FAUST2SMARTKEYB: A TOOL TO MAKE MOBILE INSTRUMENTS FOCUSING ON SKILLS TRANSFER IN THE FAUST PROGRAMMING LANGUAGE

Romain Michon[1,2]*, Julius O. Smith*[1]*, Chris Chafe*[1]*, Ge Wang*[1]*, and Matthew Wright*[1]

[1]Center for Computer Research in Music and Acoustics (CCRMA), Stanford University, USA
[2]GRAME – Centre National de Création Musicale, Lyon, France
rmichon@ccrma.stanford.edu

ABSTRACT

In this paper, we present `faust2smartkeyb`, a tool to create musical apps for Android and iOS using the FAUST programming language. The use of musical instrument physical models in this context through the FAUST Physical Modeling Library is emphasized. We also demonstrate how this system allows for the design of interfaces facilitating skills transfer from existing musical instruments.

1. INTRODUCTION

Making musical apps for mobile devices involves the use and mastery of various technologies, standards, programming languages, and techniques ranging from low level C++ programming for real-time DSP (Digital Signal Processing) to advanced interface design. This adds up to the variety of the platforms (e.g., iOS, Android, etc.) and of their associated tools (e.g., Xcode, Android Studio, etc.), standards, and languages (e.g., JAVA, C++, Objective-C, etc.).

While there exists a few tools to facilitate the design of musical apps such as libpd [1], Mobile CSOUND [2], and more recently JUCE[1] and SuperPowered,[2] none of them provides a comprehensive cross-platform environment for musical touchscreen interface design, high level DSP programming, turnkey instrument physical model prototyping, built-in sensors handling and mapping, MIDI and OSC compatibility, etc.

`faust2ios` and `faust2android` [3] partially addressed these issues. They are command line tools to convert FAUST [4] code into fully working Android and iOS applications. The user interface of apps generated using these systems corresponds to the standard UI specifications provided in the FAUST code and is made out of sliders, buttons, groups, etc. More recently, `faust2api`, a lower level tool to generate audio engines with FAUST featuring polyphony, built-in sensors mapping, MIDI and OSC (Open Sound Control) support, etc., for a wide range of platforms including Android and iOS was introduced [5].

Despite the fact that user interfaces better adapted to musical applications (e.g., piano keyboards, (x, y) controllers, etc.) can replace the standard UI of a FAUST object in apps generated by `faust2android` [6], they are far from providing a generic solution to capture musical gestures on a touchscreen and to allow for musical skill transfer.

In this paper, we introduce `faust2smartkeyb`,[3] a tool based on `faust2api` to generate Android and iOS apps using FAUST. `faust2smartkeyb` allows for the design of an extended number of musical interfaces and behaviors directly from a FAUST code. First, we describe the implementation of the system. Next, we demonstrate how to use it to implement a wide range of behaviors and mappings. Finally, we present a series of examples where physical models from the FAUST Physical Modeling Library [7] are turned into standalone instruments using `faust2smartkeyb` and implement various types of instrumental skills.

2. FAUST2SMARTKEYB

2.1. Apps Generation and General Implementation

`faust2smartkeyb` works the same way as most FAUST targets/"architectures" [8] and can be called using the `faust2smartkeyb` command-line tool:

```
faust2smartkeyb [options] faustFile.dsp
```

where `faustFile.dsp` is a FAUST file declaring a SMARTKEYBOARD interface (see §2.2) and `[options]` is a set of options allowing us to configure general parameters of the generated app (see Table 1).

Option	Description
`-android`	Generate an Android app
`-ios`	Generate an iOS app
`-effect`	Specify a FAUST effect file
`-install`	Install the app on the device (Android only)
`-nvoices`	Specify the number of polyphony voices of the DSP engine
`-reuse`	Reuse an existing app project (only update what was changed)
`-source`	Generate the source code of the app

Table 1: Selected `faust2smartkeyb` options.

The only required option is the app type (`-android` or `-ios`). Unless specified otherwise (e.g., using the `-source` option), `faust2smartkeyb` will compile the app directly in the terminal and upload it on any Android device connected to the computer if the `-install` option is provided. If `-source` is used, an Xcode[4] or an Android Studio[5] project is generated, depending on the selected app type.

[1]`https://www.juce.com` All the URLs presented in this paper were verified on March 12, 2018.

[2]`http://superpowered.com`

[3]`faust2smartkeyb` is now part of the FAUST distribution. Additional information and documentation about this tool can be found on this webpage: `https://ccrma.stanford.edu/~rmichon/smartKeyboard/`.

[4]`https://developer.apple.com/xcode/`

[5]`https://developer.android.com/studio/`

`faust2smartkeyb` is based on `faust2api` [5] and takes advantage of most of the features of this system. It provides polyphony, MIDI, and OSC support and allows SMARTKEYBOARD interfaces to interact with the DSP portion of the app at a very high level (see Figure 1).

`faust2smartkeyb` inherits some of `faust2api`'s options. For example, an external audio effect FAUST file can be specified using `-effect`. This is very useful to save computation when implementing a polyphonic synthesizer. Similarly, `-nvoices` can be used to override the default maximum number of polyphony voices (twelve) of the DSP engine generated by `faust2api`.

The DSP engine generated by `faust2api` is transferred to a template Xcode or Android Studio project (see Figure 1) and contains the SMARTKEYBOARD declaration (see §2.2). The interface of the app, which is implemented in JAVA on Android and in Objective-C on iOS, is built from this declaration. While OSC support is built-in in the DSP engine and works both on iOS and Android, MIDI support is only available on iOS thanks to Rt-MIDI. On Android, raw MIDI messages are retrieved in the JAVA portion of the app and "pushed" to the DSP engine. MIDI is only supported since Android-23 so `faust2smartkeyb` apps wont have MIDI support on older Android versions.

2.2. Architecture of a Simple `faust2smartkeyb` Code

The SMARTKEYBOARD interface can be declared anywhere in a FAUST file using the `SmartKeyboard{}` metadata:

```
declare interface "SmartKeyboard{
  // configuration keys
}";
```

It is based on the idea that a wide range of touchscreen musical interface can be implemented as a set of keyboards with different key numbers (like a table with columns and cells, essentially). Various interfaces ranging from drum pads, isomorphic keyboards, (x, y) controllers, wind instruments fingerings, etc. can be implemented using this paradigm. The position of fingers in the interface can be continuously tracked and transmitted to the DSP engine both as high level parameters formatted by the system (e.g., frequency, note on/off, gain, etc.) or low level parameters (e.g., (x, y) position, key and keyboard ID, etc.). These parameters are declared in the FAUST code using default parameter names (see Table 2 for a summary).

