Extending the Faust VST Architecture with Polyphony, Portamento and Pitch Bend

Yan Michalevsky
Department of Electrical Engineering,
Stanford University
yanm2@stanford.edu

Julius O. Smith
Center for Computer Research in Music and Acoustics (CCRMA),
Stanford University
AES Fellow
jos@ccrma.stanford.edu

Andrew Best
Blamsoft, Inc.
andrew@blamsoft.com

Abstract

We introduce the `vsti-poly.cpp` architecture for the Faust programming language. It provides several features that are important for practical use of FAUST-generated VSTi synthesizers. We focus on the VST architecture as one that has been used traditionally and is supported by many popular tools, and add several important features: polyphony, note history and pitch-bend support. These features take FAUST-generated VST instruments a step forward in terms of generating plugins that could be used in Digital Audio Workstations (DAW) for real-world music production.

Keywords

Faust, VST, Plugin, DAW

1 Introduction

FAUST [5] is a popular music/audio signal processing language developed by Yann Orlarey et al. at GRAME,[1] with contributions from a community of developers. The FAUST toolset enables the generation of standalone synthesizers as well as plugins for various operating systems and environments. Considering FAUST a convenient tool and a fast way for prototyping and even creating production level sound effects and synthesizers, we would like to use FAUST in combination with real-world music production tools and DAWs (Digital Audio Workstations).

We believe it is necessary to facilitate working with tools such as Cubase, Ableton or other DAWs providing a similar level of user experience and features. In the past ten years those tools shifted from relying on built-in PC soundblaster or external MIDI-controlled modules to a plugin based architecture. Plugins are used to generate sound and apply audio effects. Several common plugin architectures exist: VST, Apple's Audio Unit (AU), LV2 (the successor of LADSPA and DSSI under Linux OS). The VST (Virtual Studio Technology) plugin standard was released by Steinberg GmbH (famous for Cubase and other music and sound production products) in 1996, and was followed by the widespread version 2.0 in 1999 [8]. It is a particularly common format supported by many older and newer tools.

Some of the features expected from a VST plugin can be found in the VST SDK code.[2] Examining the list of MIDI events [1] can also hint at what capabilities are expected to be implemented by instrument plugins. We also draw from our experience with MIDI instruments and commercial VST plugins in order to formulate sound feature requirements.

In order for FAUST to be a practical tool for generating such plugins, it should support most of the features expected, such as the following:

- Responding to MIDI keyboard events
- Polyphony
- Portamento
- Pitch-bending (wheel controlled)
- Arpeggio
- Other effects dependent on note occurrence history

All of the plugin formats mentioned above can be generated from FAUST code with varying levels of feature support. For example, there is a very complete `faust2lv2` shell-script distributed with FAUST provided by Albert Gräf [3]. There is also a highly useful `faust2au` script by Reza Payami that is still under development. Useful VST 2.4 plugins can be generated using the `faust2vst` script, and relatively limited VSTi plugins (i.e., VST synthesizer or "instrument" plugins) can be generated using `faust2vsti`. Initial VSTi support was limited

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

[2] Specifically in the PlugCanDos namespace, declared in audioeffectx.cpp (in VST 2.4 SDK)

a single voice (implemented in the FAUST architecture file `vsti-mono.cpp`).

This paper describes the VSTi support implemented in the FAUST architecture file `vsti-poly.cpp`.[3] This effort adds polyphony support, pitch-bend, note-history, and other features described below. Pitch-bend and note history support facilitates effects such as portamento slide,[4] and creating arpeggiators. Finally, we provide an example of how it can be used to create instruments. We demonstrate using FAUST-generated VST plugins with MuLab [4] and Renoise [7] workstations. We also discuss possible future improvements and additions.

Related work

For handling MIDI events and polyphony support in a FAUST architecture file, we benefited from the MIDI plugin section of [3] and the FAUST DSSI architecture-file source code `dssi.cpp`. Additionally, `vsti-mono.cpp` was useful as a basis for our extended FAUST VSTi architecture.

