faust2api: a Comprehensive API Generator for Android and iOS

Romain Michon, Julius Smith, Chris Chafe
CCRMA
Stanford University
Stanford, CA 94305-8180
USA
{rmichon,jos,cc}@ccrma.stanford.edu

Stéphane Letz, Yann Orlarey
GRAME
Centre National de Création Musicale
11 Cours de Verdun (Gensoul)
69002, Lyon
France
{letz,orlarey}@grame.fr

Abstract

We introduce `faust2api`, a tool to generate custom DSP engines for Android and iOS using the FAUST programming language. FAUST DSP objects can easily be turned into MIDI-controllable polyphonic synthesizers or audio effects with built-in sensors support, etc. The various elements of the DSP engine can be accessed through a high-level API, made uniform across platforms and languages.

This paper provides technical details on the implementation of this system as well as an evaluation of its various features.

Keywords

FAUST, iOS, Android, Mobile Instruments

1 Introduction

Mobile devices (smart-phones, tablets, etc.) have been used as musical instruments for the past ten years, both in the industry (e.g., GarageBand[1] for iPad, Smule's apps,[2] moForte's GeoShred,[3] etc.), and in the academic community ([Tanaka, 2004], [Geiger, 2006], [Gaye et al., 2006], [Essl and Rohs, 2009] and [Wang, 2014]).

Implementing real-time Digital Signal Processing (DSP) engines from scratch on mobile platforms can be hard using standard audio APIs provided with common operating systems (we'll only cover iOS and Android here). Indeed, CoreAudio on iOS and OpenSL ES on Android are relatively low-level APIs offering customization possibilities not needed by most audio app developers. Fortunately, there exist several third party cross-platform APIs to work with real-time audio on mobile devices at a higher level (e.g., SuperPowered,[4] JUCE,[5] etc.). Additionally, several open-source tools allow to use objects written in common computer music languages such as PureData:[6] *libpd* [Brinkmann et al., 2011] and Csound:[7] *Mobile Csound Platform* (*MCP*) [Lazzarini et al., 2012] on mobile platforms.

Similarly, we introduced `faust2android` in a previous publication [Michon, 2013]: a tool allowing to turn FAUST[8] [Orlarey et al., 2009] code into a fully operational Android application. `faust2android` is based on `faust2api` [Michon et al., 2015]. It allows to turn a FAUST program into a cross-platform API usable on Android and iOS to carry out various kinds of real-time audio processing tasks.

In this paper, we present a completely redesigned version of `faust2api` offering the same features on Android and iOS:

- polyphony and MIDI support,
- audio effects chains,
- built-in sensors support,
- low latency audio,
- etc.

First, we'll give an overview of how `faust2api` works. Then, technical details on the implementation of this system will be provided. Finally, we'll evaluate it and present future directions for this project.

2 Overview

2.1 Basics

At its highest level, `faust2api` is a command line program taking a FAUST code as its main argument and generating a package containing a series of files implementing the DSP engine. Various flags can be used to customize the API. The only required flag is the target platform:

[1]`http://www.apple.com/ios/garageband`. All the URLs in this paper were verified on 01/26/17.
[2]`https://www.smule.com`
[3]`http://www.moforte.com/geoshredapp`
[4]`http://superpowered.com`
[5]`https://www.juce.com`
[6]`https://puredata.info`
[7]`http://www.csounds.com`
[8]`http://faust.grame.fr`

```
faust2api -ios myCode.dsp
```

will generate a DSP engine for iOS and

```
faust2api -android myCode.dsp
```

will generate a DSP engine for Android.

The content of each package is quite different between these two platforms (see §3), but the format of the API itself remains very similar (see Figure 1 at page 4). The iOS DSP engines generated with `faust2api` consist of a large `C++` object (`DspFaust`) accessible through a separate header file. This object can be conveniently instantiated and used in any `C++` or `Objective-C` code in an app project. A typical "life cycle" for a `DspFaust` object can be

```
DspFaust *dspFaust = new DspFaust(SR,
    blockSize); dspFaust->start();
dspFaust->stop(); delete dspFaust;
```

`start()` launches the computation of the audio blocks and `stop()` stops (pauses) the computation. These two methods can be repeated as many times as needed. The constructor allows to specify the sampling rate and the block size, and is used to instantiate the audio engine. While the configuration of the audio engine is very limited at the API level (only these two parameters can be configured through it), lots of flexibility is given to the programmer within the FAUST code. For example, if the FAUST object doesn't have any input, then no audio input will be instantiated in the audio engine, etc.