By default, the screen interface is a polyphonic chromatic keyboard with thirteen keys whose lowest key is a C5 (MIDI note number 60). A set of key/value pairs can be used to override the default look and behavior of the interface (see Table 3). Code Listing 1 presents the FAUST code of a simple app where two identical keyboards can be used to control a simple synthesizer based on a band-limited sawtooth wave oscillator and a simple exponential envelope generator. Since MIDI support is enabled by default in apps generated by `faust2smartkeyb` and that the SMARTKEYBOARD standard parameters are the same as the one used for MIDI in FAUST, this app is also controllable by any MIDI keyboard connected to the device running it. A screen-shot of the interface of the app generated from Code Listing 1 can be seen in Figure 2.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'2'
}";
```

Parameter Name	Description
`freq`	Base frequency (if any) of the current note
`bend`	Deviation from `freq` as a ratio (1 = no deviation) for continuous pitch control
`gate`	Note on (1) / Note off (0), typically changes with `freq`
`key`	Current key ID
`keyboard`	Current keyboard ID
`kbMfingers`	Number of fingers on a specific keyboard `M`
`kbMkNstatus`	Status of the current key `N` in keyboard `M`
`kbMkNx`	Normalized (0-1) x position of a finger in key `N` in keyboard `M`
`kbMkNy`	Normalized (0-1) y position of a finger in key `N` in keyboard `M`
`x`	Normalized (0-1) x position of the finger in any key
`y`	Normalized (0-1) y position of the finger in any key
`xN`	Normalized (0-1) x position of finger `N` in a key
`yN`	Normalized (0-1) y position of finger `N` in a key

Table 2: SMARTKEYBOARD standard parameters overview.

```
import("stdfaust.lib");
f = nentry("freq",200,40,2000,0.01);
g = nentry("gain",1,0,1,0.01);
t = button("gate");
envelope = t*g : si.smoo;
process = os.sawtooth(f)*envelope <: _,_;
```

Listing 1: Simple SMARTKEYBOARD FAUST app.

2.3. Preparing a FAUST Code for Continuous Pitch Control

In `faust2smartkeyb` programs, pitch is handled using the `freq` and `bend` standard parameters (see Table 2). The behavior of the formatting of these parameters can be configured using some of the keys presented in Table 3.

`freq` gives the "reference frequency" of a note and is tied to the `gate` parameter. Every time `gate` goes from 0 to 1 (which correlates with a new note event), the value of `freq` is updated. `freq` always corresponds to an integer MIDI pitch number which implies that its value is always quantized to the nearest semitone.

Pitch can be continuously updated by using the `bend` standard parameter. `bend` is a ratio that should be multiplied to `freq`. E.g.:

