MobileFaust: a Set of Tools to Make Musical Mobile Applications with the Faust Programming Language

Romain Michon, Julius O Iii Smith, Yann Orlarey

MobileFaust: a Set of Tools to Make Musical Mobile Applications with the Faust Programming Language

Romain Michon
CCRMA
Stanford University
CA 94305-8180
USA
rmichon@ccrma.stanford.edu

Julius O. III. Smith
CCRMA
Stanford University
CA 94305-8180
USA
jos@ccrma.stanford.edu

Yann Orlarey
GRAME
Lyon
France
orlarey@grame.fr

Abstract

This work presents a series of tools to turn Faust code into various elements ranging from fully functional applications to multi-platform libraries for real time audio signal processing on iOS and Android. Technical details about their use and function are provided along with audio latency and performance comparisons, and examples of applications.

Keywords

Faust, iOS, Android, DSP

1 Introduction

Mobile platforms offer a great opportunity to the world of open source audio to make sound synthesis and processing accessible to a wider audience [Yi and Lazzarini, 2012; Brinkmann et al., 2011]. The use of smartphones and tablets as musical instruments is now accepted by a large number of musicians. Not only are mobile devices widespread and owned by many, they offer a higher level user interface paradigm than computers, which often makes them more stable and simpler to use. In particular, Android devices, which are more open than iPhones and iPads (§3) offer a good compromise between open-source, stability, and ease of use.

Faust[1] [Orlarey et al., 2002] is a functional programming language for real-time digital signal processing (DSP) that generates highly efficient DSP code in a variety of languages (C, C++, LLVM, asmjs, and more) that can be compiled into a variety of forms using a system of wrappers. These wrappers, called *architecture files*, describe how to adapt the DSP computation to the external world [Fober et al., 2011]. Therefore, it is easy to go from Faust to standalone applications for different kinds of platforms, Web applications, audio plug-ins, externals for music programming languages, and so on.

This paper presents a series of tools that can turn Faust code into various elements ranging from fully functional applications to multi-platform libraries for real-time audio signal processing on iOS and Android. Technical details about their use and function are provided, along with audio latency and performance comparisons, and examples of applications.

2 Faust2api

The main idea of `faust2api` is to provide iOS and Android developers with a system that generates custom high-level APIs for real-time audio signal processing. Even though the APIs work quite differently "under the hood" on iOS than on Android, they are accessed similarly on the two platforms.

The `faust2api` script operates as a command-line tool in a shell. A Faust source file is provided as an argument, along with the option `-ios` or `-android` specifying the desired architecture, and one or more source files are created as output (a single C++ header file for iOS, and a directory containing both Java and C++ source files for Android). The library takes care of starting the audio engine and instantiating the DSP code, as well as connecting them together. At the API level, this is all done by the C++ method `init(sr,bs)` which takes the desired sampling-rate and audio buffer-size as arguments. Computing of the audio process is launched by a `start()` method. Finally, the audio engine can be closed and the memory freed by simply calling `stop()`.[2]

On both iOS and Android, the audio process runs in its own high-priority thread. The various parameters of the Faust object can be accessed and written via `getParam(path)` and `setParam(path)` where the parameter `path` is

[1] `http://faust.grame.fr`

[2] Detailed documentation of the API can be found here: `https://ccrma.stanford.edu/~rmichon/mobileFaust/#ref`

the parameter's path in the user-interface tree defined in the FAUST code (as discussed further below in §3.3 on OSC and MIDI support).

If the FAUST object provided to `faust2api` has no inputs, and has `freq`, `gain`, and `gate` parameters defined, it is considered as an instrument and automatically made polyphonic. The different voices (eight by default, but this can be changed) can be triggered using a `keyOn()` method that takes a MIDI note number and MIDI velocity as an argument. This method is linked to the `freq`, `gain`, and `gate` parameters (§3.1) and allocates a new voice every time it is called. The `keyOff()` method sets the `gate` parameter of the voice to zero and waits until the level of the voice falls below -60 dB to deallocate it.

2.1 iOS

The command "`faust2api -ios faustCode.dsp`" will generate a single C++ header file that can be included in any iOS app project. The API relies on the `AVAudioSession`[3] framework to connect to the audio engine.

"Touch to sound" and "round-trip" latency measurements for iOS audio applications generated by `faust2api` were carried out on an iPad2 and an iPhone5 (Fig. 1).

Device	Touch to Sound	Round-Trip
iPhone5	36 ms	13 ms
iPad2	45 ms	15 ms

Figure 1: Audio Latency for Different iOS Devices Using the FAUST Library.