2 Design

Following the convention introduced by Albert Gräf for `faust2pd` [2] and `faust2lv2` [3] et al. [6], the VST architecture file implements functionality for recognizing the "freq", "gate" and "gain" FAUST-control labels to set the note and velocity upon MIDI Note-On events (0x90) and to set the gate to 0 for a MIDI Note-Off event (0x80). One approach to implementing polyphony for the VSTi architecture is doing it similarly to the DSSI plugin architecture. The "freq", "gate" and "gain" are mapped to the controls multiple times which enables playing simultaneously a predefined maximum number of notes.

We combine the approaches taken in `vsti-mono.cpp` and `dssi.cpp`. Figure 1 shows a UML diagram describing our design (`vsti-poly.cpp`). A VST host interacts with the VST plugin through the *AudioEffectX* interface. The `Faust` class defines the functionality of the plugin by implementing that interface. The `mydsp` class performs the signal processing and synthesis—it is the code that is actually produced by the FAUST compiler. We instantiate `mydsp` for each voice (`Voice` class).

[3]It is expected that this name will later change to `vsti.cpp`. The `faust2vsti` command-line script will of course be updated as well in that case.

[4]Although for a monophonic synthesizer portamento can be implemented by smoothing the input frequency.

The VST plugin controls are created and updated using the `vstUI` class. There is an instance of `vstUI` held by the `Faust` class which is used for knobs and sliders controlled by the user via the graphical interface or by mapping MIDI controls. This instance is for controlling parameters that are global and should affect every note played. The instances of `vstUI` that are created as part of each `Voice` instance are for controlling per note parameters (frequency, gain, previously played frequency and gate). The `Faust` class implementation of the `setParameter` interface method is broadcasting any change in the global plugin parameter to all `Voice` instances.

Handling MIDI events

FAUST VSTi architecture handles MIDI events delegated by the VST host. The host sends the events to the plugin by calling `processEvents`. An event of type `kVstMidiType` indicates a MIDI event.

Note On

A MIDI note-on event (status byte is 0x9) results in searching for a free voice instance to handle the new note in the `freeVoices` list contained in the `Faust` class. The search proceeds in a classic round robin pattern as found in hardware synthesizers. If a free voice is found, the voice is designated as the new voice, otherwise the oldest playing voice is stolen and designated as the new voice. Its frequency is set according to the note number, the gain parameter is set according to the note velocity, and the gate is set to 1. An entry is added to `playingVoices`, mapping the note to the voice index, and the voice index is removed from the `freeVoices` list. The previously played note is saved in order to enable the portamento slide.

The VST format operates with multiple samples in a processing block. The note-on event includes a sample offset within the current block. These deltas are stored in a list so that multiple note-on events can be handled in the block. The note to voice allocation occurs within the processing loop, so that each note starts at its correct sample position within the block.

Note Off

A MIDI note-off event (status byte is 0x8) results in searching for the corresponding `Voice` instance in the `playingVoices` list contained in the `Faust` class. The gate is then set to 0. Because the voice may have a release tail after the gate is zeroed, a silence detection algorithm is used to determine when the voice index should