```
f = nentry("freq",200,40,2000,0.01);
bend = nentry("bend",1,0,10,0.01) : si.
    polySmooth(t,0.999,1);
freq = f*bend;
```

The state of polyphonic voices is conserved in memory until the app is ended. Thus, the value of `bend` might jump from one value to another when a new voice is activated. `polySmooth()` is used here to smooth the value of `bend` to prevent clicks, only after the

Key	**Description**
`Inter-Keyboard Slide`	Enables slide between keyboards
`Keyboard N - Key M - Label`	Specify text in a specific key and keyboard
`Keyboard N - Lowest Key`	MIDI key number of the lowest key on a specific keyboard
`Keyboard N - Number of Keys`	Number of keys of a specific keyboard
`Keyboard N - Orientation`	Orientation (left to right or right to left) of a specific keyboard
`Keyboard N - Piano Keyboard`	Activate piano keyboard mode (black keys) on a specific keyboard
`Keyboard N - Root Position`	Position of the root on a specific keyboard
`Keyboard N - Scale`	Specify the scale of a specific keyboard
`Keyboard N - Send Freq`	Send `freq` and `bend` from a specific keyboard
`Keyboard N - Send Key X`	Activates the `kbMkNx` standard parameter
`Keyboard N - Send Key Y`	Activates the `kbMkNy` standard parameter
`Keyboard N - Send Key Status`	Activates the `kbMkNstatus` standard parameter
`Keyboard N - Send Numbered X`	Activates the `xN` standard parameter
`Keyboard N - Send Numbered Y`	Activates the `xY` standard parameter
`Keyboard N - Send X`	Activates the `x` standard parameter
`Keyboard N - Send Y`	Activates the `y` standard parameter
`Keyboard N - Show Labels`	Show key labels on a specific keyboard
`Keyboard N - Static Mode`	Fix key appearance on a specific keyboard
`Number of Keyboards`	Number of keyboards in the interface
`Max Fingers`	Maximum number of fingers allowed in the interface
`Max Keyboard Polyphony`	Maximum keyboards polyphony voices
`Mono Mode`	Mode when keyboards are monophonic
`Rounding Cycles`	Number of cycles of pitch rounding
`Rounding Mode`	Pitch rounding mode
`Rounding Smooth`	Smoothness of pitch rounding
`Rounding Threshold`	Pitch rounding threshold
`Rounding Update Speed`	Pitch rounding update speed
`Send Current Key`	Activates the `key` standard parameter
`Send Current Keyboard`	Activates the `keyboard` standard parameter
`Send Fingers Count`	Activates the `kbMfingers` standard parameter
`Send Sensors`	Send sensor values

Table 3: `faust2smartkeyb` keys overview.

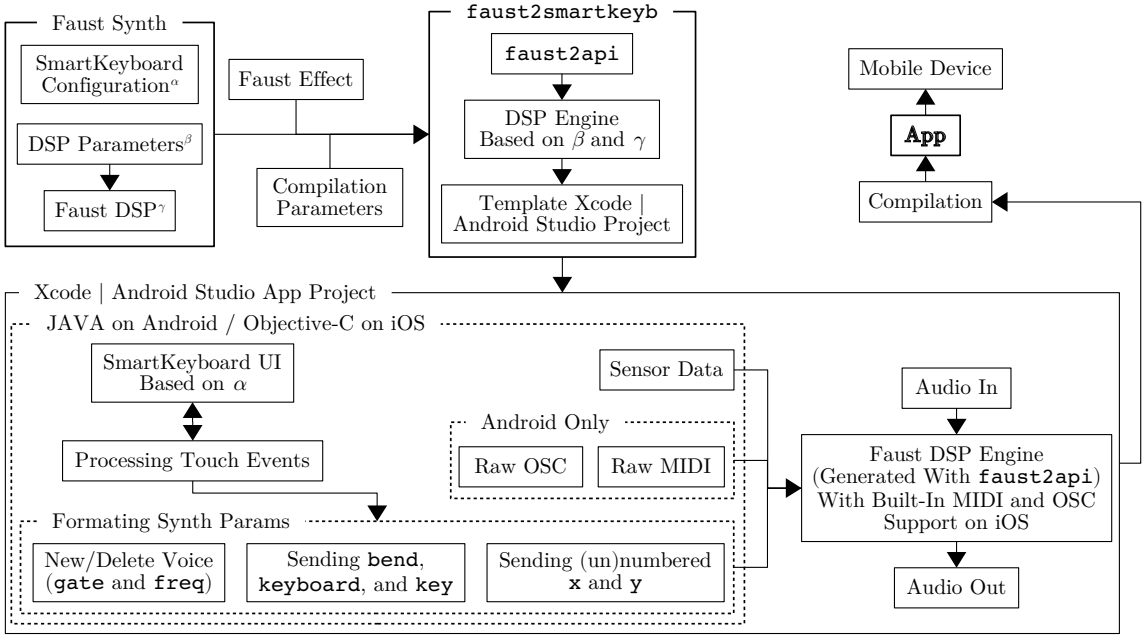

Figure 1: Overview of `faust2smartkeyb`.

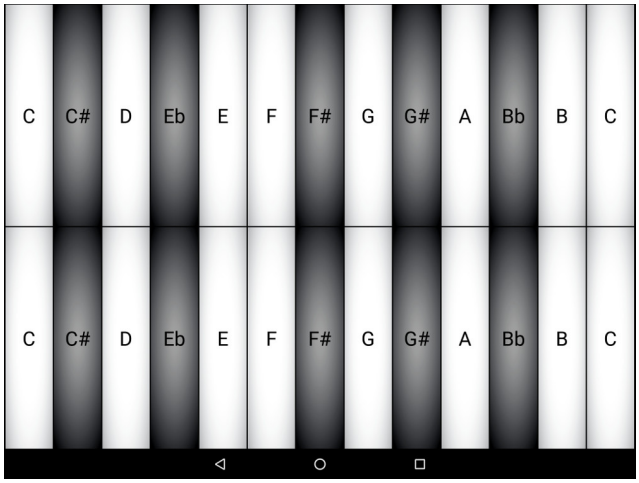

Figure 2: Simple SMARTKEYBOARD interface.

voice started. This suppresses any potential "sweep" that might occur if the value of `bend` changes abruptly at the beginning of a note.

2.4. Configuring Continuous Pitch Control

The `Rounding Mode` configuration key has a significant impact on the behavior of `freq`, `bend`, and `gate`.

When `Rounding Mode = 0`, pitch is fully "quantized," and the value of `bend` is always 1. Additionally, a new note is triggered every time a finger slides to a new key, impacting the value of `freq` and `gate`.

When `Rounding Mode = 1`, continuous pitch control is activated, and the value of `bend` is constantly updated in function the position of the finger on the screen. New note events updating the value of `freq` and `gate` are only triggered when fingers start touching the screen. While this mode might be useful in some cases, it is hard to use when playing tonal music as any new note might be "out of tune."

When `Rounding Mode = 2`, "pitch rounding" is activated and the value of `bend` is rounded to match the nearest quantized semitone when the finger is not moving on the screen. This allows generated sounds to be "in tune" without preventing slides, vibratos, etc. While the design of such a system has been previously studied, [9] we decided to implement our own algorithm for this (see Figure 3). `touchDiff` is the distance on the screen between two touch events for a specific finger. This value is smoothed (`sTouchDiff`) using a unity-dc-gain one pole lowpass filter in a separate thread running at a rate defined by configuration key `Rounding Update Speed`. `Rounding Smooth` corresponds to the pole of the lowpass filter used for smoothing (0.9 by default). A separate thread is needed since the callback of touch

events is only called when events are received. If `sTouchDiff` is greater than `Rounding Threshold` during a certain number of cycles defined by `Rounding Cycles`, then rounding is deactivated and the value of `bend` corresponds to the exact position of the finger on the screen. If rounding is activated, the value of `bend` is rounded to match the nearest pitch of the chromatic scale.

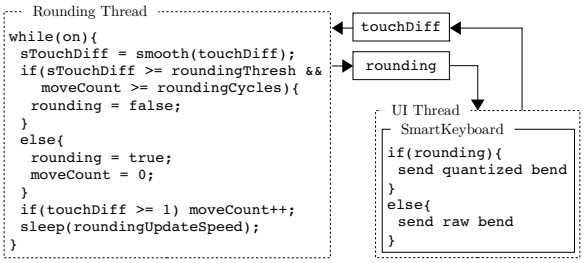

Figure 3: SMARTKEYBOARD pitch rounding *pseudo code* algorithm.

2.5. Using Specific Scales

A wide range of musical scales (see Table 4), all compatible with the system described in §2.4, can be used with the SMARTKEYBOARD interface and configured using the `Keyboard N - Scale` key (see Table 3). When other scales than the chromatic scale are used, keys on the keyboard all have the same color.

Scale ID	Scale Name
0	Chromatic
1	Major
2	Minor
3	Harmonic Minor
4	Dorian
5	South-East Asian
6	Minor Pentatonic
7	Minor Blues
8	Japanese
9	Major Pentatonic
10	Major Blues
11	Mixolydian
12	Klezmer

Table 4: SMARTKEYBOARD scales configurable with the `Keyboard N - Scale` key.

Custom scales and temperaments can be implemented using the `Keyboard N - Scale` configuration key. It allows us to specify a series of intervals to be repeated along the keyboard (not necessarily at the octave). Intervals are provided as semitones and can have a decimal value. For example, the chromatic scale can be implemented as:

```
Keyboard N - Scale = {1}
```

Similarly, the standard equal-tempered major scale can be specified as:

```
Keyboard N - Scale = {2,2,1,2,2,2,1}
```

A 5-limit just intoned major scale (rounded to the nearest 0.01 cents) could be:

```
Keyboard N - Scale =
    {2.0391,1.8243,1.1173,2.0391,2.0391,
  1.8243,1.1173}
```

Equal-tempered Bohlen-Pierce (dividing 3:1 into 13 equal intervals) would be:

```
Keyboard N - Scale = {146.304230835802}
```

Alternatively, custom scales and pitch mappings can be implemented directly from the FAUST code using some of the lower level standard parameters returned by the SMARTKEYBOARD interface (e.g., `x`, `y`, `key`, `keyboard`, etc.).

2.6. Handling Polyphony and Monophony

By default, the DSP engine generated by `faust2api` has twelve polyphony voices. This parameter can be overridden using the `-nvoices` option when executing the `faust2smartkeyb` command. This system works independently from the monophonic/polyphonic configuration of the SMARTKEYBOARD interface. Indeed, even when a keyboard is monophonic, a polyphonic synthesizer might still be needed to leave time for the release of an envelope generator, for example.

The `Max Keyboard Polyphony` key defines the maximum number of voices of polyphony of a SMARTKEYBOARD interface. Polyphony is tied to fingers present on the screen, in other words, one finger corresponds to one voice. If `Max Keyboard Polyphony = 1`, then the interface becomes "monophonic." The monophonic behavior of the system is configured using the `Mono Mode` key (see Table 5). Each mode might be useful for a specific context. For example, *Mode 3* might be great to use keyboards in the interface as independent guitar strings, etc. More examples of this type of use are provided in §3.

Mono Mode	Description
0	Focus stays on the same finger even if other fingers touch the interface.
1	Focus always goes to the latest finger to touch the interface (voice stealing). When the focused finger leaves the interface, focus is transfered to the closest finger.
2	Sames as 1, but the voice is terminated when the focused finger leaves the interface.
3	Sames as 1, but focus is given to new fingers only if their pitch is higher than the current note.
4	Sames as 2, but focus is given to new fingers only if their pitch is lower than the current note.

Table 5: Different monophonic modes configured using the `Mono Mode` key in SMARTKEYBOARD interfaces.

2.7. Other Modes

In some cases, both the monophonic and the polyphonic paradigms are not adapted. For example, when implementing an instrument based on a physical model, it might be necessary to use a single

voice and constantly run it. This might be the case of a virtual wind instrument where notes are "triggered" by some of the continuous parameters of the embouchure (see §3.4) and not by discrete events such as the one created by a key. This type of system can be implemented by setting the `Max Keyboard Polyphony` key to zero. In that case, the first available voice is triggered and ran until the app is killed. Adding new fingers on the screen will have no impact on that and the `gate` parameter wont be sent to the DSP engine. `freq` will keep being sent unless the `Keyboard N - Send Freq` is set to zero. Since this parameter is keyboard specific, some keyboards in the interface might be used for pitch control while others might be used for other types of applications (e.g., X/Y controller, etc.). Various examples of this type of use are presented in §3.

It might be useful in some cases to number the standard `x` and `y` parameters in function of the fingers present on the screen. This can be easily accomplished by setting the `Keyboard N - Count Fingers` key to one. In that case, the first finger to touch the screen will send the `x0` and `y0` standard parameters to the DSP engine, the second finger `x1` and `y1`, and so on.

This section just gave an overview of some of the features of `faust2smartkeyb`. More details about this tool can be found in its documentation [6] as well as on the corresponding online tutorials.[7]

3. SKILL TRANSFER AND SCREEN INTERFACE: FAUST2SMARTKEYB APPS EXAMPLES

Implementation of skill transfer is one of the primary goals of `faust2smartkeyb`. It is a crucial factor in making a successful Digital Musical Instruments (DMI) as it can help accelerate its learning and make it quickly usable by a large number of performers. A wide range of screen controllers mimicking the interface of existing instruments can be implemented using the SMARTKEYBOARD interface.

This section presents a few examples where traditional acoustic instruments served as models and were turned into digital one running on mobile devices using physical models from the FAUST Physical Modeling Library [7] and `faust2smartkeyb`. We demonstrate that in most cases, the implementation of such instruments can be approached in two different ways. The first one consists of only specifying a single element of an instrument (e.g., one string of a guitar or a violin, membrane of a drum, etc.) and then use the polyphonic features of `faust2smartkeyb` to implement the ability of the instrument to generate several sounds simultaneously. In the other approach, the instrument is modeled in its whole (e.g., four strings for a violin, six strings in a guitar, etc.) and the mapping between the interface and the model is handled directly in the FAUST code.

While the goal of this section is not to be exhaustive, it should provide enough material to demonstrate how to implement most traditional musical instruments and more.

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

[7] `https://ccrma.stanford.edu/~rmichon/faustTutorials/`

3.1. Plucked Strings Instruments: the Guitar

3.1.1. Piano Keyboard Paradigm

Plucked string instruments such as the guitar, the banjo, etc. are relatively close to struck string instruments (e.g., the piano, etc.) as they are excited by punctual events (unlike bowed strings or wind instruments, where energy must be constantly introduced in the system for it to produce any sound). For this reason, controlling these types of instrument with a "piano keyboard like" interface makes a lot of sense as the performer expect sound to be heard when a key is pressed. A good commercial example of such DMI is GeoShred,[8] where a new pluck is triggered every time a finger touches a virtual string on the touch screen (this behavior might slightly change depending on the configuration of the interface).

Listing 2 presents a `faust2smartkeyb` code implementing an instrument working in a similar way as GeoShred. Six parallel keyboards are used to represent six parallel strings. They are all monophonic and implement "voice stealing" with priority to higher pitches which means that the current note is terminated when a new finger touches the same keyboard only if the pitch of the note to trigger is higher than the current one (like on a physical electric guitar string). Even though keyboards are monophonic, the overall instrument is polyphonic and several strings can be excited at the same time, taking advantage of the voice allocation system of `faust2smartkeyb`.

Keyboards are placed one fourth apart from each other, in a similar way as on a guitar neck, in order to facilitate skills transfer for guitar players. Finally, slides and vibratos can be carried out on the same string just by continuously moving the finger along the virtual keyboard.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'6',
  'Max Keyboard Polyphony':'1',
  'Mono Mode':'3','Rounding Mode':'2',
  'Keyboard 0 - Number of Keys':'13',
  [...same for all other keyboards...]
  'Keyboard 0 - Lowest Key':'72',
  'Keyboard 1 - Lowest Key':'67',
  'Keyboard 2 - Lowest Key':'62',
  'Keyboard 3 - Lowest Key':'57',
  'Keyboard 4 - Lowest Key':'52',
  'Keyboard 5 - Lowest Key':'47'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
f = hslider("freq",300,50,2000,0.01);
bend = hslider("bend",1,0,10,0.01) :
  si.polySmooth(gate,0.999,1);
gain = hslider("gain",1,0,1,0.01);
s = hslider("sustain",0,0,1,1);
t = button("gate");

// MODEL PARAMETERS
gate = t+s : min(1);
freq = f*bend : max(60);
stringLength = freq : pm.f2l;
pluckPosition = 0.8;
```