The value of the different parameters of a FAUST object can be easily modified once the `DspFaust` object is created and is running. For example, the `freq` parameter of the simple FAUST code

```
f = nentry("freq",440,50,1000,0.01);
    process = osc(f);
```

can be modified simply by calling

```
dspFaust->setParamValue("freq",440);
```

FAUST user-interface elements (`nentry` here) are ignored by `faust2api` and simply used as a way to declare parameters controllable in the API. API packages generated by `faust2api` also contain a markdown documentation providing information on how to use the API as well a list of all the parameters controllable with `setParamValue()`.

The structure of the DSP engine package is quite different for Android since it contains both `C++` and `JAVA` files (see §3). Otherwise, the same steps can be used to work with the `DspFaust` object.

2.2 MIDI Support

MIDI support can be easily added to a `DspFaust` object simply by providing the `-midi` flag when calling `faust2api`. MIDI support works the same way on Android and iOS: all MIDI devices connected to the mobile device before the app is launched can control the FAUST object, and any new device connected while the app is running will also be able to control it.

Standard FAUST MIDI meta-data[9] can be used to assign MIDI CCs to specific parameters. For example, the `freq` parameter of the previous code could be controlled by MIDI CC 52 simply by writing

```
f = nentry("freq[midi: ctrl
    52]",440,50,1000,0.01);
```

2.3 Polyphony

FAUST objects can be conveniently turned into polyphonic synthesizers simply by specifying the maximum number of voices of polyphony when calling `faust2api` using the `-nvoices` flag. In practice, only active voices are allocated and computed, so this number is just used as a safeguard.

As used for many years by the various tools for making FAUST synthesizers, such as `faust2pd`, compatibility with the `-nvoices` option requires the `freq`, `gain` and `gate` parameters to be defined. `faust2api` automatically takes care of converting MIDI note numbers to frequency values in Hz for `freq`, MIDI velocity to linear amplitude-gain for `gain`, and note-on (1) and note-off (0) for `gate`:

```
f = nentry("freq",440,50,1000,0.01); g
    = nentry("gain",1,0,1,0.01);
t = button("gate"); process = osc(f)*g*
    t;
```

Here, `t` could be used to trigger an envelope generator, for example. In such a case, the voice would stop being computed only after `t` is set to 0 and the tail-off amplitude becomes smaller than -60dB (configurable using macros in the application code).

A wide range of methods is accessible to work with voices. A "typical" life cycle for a MIDI note can be

```
long voiceAddress = dspFaust->keyOn(
    note,velocity);
dspFaust->setVoiceParamValue("param",
    voiceAddress,paramValue);
```

[9]http://faust.grame.fr/images/faust-quick-reference.pdf

```
dspFaust->keyOff(note);
```

setVoiceParamValue() can be used to change the value of a parameter for a specific voice.

Alternatively, voices can be allocated without specifying a note number and a velocity:

```
long voiceAddress = dspFaust->newVoice
    ();
dspFaust->setVoiceParamValue("param",
    voiceAddress,paramValue);
dspFaust->deleteVoice(voiceAddress);
```

For example, this can be very convenient to associate voices to specific fingers on a touchscreen.

When MIDI support is enabled in faust2api, MIDI events will automatically interact with voices. Thus, if a MIDI keyboard is connected to the mobile device, it will be able to control the FAUST object without additional configuration steps.

2.4 Adding Audio Effects

In most cases, effects don't need to be reimplemented for each voice of polyphony and can be placed at the end of the DSP chain. faust2api allows to provide a FAUST object implementing the effects chain to be connected to the output of the polyphonic synthesizer. This can be done simply by giving the -effect flag followed by a FAUST effects chain file name (e.g., effect.dsp) when calling faust2api:

```
faust2api -android -nvoices 12 -effect
    effect.dsp synth.dsp
```

The parameters of the effect automatically become available in the DspFaust object and can be controlled using the setParamValue() method.

2.5 Working With Sensors

The built-in accelerometer and gyroscope of a mobile device can be easily assigned to any of the parameters of a FAUST object using the acc or gyr meta-data:

```
g = nentry("gain[acc: 0 0 -10 0
    10]",1,0,1,0.01);
```

Complex mappings can be implemented using this system. This feature is not documented here, but more information about it is available in [Michon, 2017]. This reference also provides a series of tutorials on how to use faust2api.