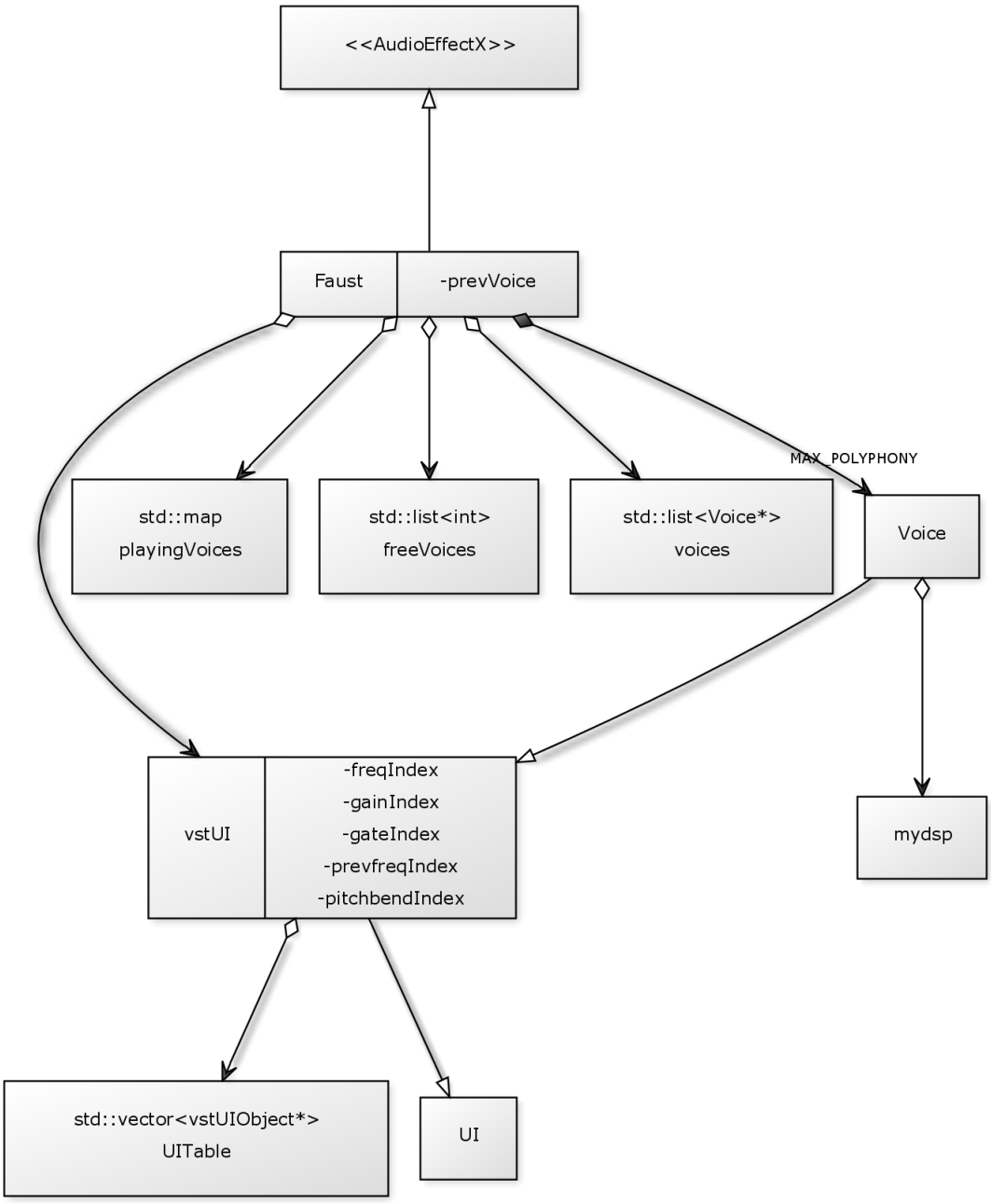

Figure 1: FAUST VSTi design

be added to the `freeVoices` list. The voice output must be below the silence threshold for an entire block before it is marked as free. Silence detection allows sounding voices to not be reallocated prematurely and also provides better CPU efficiency compared to always processing all voices. Like note-on events, note-off events are sample accurate within a block.

Pitch Bend

A MIDI pitch bend is indicated by status byte 0xE. The MIDI event pitch argument has values in the range 0..16384. We normalize it to be in the range -1..1 and broadcast the value to all voices thus affecting all currently playing notes. The frequency is not updated by the architecture, as it is the responsibility of the FAUST code to use the `pitchbend` control value. This separation enables the user to ignore or handle the pitch-bend MIDI event according to the desired behavior.