[8] `http://www.moforte.com/`

```
mute = gate : si.polySmooth(gate,0.999,1);

process =
  pm.elecGuitar(stringLength,pluckPosition,
      mute,gain,gate)
  <: _,_;
```

Listing 2: `faust2smartkeyb` app implementing an electric guitar with an isomorphic keyboard.

The electric guitar string physical model is implemented in the FAUST Physical Modeling Library as `elecGuitar()`. The effect chain is declared in a separate file in order to use the `-effect` option when using `faust2smartkeyb` and involves a distortion and a reverb:

```
process = par(i,2,ef.cubicnl(0.8,0)) : dm.
    zita_rev1;
```

The pitch of the virtual string is controlled by the combination of the `freq` and `bend` standard parameters. Strings are progressively muted when the finger leaves the string. In other words, they only resonate if the associated finger remains on the screen.

3.1.2. External Plucking Paradigm

Even though the paradigm presented previously works well with plucked string instruments, it differs from that of a real guitar because of the lack of an independent interface for exciting the different strings. Listing 3 presents a `faust2smartkeyb` app where virtual strings are excited through a separate keyboard on the touch-screen. This keyboard could be easily substituted by an external controller using MIDI.

The interface contains seven keyboards: six implementing the different strings of the guitar (and tuned the same way as on this instrument: E, A, D, G, B, E) and one used as the interface to trigger the virtual strings. `Max Keyboard Polyphony` is set to zero so that a single voice is computed when the app is launched. Indeed, unlike the previous example, the six strings of the instrument are all implemented in the same process, therefore only one voice is necessary. The `freq` and `bend` standard parameters of the first six keyboards are retrieved and used to control the pitch of the six independent strings.

The seventh keyboard is configured to have six keys (one for each string). We want a specific string to be excited when a finger touches the corresponding key. Since this system should react both to *touch* and *move* events, both event types `1` and `4` are considered when formating the value of `kb6kstatus`.

The acoustic guitar physical model used in this example is implemented in the FAUST Physical Modeling Library [7] as `nylonGuitarModel()`. Here, six models (one for each string) are computed in parallel.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'7',
  'Max Keyboard Polyphony':'0',
  'Rounding Mode':'2',
  'Keyboard 0 - Number of Keys':'14',
  [...same for other keyboards 1, 2, 3, 4,
      and 5...]
  'Keyboard 6 - Number of Keys':'6',
  'Keyboard 0 - Lowest Key':'52',
  'Keyboard 1 - Lowest Key':'57',
  'Keyboard 2 - Lowest Key':'62',
  'Keyboard 3 - Lowest Key':'67',
  'Keyboard 4 - Lowest Key':'71',
  'Keyboard 5 - Lowest Key':'76',
  'Keyboard 0 - Send Keyboard Freq':'1',
  [...same for all other keybaords...],
  'Keyboard 6 - Piano Keyboard':'0',
  'Keyboard 6 - Send Key Status':'1',
  'Keyboard 6 - Key 0 - Label':'S0',
  'Keyboard 6 - Key 1 - Label':'S1',
  'Keyboard 6 - Key 2 - Label':'S2',
  'Keyboard 6 - Key 3 - Label':'S3',
  'Keyboard 6 - Key 4 - Label':'S4',
  'Keyboard 6 - Key 5 - Label':'S5'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
kbfreq(0) =
  hslider("kb0freq",164.8,20,10000,0.01);
kbbend(0) =
  hslider("kb0bend",1,0,10,0.01);
[...same for other keyboards until kb5...]
kb6kstatus(0) =
  hslider("kb6k0status",0,0,1,1)
  <: ==(1) | ==(4) : int;
kb6kstatus(1) =
  hslider("kb6k1status",0,0,1,1)
  <: ==(1) | ==(4) : int;
[...same for all other keys of kb6...]

// MODEL PARAMETERS
sl(i) = kbfreq(i)*kbbend(i) :
  pm.f2l : si.smoo;
pluckPosition = hslider("pluckPosition
  [acc: 1 0 -10 0 10]",0.5,0,1,0.01) :
  si.smoo;

// ASSEMBLING MODELS
nStrings = 6; // number of strings
guitar = par(i,nStrings,kb6kstatus(i) : ba.
    impulsify : pm .nylonGuitarModel(sl(i),
    pluckPosition)) :> _;