3 Implementation

faust2api takes advantage of the modularity on the FAUST architecture system to generate its custom DSP engines. [Letz et al., 2017] For example, turning a monophonic FAUST synthesizer into a polyphonic one can be done in a simple generic way. Both on Android and iOS, faust2api generates a large C++ file implementing all the features used by the high level API. On iOS, this API is accessed through a C++ header file that can be conveniently included in any C++ or Objective-C code. On Android, a JAVA interface allows to interact with the native (C++) block. The DSP C++ code is the same for all platforms (see Figure 2 at page 5) and is wrapped into an object implementing the polyphonic synthesizer followed by the effects chain (assuming that the -mvoices and -poly2 options were used during compilation).

In this section, we provide more information on the architecture of DSP engines generated by faust2api for Android and iOS.

3.1 iOS

The global architecture of API packages generated by faust2api is relatively simple on iOS since C++ code can be used directly in Objective-C (which is one of the two languages used to make iOS applications along with swift). The FAUST synthesizer object gets automatically connected to the audio engine implemented using CoreAudio. As explained in the previous section, the sampling rate and the buffer length are defined by the programmer when the DspFaust object is created. The number of instantiated inputs and outputs is determined by the FAUST code. By default, the system deactivates gain correction on the input but this can be changed using a macro in the including source code.

MIDI support is implemented using RtMidi [Scavone and Cook, 2005], which is automatically added to the API if the -midi option was used for compilation. Alternatively, programmers might choose to use the propagateMidi() method to send raw MIDI events to the DspFaust object in case they would like to implement their own MIDI receiver.

The same approach can be used for built-in sensors using the propagateAcc() and propagateGyr() methods.

3.2 Android

Android applications are primarily written in JAVA. However, despite the fact that the FAUST compiler can generate JAVA code, it is not a

Basic Elements	**Parameters Control**
`DspFaust`: Constructor	`getParamsCount`: Get number of params
`~DspFaust`: Destructor	`setParamValue`: Set param value
`start`: Start audio processing	`getParamValue`: Get param value
`stop`: Stop audio processing	`getParamAddress`: Get param address
`isRunning`: True if processing is on	`getParamMin`: Get param min value
`getJSONUI`: Get UI JSON description	`getParamMax`: Get param max value
`getJSONMeta`: Get Metadata JSON	`getParamInit`: Get param init value
	`getParamTooltip`: Get param description
Polyphony	
`keyOn`: Start a new note	**Other Functions**
`keyOff`: Stop a note	`propagateMidi`: Propagate raw MIDI messages
`newVoice`: Start a new voice	`propagateAcc`: Propagate raw accel data
`deleteVoice`: Delete a voice	`setAccConverter`: Set accel mapping
`allNotesOff`: Terminate all active voices	`propagateGyr`: Propagate raw gyro data
`setVoiceParamValue`: Set param value for a specific voice	`setGyrConverter`: Set gyro mapping
`getVoiceParamValue`: Get param value for a specific voice	`getCPULoad`: Get CPU load

Figure 1: Overview of the API functions.

good choice for real-time audio signal processing [Michon, 2013]. Thus, DSP packages generated by `faust2api` contain elements implemented both in `JAVA` and `C++`.

The native portion of the package (`C++`) implements the DSP elements as well as the audio engine (see Figure 2) which is based on `OpenSL ES`.[10] The audio engine is configured to have the same behavior as on iOS. Native elements are wrapped into a shared library accessible in `JAVA` through a *Java Native Interface* (JNI) using the *Android Native Development Kit* (NDK).[11]

MIDI receivers can only be created in `JAVA` on Android (and only since Android API 23), thus MIDI support is implemented in the `JAVA` portion. Like on iOS, the `propagateMidi()` method can be used to implement custom MIDI receivers.

While raw sensor data can be retrieved in `C++` on Android, we decided to implement a system similar to the one used for MIDI, where raw sensor data are pushed from the `JAVA` layer to the native one.