"Round-trip" latency was measured by creating a simple app that just plays back any sound that comes to its audio input (in our case, the audio input jack) and by comparing how long it takes for the iOS device to play back an impulse sent to this app.

"Touch to sound" latency was measured by creating a simple test app where a button on the screen is used to trigger an impulse. The audio output jack of the iOS device was connected to an ADC[4] in order to be able to record the impulse on a computer. A microphone connected to the same ADC on a different channel was placed at the top of the screen of the device. The latency measurements were carried out by measuring the time difference between the "acoustic" impulse detected by the microphone and the synthesized one (Fig. 2).

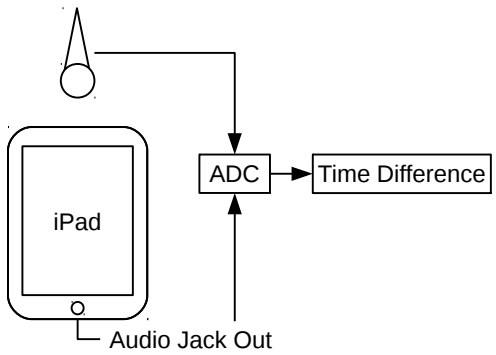

Figure 2: Touch to Sound Audio Latency Measurement Set-Up.

2.2 Android

`faust2api` is slightly more complex to use on Android than iOS. Indeed, Android apps are primarily programmed in Java. However, this language is not very well suited for real time DSP so most of the library generated by `faust2api` is written in C++ with a Java interface.

The audio engine is accessed, controlled and connected to the DSP code generated by FAUST on the "native" side of the library where everything is computed in a high-priority thread. This allows the signal processing part of the app to be fully independent from the Java side.

The native portion of the library is compiled as a shared library using the Android NDK[5] and can be controlled in Java using a JNI[6] interface generated by SWIG.[7] More details about the way this system works can be found in [Michon, 2013].

In practice, `faust2api` will generate the Android API by using the `-android` option instead of `-ios` (cf. previous section) and will provide a set of Java and C++ files to be copied in the Android app project.[8]

[3] `https://developer.apple.com/library/ios/-documentation/AVFoundation/Reference/-AVAudioSession_ClassReference/index.html#//-apple_ref/occ/cl/AVAudioSession`

[4] *Analog to Digital Converter*

[5] *Native Development Toolkit*: `https://developer.android.com/tools/sdk/ndk/`

[6] *Java Native Interface*

[7] *Simplified Wrapper and Interface Generator*: `http://www.swig.org/`

[8] A tutorial on how to do this can be found here: `https://ccrma.stanford.edu/~rmichon/mobileFaust/#f2apAndroid`

Latency measurements using the same techniques presented in the previous section were carried out on a Samsung Galaxy S5, a Google Nexus 5, and a Google Nexus 7 that were all running on Android 5.0 (Lollipop). It is difficult to make a complete comparison here in the same way as for iOS because latency varies greatly between devices and manufacturers. The main observation that can be made though is that audio latency is much larger on Android than iOS.

Device	Touch to Sound	Round-Trip
Samsung Galaxy S5	72 ms	78 ms
Google Nexus 5	90 ms	92 ms
Google Nexus 7	130 ms	130 ms

Figure 3: Audio Latency of Different Android Devices Using the FAUST Library.

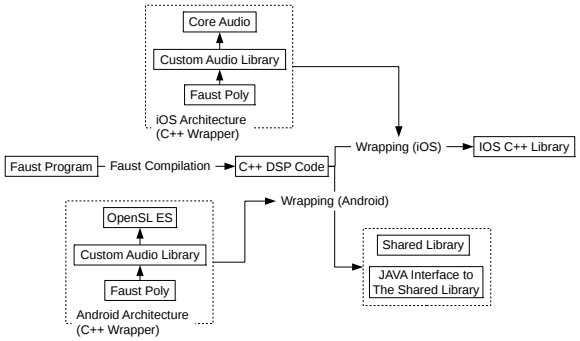

Figure 4: `faust2api` Overview.

3 Faust2android

While a preliminary version of `faust2android` was presented in [Michon, 2013] it has been totally rewritten since then and offers a large number of new functionalities.

`faust2android` is built on top of `faust2api` (Fig. 9). Its user interface is constructed using the JSON description provided by the shared library generated by `faust2api`. All the standard FAUST UI elements are available: horizontal and vertical groups, horizontal and vertical sliders, numerical entries, knobs, checkboxes, buttons, drop-down menus, radio buttons, bargraphs, etc. Some examples are shown in figure 5. The values of the parameters of the audio process running on the native side are changed using the `setParam()` function of the API.