process = guitar <: _,_;
```

Listing 3: `faust2smartkeyb` app implementing an acoustic guitar with an independent plucking interface.

3.2. Bowed Strings Instruments: the Violin

Unlike plucked string instruments (see §3.1), bowed string instruments must be constantly excited to generate sound. Thus, parameters linked to bowing (i.e., bow pressure, bow velocity, etc.) must be continuously controlled. The `faust2smartkeyb` code presented in Listing 4 is a violin app where each string is represented by one keyboard in the interface (in a similar way than the guitar presented in §3.1). This interface is common to all strings that are activated when they are touched on the screen.

The SMARTKEYBOARD configuration declares 5 keyboards (4 strings and one control surface for bowing). “String keyboards”

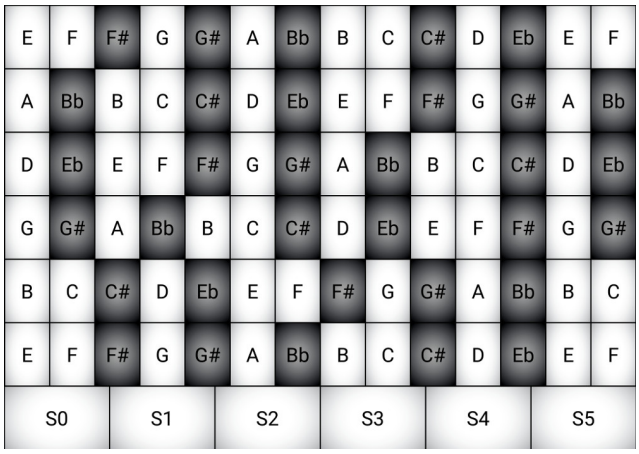

Figure 4: Screen-shot of the interface of the app generated from the code presented in Listing 3.

are tuned like on a violin (G, D, A, E) and are configured to be monophonic and implement "pitch stealing" when a higher pitch is selected (see §3.1). Bow velocity is computed by measuring the displacement of the finger touching the 5th keyboard (`bowVel`). Bow pressure just corresponds to the y position of the finger on this keyboard. Strings are activated when at least one finger is touching the corresponding keyboard (`as(i)`).

The app doesn't take advantage of the polyphony support of `faust2smartkeyb` and a single voice is constantly ran after the app is launched (`Max Keyboard Polyphony = 0`). Four virtual strings based on a simple violin string model (`violinModel()`) implemented in the FAUST Physical Modeling Library are declared in parallel and activated in function of events happening on the screen.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'5',
  'Max Keyboard Polyphony':'0',
  'Rounding Mode':'2',
  'Send Fingers Count':'1',
  'Keyboard 0 - Number of Keys':'19',
  [...same for next 3 keyboards...]
  'Keyboard 4 - Number of Keys':'1',
  'Keyboard 0 - Lowest Key':'55',
  'Keyboard 1 - Lowest Key':'62',
  'Keyboard 2 - Lowest Key':'69',
  'Keyboard 3 - Lowest Key':'76',
  'Keyboard 0 - Send Keyboard Freq':'1',
  [...same for next 3 keyboards...]
  'Keyboard 4 - Send Freq':'0',
  'Keyboard 4 - Send Key X':'1',
  'Keyboard 4 - Send Key Y':'1',
  'Keyboard 4 - Static Mode':'1',
  'Keyboard 4 - Key 0 - Label':'Bow'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
kbfreq(0) =
  hslider("kb0freq",220,20,10000,0.01);
kbbend(0) =
  hslider("kb0bend",1,0,10,0.01);
[...same for the 3 next keyboards...]
kb4k0x =
  hslider("kb4k0x",0,0,1,1) : si.smoo;
kb4k0y =
  hslider("kb4k0y",0,0,1,1) : si.smoo;
kbfingers(0) =
  hslider("kb0fingers",0,0,10,1) : int;
[...same for the 3 next keyboards...]

// MODEL PARAMETERS
// strings lengths
sl(i) = kbfreq(i)*kbbend(i) :
  pm.f2l : si.smoo;
// activates string
as(i) = kbfingers(i)>0;
bowPress = kb4k0y;
// finger displacement on screen
bowVel = kb4k0x-kb4k0x' : abs : *(8000) :
    min(1) : si.smoo;
bowPos = 0.7;

// ASSEMBLING MODELS
// essentially 4 parallel violin strings
model = par(i,4,pm.violinModel(sl(i),
    bowPress,bowVel*as(i),bowPos))
  :> _;

process = model <: _,_;
```

Listing 4: `faust2smartkeyb` app implementing a violin with an independent interface for bowing.

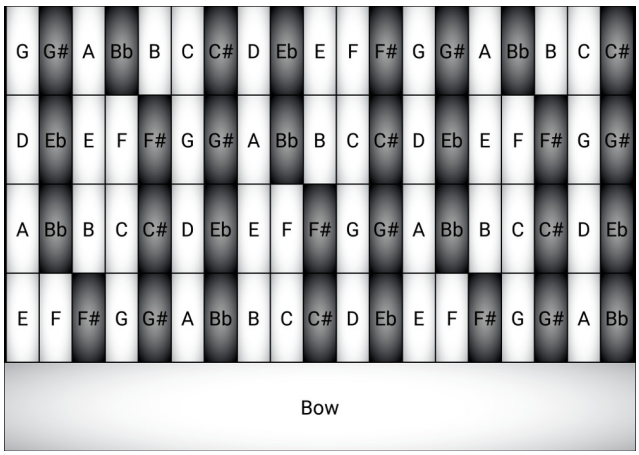

Figure 5: Screen-shot of the interface of the app generated from the code presented in Listing 4.

Alternatively, the bowing interface could be removed and the bow velocity could be calculated based on the displacement on the y axis of a finger on a keyboard, allowing one to excite the string and control its pitch with a single finger. However, concentrating so many parameters on a single gesture tends to limit the affor-

dances of the instrument. The code presented in Listing 4 could be easily modified to implement this behavior.

3.3. Percussion Instruments: Polyphonic Keyboard and Independent Instruments Paradigms

Just like plucked string instruments (see §3.1), percussion instruments can be implemented using `faust2smartkeyb` either as polyphonic instruments or as a constantly running synthesizer implementing multiple instruments in parallel. In the first case (see the djembes example below), a new voice is allocated every time the instrument is stroke. A voice might implement several models and choose one of them in function of the pad/key being touched. Another option is to use a single scalable model whose properties will change every time a voice is started. In the second case (see the bells example below), a single voice implementing several models in parallel is initiated when the app is launched and models are excited in function of the pad/key touched in the interface. The following subsections provide examples of these two paradigms.

3.3.1. Set of Djembes: Example of Polyphonic Keyboard Paradigm for Percussion Instruments

The code presented in Listing 5 implements a SMARTKEYBOARD app where three pads can be used to play three djembes of different sizes. A single model whose fundamental frequency is adjusted in function of the virtual pad being stroke is used. This app takes advantage of the polyphony system of `faust2smartkeyb` and a new voice is instantiated every time a new strike happens on the touchscreen.