[10]`https://www.khronos.org/opensles`

[11]`https://developer.android.com/ndk/index.html`

4 Evaluation

4.1 Use in Other Frameworks

`faust2api` is now used at the core of `faust2android` [Michon, 2013] and `faust2ios`. It is also used as the basis for our new `SmartKeyboard`[12] tool (currently under development), allowing to generate musical applications with advanced user interfaces on Android and iOS. Figure 3 presents *Nuance*, [Michon et al., 2016] a musical instrument based on `faust2api` and `SmartKeyboard`.

4.2 Audio Latency

We measured the "touch-to-sound" and the "round-trip" audio latency of apps based on `faust2api` for various devices using the techniques described by *Google* on their website.[13] The "touch-to-sound" latency is the time it takes to generate a sound after a touch event was registered on the touch screen of the device. The "round-trip" latency is the time it takes to process an analog signal recorded by the built-in microphone or acquired by the line input.

Latency performance hasn't improved on iOS (see Table 1) compared to our previous study [Michon et al., 2015], except for newer devices

[12]`https://ccrma.stanford.edu/~rmichon/smartKeyboard`

[13]`https://source.android.com/devices/audio/latency_measurements.html`

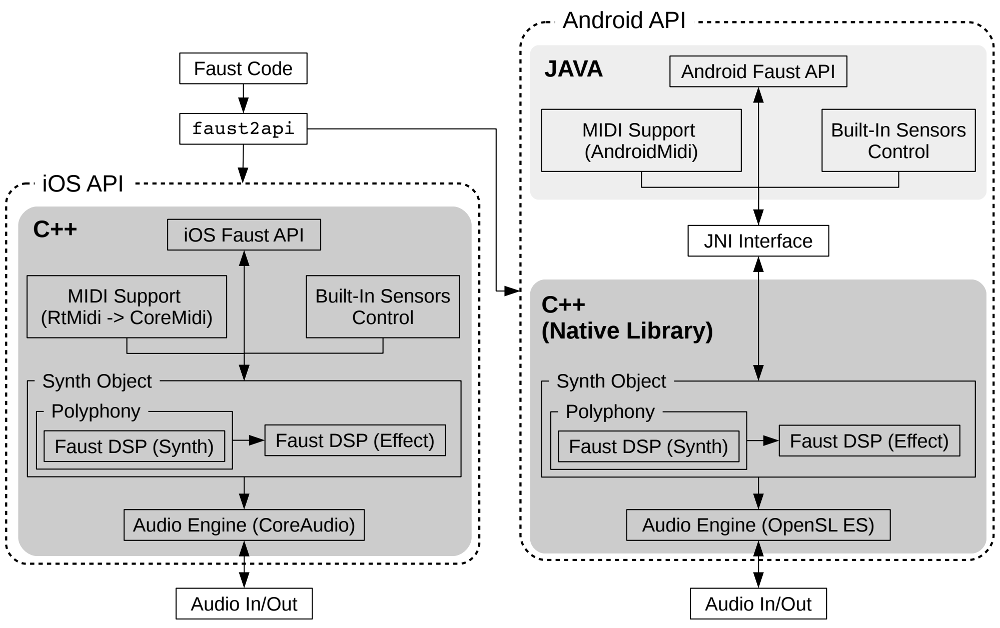

Figure 2: Overview of DSP engines generated with faust2api.

Figure 3: *Nuance*: a musical instrument using faust2api.

such as the *iPad Pro*. On the other hand, Android made huge progress (see Table 2), thanks to tremendous work carried out by *Google*, as well as our completely rewritten audio engine.

Table 2 shows that a “reasonable” latency can only be achieved with the latest version of Android, which confirms the measurements made by Google.[14] Unfortunately, such performances can only be attained on a few devices supported by *Google*, and configured with a specific sampling rate and buffer length.

5 Future Directions

We believe that faust2api has reached a mature and stable state. However, many elements

Device	Touch to Sound	Round Trip
iPhone6	30 ms	13 ms
iPhone5	36 ms	13 ms
iPodTouch	36 ms	13 ms
iPadPro	28 ms	12 ms
iPadAir2	35 ms	13 ms
iPad2	45 ms	15 ms

Table 1: Audio latency for different iOS devices using faust2api.

