply FAUCK in computer music pedagogical settings, as well as towards DSP-based physical modeling and computer-mediated instruments design for laptop orchestras.

FAUCK is open-source and is part of the `ChuGin` repository: `https://github.com/ccrma/chugins`.

Acknowledgments

We thank our colleagues at CCRMA and GRAME for their suggestions and support.

6. REFERENCES

[1] Y. Orlarey, S. Letz, and D. Fober, *New Computational Paradigms for Computer Music*, Paris, France, 2009, ch. FAUST : an Efficient Functional Approach to DSP Programming.

[2] Y. Orlarey, D. Fober, and S. Letz, "An algebra for block diagram languages," in *Proceedings of the International Computer Music Conference*, 2002.

[3] 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*, 2011.

[4] R. Michon, J. O. Smith, and Y. Orlarey, "Mobilefaust: a set of tools to make musical mobile applications with the faust programming language," in *Proceedings of the Linux Audio Conference*, 2015.

[5] G. Wang, "The chuck audio programming language," Ph.D. dissertation, Princeton University, 2008.

[6] G. Wang, P. R. Cook, and S. Salazar, "Chuck: A strongly timed computer music language," *Computer Music Journal*, vol. 39, no. 4, pp. 10–29, 2015.

[7] G. Wang and P. R. Cook, "On-the-fly programming: Using code as an expressive musical instrument," in *New Interfaces for Musical Expression*, 2004.

[8] D. Fober, Y. Orlarey, and S. Letz, "Faust architectures design and osc support," in *Proceedings of the 14th International Conference on Digital Audio Effects (DAFx-11)*, Paris, France, September 2011.

[9] S. Salazar and G. Wang, "Chugens, chubgraphs, and chugins: 3 tiers for extending chuck," in *Proceedings of the International Computer Music Conference*, 2012.

[10] S. Letz, D. Fober, and Y. Orlarey, "Comment embarquer le compilateur faust dans vos applications?" in *Proceedings of the Journées de l'Informatique Musicale*, Paris, France, May 2013.

[11] A. Gräf, "An llvm bitcode interface between pure and faust," in *Proceedings of the Linux Audio Conference (LAC-11)*, Maynooth, Ireland, May 2011.

[12] A. Gräf, "Signal processing in the pure programming language," in *Proceedings of the Linux Audio Conference (LAC-09)*, Parma, Italy, April 2009.

[13] A. Gräf, "pd-faust: An integrated environment for running faust objects in pd," in *Proceedings of the Linux Audio Conference (LAC-12)*, Stanford, California, April 2012.

[14] "Faustgen," 2012. [Online]. Available: http://faust.grame.fr/news/2012/12/11/faustgen.html

[15] "Gen documentation," 2016. [Online]. Available: https://cycling74.com/wiki/index.php?title=gen~_For_Beginners

[16] V. Lazzarini, "Faust programs in csound," *Revue Francophone d'Informatique Musicale*, no. 4, Fall 2014.

[17] "The web audio api," 2015. [Online]. Available: https://www.w3.org/TR/webaudio/

[18] S. Denoux, Y. Orlarey, S. Letz, and D. Fober, "Compose with faust in the web," in *Proceedings of the Web Audio Conference*, Paris, France, January 2015.

[19] S. Letz, S. Denoux, Y. Orlarey, and D. Fober, "Faust audio dsp language in the web," in *Proceedings of the Linux Audio Conference (LAC-15)*, Mainz, Germany, April 2015.

[20] O. Larkin, "Using the faust dsp language and the libfaust jit compiler with juce," in *Proceedings of the JUCE Summit*, London, UK, November 2015.

[21] S. Denoux, S. Letz, Y. Orlarey, and D. Fober, "Faustlive: Just-in-time faust compiler... and much more," in *Proceedings of the Linux Audio Conference (LAC-12)*, Karlsruhe, Germany, April 2014.

[22] S. Salazar, G. Wang, and P. R. Cook, "miniaudicle and chuck shell: New interfaces for chuck development and performance." in *Proceedings of the International Computer Music Conference*, 2006.