The interface is made out of two polyphonic keyboards (one with two keys and one with one key). The (x, y) position of the finger on the keys/pads are retrieved and used to compute the excitation position (`exPos`) on the model. The fundamental frequency (`rootFreq`) of the model is selected in function of the pad being touched. The djembe physical model used in this program is implemented in the FAUST Physical Modeling Library.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'2',
  'Keyboard 0 - Number of Keys':'2',
  'Keyboard 1 - Number of Keys':'1',
  'Keyboard 0 - Static Mode':'1',
  'Keyboard 1 - Static Mode':'1',
  'Keyboard 0 - Send X':'1',
  'Keyboard 0 - Send Y':'1',
  'Keyboard 1 - Send X':'1',
  'Keyboard 1 - Send Y':'1',
  'Keyboard 0 - Piano Keyboard':'0',
  'Keyboard 1 - Piano Keyboard':'0',
  'Keyboard 0 - Key 0 - Label':'High',
  'Keyboard 0 - Key 1 - Label':'Mid',
  'Keyboard 1 - Key 0 - Label':'Low'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
gate = button("gate");
x = hslider("x",1,0,1,0.001);
y = hslider("y",1,0,1,0.001);
keyboard =
  hslider("keyboard",0,0,1,1) : int;
key = hslider("key",0,0,1,1) : int;

djembeInstrument =
  pm.djembe(rootFreq,exPos,strikeSharpness,
      gain,gate)
with{
  bFreq = 60; // freq of the lowest djembe
  padID = 2-(keyboard*2+key);
  rootFreq = bFreq*(padID+1);
  exPos = min((x*2-1 : abs),(y*2-1 : abs));
  strikeSharpness = 0.5;
  gain = 2;
};

process = djembeInstrument <: _,_;
```

Listing 5: `faust2smartkeyb` app implementing a set of djembes.

A similar approach could be used to map keys/pads to completely different models by declaring them in the same FAUST code (i.e., voice in this case) and activating them in function the key being touched.

3.3.2. Set of Bells: Examples of Independent Instrument Paradigm for Percussion Instruments

The code presented in Listing 6 implements a SMARTKEYBOARD app where four different bells are associated to four different pads on the touchscreen. The strike position on each pad is used to control the excitation position on the corresponding virtual bell.

The SMARTKEYBOARD interface is made out of two keyboards of two keys. A single voice is instantiated whenever the app is launched (`Max Keyboard Polyphony = 0`). Four bell physical models from the FAUST Physical Modeling Library are ran in parallel. The status of each key in the interface is retrieved and used to trigger the excitation for each bell independently.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'2',
  'Max Keyboard Polyphony':'0',
  'Keyboard 0 - Number of Keys':'2',
  'Keyboard 1 - Number of Keys':'2',
  'Keyboard 0 - Send Freq':'0',
  'Keyboard 1 - Send Freq':'0',
  'Keyboard 0 - Piano Keyboard':'0',
  'Keyboard 1 - Piano Keyboard':'0',
  'Keyboard 0 - Send Key Status':'1',
  'Keyboard 1 - Send Key Status':'1',
  'Keyboard 0 - Send X':'1',
  'Keyboard 0 - Send Y':'1',
  'Keyboard 1 - Send X':'1',
  'Keyboard 1 - Send Y':'1',
  'Keyboard 0 - Key 0 - Label':'English',
  'Keyboard 0 - Key 1 - Label':'French',
  'Keyboard 1 - Key 0 - Label':'German',
  'Keyboard 1 - Key 1 - Label':'Russian'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
```

```
kb0k0status = hslider(
  "kb0k0status",0,0,1,1) : min(1) : int;
kb0k1status = hslider(
  "kb0k1status",0,0,1,1) : min(1) : int;
kb1k0status = hslider(
  "kb1k0status",0,0,1,1) : min(1) : int;
kb1k1status = hslider(
  "kb1k1status",0,0,1,1) : min(1) : int;
x = hslider("x",1,0,1,0.001);
y = hslider("y",1,0,1,0.001);

// MODEL PARAMETERS
strikeCutoff = 6500; strikeSharpness = 0.5;
strikeGain = 1; nModes = 10;
t60 = 30; // resonance duration
nExPos = 7; // number of strike positions
exPos = min((x*2-1 : abs),(y*2-1 : abs))*
  (nExPos-1) : int;

// ASSEMBLING MODELS
bells =
    (kb0k0status : pm.strikeModel(10,
        strikeCutoff,strikeSharpness,
        strikeGain) : pm.englishBellModel(
        nModes,exPos,t60,1,3)) +
    (kb0k1status : pm.strikeModel(10,
        strikeCutoff,strikeSharpness,
        strikeGain) : pm.frenchBellModel(
        nModes,exPos,t60,1,3)) +
    (kb1k0status : pm.strikeModel(10,
        strikeCutoff,strikeSharpness,
        strikeGain) : pm.germanBellModel(
        nModes,exPos,t60,1,2.5)) +
    (kb1k1status : pm.strikeModel(10,
        strikeCutoff,strikeSharpness,
        strikeGain) : pm.russianBellModel(
        nModes,exPos,t60,1,3))
  :> *(0.2);

process = bells <: _,_;
```

Listing 6: `faust2smartkeyb` app implementing a set of bells.

This approach is often better suited for physical-model-based percussion instruments as it is much closer to how acoustic musical instrument work. Indeed, unlike the djembe examples, all bell models are constantly ran here and no concept of polyphony is used.

3.4. Wind Instruments: Key Combinations and Continuous Control

As for instruments from the previous categories treated in this section, wind instruments can be implemented with `faust2smartkeyb` using either the "polyphonic keyboard" or the "full model" paradigm. This second case is particularly relevant for wind instruments that are often monophonic and where pitch is usually selected by combining several keys (unlike a piano keyboard where one key corresponds to one pitch). The `faust2smartkeyb` code presented in Listing 7 implements a clarinet app which is meant to be ran on a small screen device (i.e., a smart-phone). The device is expected to be held with two hands with thumbs underneath and all other fingers on the screen. The instrument is played by blowing onto the built-in microphone which is used to control breath pressure. Different buttons on the screen interface represent the keys of the instrument. The y axis of the built-in accelerometer controls the "bell opening" parameter which acts as a mute on the instrument.