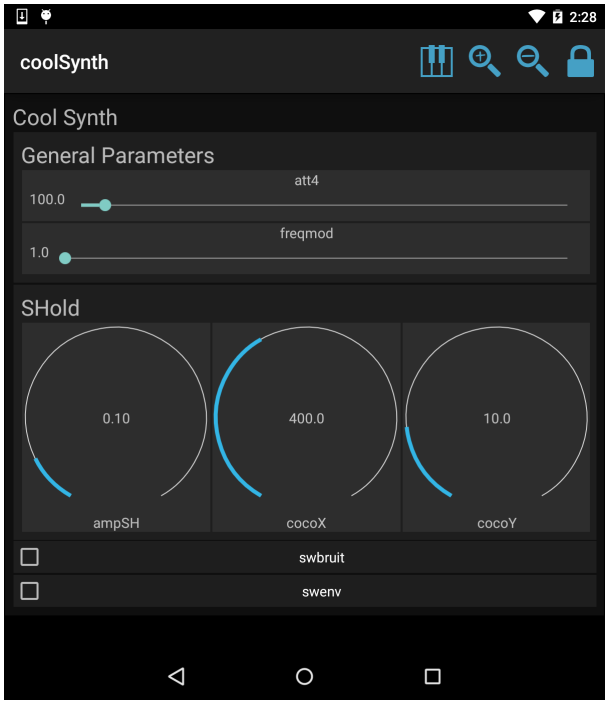

Figure 5: Example of interface generated by `faust2android` containing groups, sliders, knobs and checkboxes.

3.1 Keyboard and Multitouch Interface

`faust2android` allows assignment of more interactive interfaces to the FAUST process. For that, three different metadata items can be added to the top-level group of a FAUST program. In FAUST, a metadata item consists of a key:value pair, specified between square brackets within a title string, i.e., `"Some Title [key:value]..."`.

The `[style:keyboard]` metadata item specifies that the `freq`, `gain`, and `gate` parameters in the FAUST code should be assigned to a piano keyboard that can be opened by touching the "keyboard icon" in the top right corner of the app. Also, these three parameters will be automatically removed from the main interface for controlling the other parameters.

The following example program illustrates a simple usage:

```
import("music.lib");
s = button("gate");
g = hslider("gain",0.1,0,1,0.001);
f = hslider("freq",100,20,10000,1);
process = vgroup("[style:keyboard]",
    s*g*osc(f));
```

This interface uses the polyphonic capabilities of `faust2api`. Touching a key on the key-

board determines the reference pitch of the note but sliding the finger across the X axis of the screen allows the user to continuously control it. The Y axis determines the gain of the note. If a MIDI keyboard is plugged into the Android device, it will be able to control the keyboard interface (§3.3).

The `[style:multi]` metadata item will create a simple interface in which parameters are represented by moveable dots on the screen. Each dot can have two parameters assigned to it, corresponding to x and y screen coordinates. This interface can also be opened by touching the keyboard icon on the top right corner of the screen. Parameters are linked to the interface via `[multi:x]` metadata where `x` is the ID of the parameter in the interface. For example, the FAUST program

```
import("music.lib");
freq = hslider("freq[multi:0]",
    440,50,2000,0.1);
process = hgroup("[style:multi]",
    osc(freq));
```

creates an app in which the frequency parameter of a sine oscillator is controlled by the X axis of the dot in the multitouch interface. Parameters that have the accelerometer assigned to them (cf. §3.2) will continue to be driven by the accelerometer in the interface.

Finally, the `[style:multikeyboard]` metadata combines the keyboard and multitouch interface into one (Fig. 6).

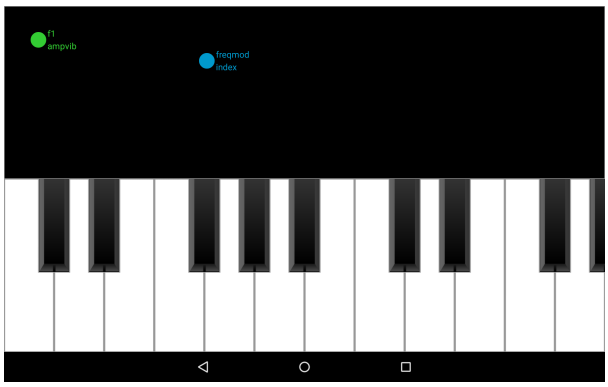

Figure 6: Example of "Multi-Keyboard" Interface of an Application Generated by `faust2android`.