Device	Touch to Sound	Round Trip	OS
HTC Nexus 9	29 ms	15 ms	7.0
Huawei Nexus 6p	31 ms	17 ms	7.0
Asus Nexus 7	37 ms	48 ms	7.0
Samsung Gal. S5	37 ms	48 ms	5.0

Table 2: Audio latency for different Android devices using faust2api.

can be improved:

First, while basic MIDI support is provided, we haven’t tested it with complex MIDI interfaces such as the one using the Multidimensional Polyphonic Expression (MPE) standard (e.g. LinnStrument,[15] ROLI Seaboard,[16] etc.).

[14] https://source.android.com/devices/audio/latency_measurements.html\#measurements

[15] http://www.rogerlinndesign.com/linnstrument.html

[16] https://roli.com/products/

Currently, specific parameters of the various elements of the API (such as audio engine, MIDI behavior, etc.) can only be configured using source-code macros. We would like to provide a more systematic and in some cases dynamic way of controlling them.

Finally, we plan to add more targets to `faust2api` for various kinds of platforms to help design elements such as audio plug-ins, standalone applications, and embedded systems.

6 Conclusions

Faust gives access to dozens of high quality open source sound processors and generators ranging from specialized types of filters, to virtual analog oscillators, etc. Thanks to `faust2api`, all these elements can be easily embedded and controlled in any Android or iOS app in a very simple manner.

One of the new experimental features of the Faust compiler allows to select at run time the portions of a Faust object that are computed. This makes it possible to create very large objects embedding multiple synthesizers and effects. We believe that this feature, in combination with `faust2api`, will allow to design complex Faust-based DSP engines for a wide range of platforms.

References

Peter Brinkmann, Peter Kirn, Richard Lawler, Chris McCormick, Martin Roth, and Hans-Christoph Steiner. 2011. Embedding PureData with libpd. In *Proceedings of the Pure Data Convention*, Weinmar, Germany.

Georg Essl and Michael Rohs. 2009. Interactivity for mobile music-making. *Organised Sound*, 14(2):197–207.

Lalya Gaye, Lars Erik Holmquist, Frauke Behrendt, and Atau Tanaka. 2006. Mobile music technology: Report on an emerging community. In *Proceedings of the International Conference on New Interfaces for Musical Expression (NIME-06)*, Paris, France, June.

Günter Geiger. 2006. Using the touch screen as a controller for portable computer music instruments. In *Proceedings of the 2006 International Conference on New Interfaces for Musical Expression (NIME-06)*, Paris, France.

`seaboard-grand`

Victor Lazzarini, Steven Yi, Joseph Timoney, Damian Keller, and Marco Pimenta. 2012. The mobile Csound platform. In *Proceedings of the International Conference on Computer Music (ICMC-12)*, Ljubljana, Slovenia, September.

Stéphane Letz, Yann Orlarey, Dominique Fober, and Romain Michon. 2017. Polyphony, sample-accurate control and MIDI support for FAUST DSP using combinable architecture files. In *Proceedings of Linux Audio Conference (LAC-17)*, Saint-Etienne, France.

Romain Michon, Julius Orion Smith, and Yann Orlarey. 2015. MobileFaust: a set of tools to make musical mobile applications with the Faust programming language. In *Proceedings of the Linux Audio Conference (LAC-15)*, Mainz, Germany, April.

Romain Michon, Julius O. Smith, Chris Chafe, Matthew Wright, and Ge Wang. 2016. Nuance: Adding multi-touch force detection to the iPad. In *Proceedings of the Sound and Music Computing Conference (SMC-16)*, Hamburg, Germany.

Romain Michon. 2013. faust2android: a Faust architecture for Android. In *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx-13)*, Maynooth, Ireland, September.

Romain Michon. 2017. Faust tutorials. Webpage. `https://ccrma.stanford.edu/~rmichon/faustTutorials`.

Yann Orlarey, Stéphane Letz, and Dominique Fober, 2009. *New Computational Paradigms for Computer Music*, chapter "Faust : an Efficient Functional Approach to DSP Programming". Delatour, Paris, France.

Gary Scavone and Perry Cook. 2005. RtMidi, RtAudio, and a synthesis toolkit (STK) update. In *Proceedings of the 2005 International Computer Music Conference*, Barcelona, Spain.

Atau Tanaka. 2004. Mobile music making. In *Proceedings of the 2004 conference on New interfaces for musical expression (NIME04)*, National University of Singapore.

Ge Wang. 2014. Ocarina: Designing the iPhone's Magic Flute. *Computer Music Journal*, 38(2):8–21, Summer.