The screen interface is made out of two keyboards of four and five keys, respectively. The highest key on both keyboards can be used to switch between octaves (see Figure 6). The key on the first keyboard switches octaves up (`octaveShiftUp`) and the key on the second keyboard octaves down (`octaveShiftDown`). These keys are meant to be touched by the "baby finger" of both hands. Other keys reproduce a simplified version of clarinet fingerings presented in Figure 6. This mapping was designed to leverage existing skills while adapting them to what can be implemented on a touchscreen. This type of behavior is created by retrieving the status of all keys in the interface by using the `kbMkNstatus` standard parameter and comparing them to expected fingers combinations. The length of the tube of the clarinet physical model is modulated in function of all these elements. The model is part of the FAUST Physical Modeling Library. The `pressure` parameter is computed by using an envelope follower (`an.amp_follower_ud()`) on the signal of the built-in microphone of the device.

```
declare interface "SmartKeyboard{
  'Number of Keyboards':'2',
  'Max Keyboard Polyphony':'0',
  'Keyboard 0 - Number of Keys':'4',
  'Keyboard 1 - Number of Keys':'5',
  'Keyboard 0 - Send Freq':'0',
  'Keyboard 1 - Send Freq':'0',
  'Keyboard 0 - Piano Keyboard':'0',
  'Keyboard 1 - Piano Keyboard':'0',
  'Keyboard 0 - Send Key Status':'1',
  'Keyboard 1 - Send Key Status':'1',
  'Keyboard 0 - Key 3 - Label':'O+',
  'Keyboard 1 - Key 4 - Label':'O-'
}";

import("stdfaust.lib");

// SMARTKEYBOARD PARAMETERS
kb0k0status = hslider(
  "kb0k0status",0,0,1,1) : min(1) : int;
kb0k1status = hslider(
  "kb0k1status",0,0,1,1) : min(1) : int;
[...same for all other keys...]

// MODEL PARAMETERS
bellOpening = hslider(
  "bellOpening[acc: 1 1 -10 0 10]",0.5,0.3,
  0.7,0.01) : si.smoo;
basePitch = 73; // C#4
// calculate pitch shift in function of
    keys combination
pitchShift =
  ((kb0k0status == 0) & (kb0k1status == 1)
      & (kb0k2status == 0) &
    (kb1k0status == 0) & (kb1k1status == 0)
        & (kb1k2status == 0) &
    (kb1k3status == 0))*(-1) + // C
```

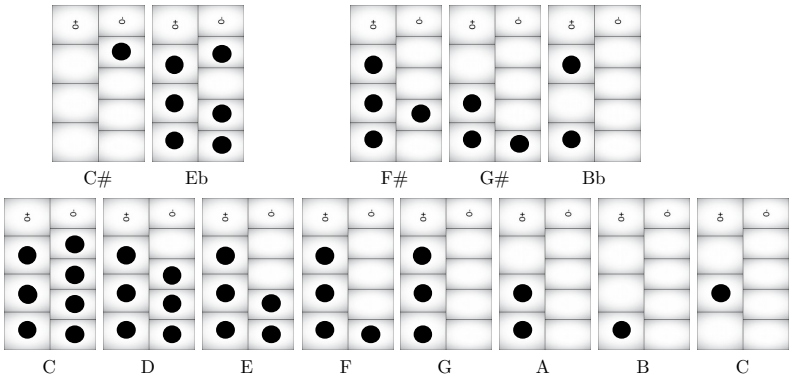

Figure 6: Fingers mapping of the interface of the app generated from the code presented in Listing 7.

```
  [...same for other notes of the chromatic
       scale...]
  ((kb0k0status == 1) & (kb0k1status == 1)
      & (kb0k2status == 1) &
    (kb1k0status == 1) & (kb1k1status == 1)
        & (kb1k2status == 1) &
    (kb1k3status == 1))*(-13); // C
octaveShiftUp =
  +(kb0k3status : ba.impulsify)~_;
octaveShiftDown =
  +(kb1k4status : ba.impulsify)~_;
octaveShift =
  (octaveShiftUp-octaveShiftDown)*(12);
tubeLength =
  basePitch+pitchShift+octaveShift :
  ba.midikey2hz : pm.f2l : si.smoo;
reedStiffness = 0.5;

model(pressure) =
  pm.clarinetModel(tubeLength,pressure,
      reedStiffness,bellOpening);

process = an.amp_follower_ud(0.02,0.02)*0.7
     : model <: _,_;
```

Listing 7: `faust2smartkeyb` app implementing a clarinet.

4. CONCLUSIONS

`faust2smartkeyb` has been tested and evaluated in the framework of several workshops[10, 11, 12] that significantly contributed to its improvement. However, it is a large project and there probably remains bugs to be fixed. Additionally, despite the fact that we haven't found a touchscreen interface for live music performance that can't be implemented with this system yet, many cases probably haven't been tested (or thought of) and there definitely exists rooms for improvements.

Thanks to new technologies and standards such as WebAssembly and the Audio Worklets, we believe that the future of mobile device apps is in the web. GRAME's research team has deployed a tremendous amount of effort to adapt FAUST to these new web standards for audio. In this context, we would like to port our SMARTKEYBOARD interface to JavaScript.

Mastering a musical instrument is a time consuming process. While skill transfer can help reduce its duration, we do not claim that the instruments presented in this paper are faster to learn than any other type of instrument. Virtuosity can be afforded by the instrument, but it still depends on the musicianship of the performer.

5. REFERENCES

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

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

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

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

[5] Romain Michon, Julius Smith, Chris Chafe, Stéphane Letz, and Yann Orlarey, "faust2api: a comprehensive api generator for android and ios," in *Proceedings of the Linux Audio Conference (LAC-17)*, Saint-Etienne, France, May 2017, Submitted for review.

[6] Romain Michon, Julius Orion Smith, and Yann Orlarey, "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 2015.

[7] Romain Michon, Julius O. Smith, Chris Chafe, Ge Wang, and Matt Wright, "The faust physical modeling library: a modular playground for the digital luthier," in *Proceedings of the 1st International Faust Conference (IFC-18)*, Mainz (Germany), 2018, Submitted for review.

[8] GRAME – Centre National de Création Musicale, Lyon, France, *FAUST Quick Reference*, June 2017.

[9] Olivier Perrotin and Christophe d'Alessandro, "Adaptive mapping for improved pitch accuracy on touch user interfaces," in *Proceedings of the International Conference on New Interfaces for Musical Expression*, Daejeon, South Korea, May 2013.

[10] "CCRMA 2016 composed instrument workshop: Intersections of 3D printing and digital audio for mobile platforms," Web-Page, 2016, URL: `https://ccrma.stanford.edu/~rmichon/composedInstrumentWorkshop/`.

[11] "Aalborg university 2017 augmented smartphone workshop," Web-Page, 2017, `https://ccrma.stanford.edu/~rmichon/copAugSmartWorkshop/`.

[12] "2017 ccrma mobile synth workshop series," Web-Page, 2017, `https://ccrma.stanford.edu/~rmichon/mobileSynth`.