3.2 Using the Built-In Accelerometer

The Accelerometer can be used to control some elements of the user interface. Assignments are made in the *Accelerometer Parameters* panel that can be opened by holding the label of a parameter for more than one second (Fig. 7). From here, the mapping of an accelerometer to a parameter can be configured precisely to create complex linear and non-linear behaviors. For instance, the user can choose which axis will control the parameter (x, y, or z), its motion orientation, and sensitivity.

Three different modes can be used to control the orientation of the accelerometer, *normal*, *inverted*, and *bell*. In *bell* mode, the maximum value of the accelerometer is output when it is in center position and the minimum value when it is fully inclined to the left or right.

Sensitivity can be configured with three different parameters, *min*, *max*, and *center*, that are all expressed in $m/s^2 \times 10^{-1}$. As an example, setting *min* to -1, *max* to 1, and *center* to 0 will create a linear behavior where the minimum value of the parameter being controlled is given at position -90 degrees and the maximum value at position +90 degrees. Any acceleration beyond these limits will be clipped.

All these parameters can be configured from the FAUST code using metadata by specifying `[acc: a b c d e]`, where `a` is the axis (0 for x, 1 for y, 2 for z), `b` the orientation (0 for *normal*, 1 for *inverted*, 2 for *bell*), `c` the minimum, `d` the maximum and `e` the center.

Raw data from the accelerometers are passed directly to the FAUST audio process. Filtering can be carried out in FAUST which is better suited for that kind of task than Java.

Finally, the accelerometer parameters window is only accessible if the app is unlocked by touching the "lock" icon on the top right corner of the screen (Fig. 5). Apps can be locked to prevent users from opening a configuration window or rotating the screen during a performance.

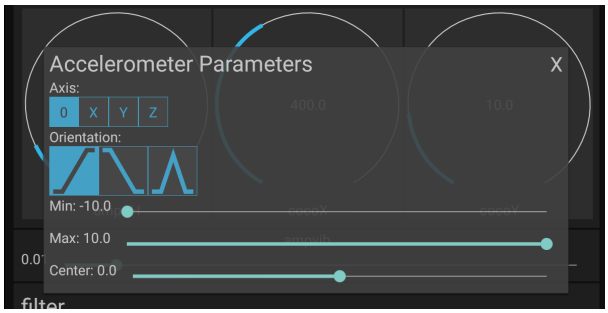

Figure 7: Accelerometer Configuration Panel of an Application Generated by `faust2android`.

3.3 OSC and MIDI Support

OSC support is enabled by default for all the parameters of applications generated by `faust2android`. The OSC address of a parameter corresponds to the path to this parameter in the FAUST code. For example, the OSC address of the `freq` parameter of the FAUST code

```
freq = hslider("freq",
    440,50,2000,0.1);
process = hgroup("Main",osc(freq));
```

will be `/Main/freq`.

MIDI support is also enabled by default in apps generated by `faust2android`. MIDI Key Number is automatically mapped to the `freq` parameter by converting it to frequency in Hz, and similarly MIDI velocity → `gain`. Note on/off events control the `gate` parameter, just like the `keyOn()` and `keyOff()` functions of `faust2api`. Synthesizer apps generated with `faust2android` all have eight voices of polyphony.

MIDI control numbers can be assigned to specific parameters from the FAUST code using the `[midictl:x]` metadata where `x` is the MIDI control number.

3.4 Audio IO Configuration

Android applications generated with `faust2android` automatically choose the best sampling rate and buffer size as a function of the device that is running them (for *Nexus*[9] devices only). Indeed, it was explained during the *Google I/O 2013 conference on High Performance Audio*[10] that Android phones and tablets achieve better audio latency performance if they run with a specific buffer size and sampling rate (see Fig. 8). Users may override these default values in the settings menu of the app.

Device	Sampling Rate	Buffer Size
Nexus S	44100	880
Galaxy Nexus	44100	144
Nexus 4	44100	240
Nexus 7	44100	512
Nexus 10	44100	256
Others	44100	512

Figure 8: Preferred Buffer Sizes and Sampling Rates for Various Android Devices.

[9] `http://www.google.com/nexus/`

[10] `http://youtu.be/d3kfEeMZ65c`

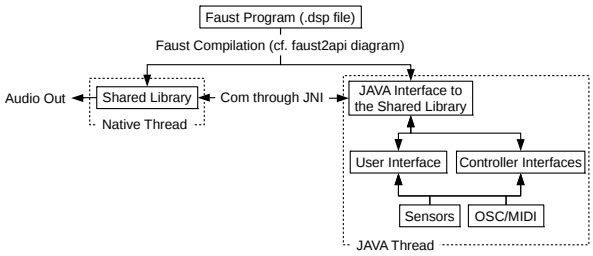

Figure 9: `faust2android` Overview.

3.5 Easy App Generation

While it is relatively simple to use `faust2android`, it requires the programmer to have an important number of dependencies installed (Android SDK and NDK, etc.). However, FAUSTLIVE [Denoux et al., 2014] and the FAUST Online Compiler [Michon and Orlarey, 2012] make the process of turning FAUST code into an Android application very simple. Indeed, when the user chooses to compile a FAUST program as an Android app, a QR code pointing to the generated app package is displayed that can be scanned by the device where the user want the app to be installed.

4 Applications

The FAUST distribution contains a collection of libraries that implement a large number of common and less-common audio effects, filters, and synthesizers. With `faust2api`, iOS and Android programmers who don't know signal processing or who never worked with real-time audio can easily integrate any of the pre-written FAUST modules into their project without having to write a single line of DSP code. On the other hand, this tool also gives the opportunity to FAUST developers to have their work used by more people. A concrete use of this tool was made this year in the *Music 256b* class[11] *"Mobile Music"* offered at Stanford University's Center for Computer Research in Music and Acoustics (CCRMA)[12] where students were given the opportunity to use `faust2api` in their final projects.

Another use of applications generated by `faust2android` and `faust2ios` is the *SmartFaust*[13] project led by Xavier Garcia and Christophe Lebreton at GRAME. The idea was

[11] `https://ccrma.stanford.edu/courses/256b-winter-2015/`

[12] `http://ccrma.stanford.edu`

[13] `http://www.grame.fr/anything_slides/concert-smartfaust`

to make a series of concerts where the music is made by the audience with their mobile phones. Several applications were put on the *Apple Store* and the *Google Play Store* that people could download prior to the concert. This project led to more metadata for controlling the user interfaces; for example, it is possible to choose to not integrate a UI element to the interface. This enables the FAUST programmer to control some specific parameters with the accelerometer (using metadata too) without displaying them in the interface. `faust2android` can also generate "concert apps", where the user can switch between different FAUST objects within the same application.

5 Conclusions

Several tools that use FAUST to help design or make ready-to-use Android and iOS applications were presented. We believe that they make the development of musical applications on mobile platforms easier and that they will contribute to making the use of FAUST objects more accessible to musicians and performers.

While iOS real-time audio applications provide much better (smaller) audio latency than Android, the various restrictions imposed by Apple on their deployment makes them less accessible which is a big issue for the use that we make of them with FAUST. Therefore, we hope that Google will resolve the audio latency issues for Android applications in the near future.

FAUSTLIVE and the Online Compiler provide easy ways to use the tools presented in this paper. However, we think that enhancing them with an online platform where FAUST developers can easily share their work with others in order to create a repository of FAUST resources would be a great addition.

6 Acknowledgments

Part of this work has been implemented under the FEEVER project [ANR-13-BS02-0008] supported by the Agence Nationale pour la Recherche.

References

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

Sarah Denoux, Stephane Letz, Yann Orlarey, and Dominique Fober. 2014. FaustLive: just-in-time Faust compiler and much more. In *Proceedings of the Linux Audio Conference (LAC-14)*, pages 102–107. ZKM, Karlsruhe (Germany), May.

Dominique Fober, Yann Orlarey, and Stéphane Letz. 2011. Faust architecture design and OSC support. In *Proceedings of the Conference on Digital Audio Effects (DAFx-11)*, pages 213–216, IRCAM, Paris, France.

Romain Michon and Yann Orlarey. 2012. The Faust online compiler: a web-based IDE for the Faust programming language. In *Proceedings of the Linux Audio Conference (LAC-12)*, pages 111–116. CCRMA, Stanford University (USA).

Romain Michon. 2013. Faust2android: a Faust architecture for Android. In *Proceedings of the 16th International Conference on Digital Audio Effects (DAFx-2013)*, pages 98–102. National University of Ireland (Maynooth), September.

Yann Orlarey, Dominique Fober, and Stephane Letz. 2002. An algebra for block diagram languages. In *Proceedings of the International Computer Music Conference (ICMA)*, pages 542–547. Gothenburg, Sweden.

Steven Yi and Victor Lazzarini. 2012. Csound for Android. In *Proceedings of the Linux Audio Conference (LAC-12)*, pages 233–239. CCRMA, Stanford University (USA).