All-notes-off Event

The All-notes-off MIDI event is indicated by a note number of 0, and velocity 0. Like the single note-off event, the voice gate is set to 0 and

entered into the release silence detection state. This is done for all active voices.

Portamento Slide Implementation

We demonstrated the very common portamento slide effect by creating a FAUST VSTi based on the sawtooth synthesizer that is part of the FAUST oscillator library (`oscillator.lib`). We added a portamento control that can take values in the range 0.01..0.3. The portamento effect is achieved by mixing two exponentials, one decaying and one reaching saturation with characteristic time τ that is equal to the value of the portamento control.

$$f_{mixed} = f_{new} \cdot \left(1 - e^{-\frac{t}{\tau \cdot SR}}\right) + f_{prev} \cdot e^{-\frac{t}{\tau \cdot SR}}$$

where SR is the sampling frequency, t is the time that has passed since the new note was played and f_{new} and f_{prev} are the new and previously played frequencies, respectively. This instrument also supports pitch bending controlled by the pitch-bend wheel. The f_{new} is actually a sum of note frequency and the value of the pitch-bend control (in the range -1..1) multiplied by 20. The demo synthesizer source code is presented in Alg. 1.

A demonstration of music production using FAUST can be found at `http://stanford.edu/~yanm2/music/faustloop.mp3`.

This short loop was produced using only FAUST-generated VSTi plugins, with the exception of the drums.

Figure 2: VST plugin generated by FAUST as it appears in MuLab. Using predefined control names "freq", "gain", "gate", "prevfreq" and "pitchbend" automatically maps the controls to MIDI event parameters.

3 Installation and Basic Usage

Basic installation instructions are provided in [3]. If you are using an up-to-date version of FAUST you should already have the `vsti-poly.cpp` architecture file, and running

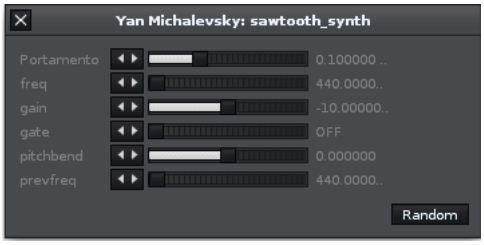

Figure 3: VST plugin generated by FAUST as it appears in Renoise tracker.

```
make install
```

should make `faust2vsti` tool accessible from any directory. Running

```
faust2vsti <yourfaustcode.dsp>
```

will create a VST effect or synthesizer from the .dsp file. To produce only the source code (.cpp) run

```
faust -a vsti-poly.cpp
-o <output filename>
<yourfaustcode.dsp>
```

`vsti-poly.cpp` currently supports both VST audio processing plugins and VSTi-MIDI-driven software synthesizer plugins. In the future we expect to consolidate all the VST related architecture files under the FAUST project.

4 Future Work

In this section we briefly offer suggestions for future development, based on our observations during this project.

Inherent portamento slide support

Portamento-slide is common to many synthesizers, for which reason it may be a good idea to incorporate the support for it into the architecture file. This effect requires a gradual change of frequency that can be performed by `vsti-poly.cpp`. The speed of transition to the new frequency could be determined by a "portamento" control as is done with other controls recognized by the architecture.

Inherent pitch-bend support

Pitch-bending is also common to many synthesizers and requires a change of frequency. This change in frequency can be done by the architecture prior to calling `mydsp::compute`[5].

[5] This of course requires the synth to use a "freq" control and not only note identifier as we suggest in the next paragraph. It would also require a way to specify the bending range.

Algorithm 1 sawtooth-synth: sawtooth with portamento and pitch-bend in FAUST

```
declare name "Sawtooth-Synth";

import("music.lib");
import("oscillator.lib");

gate = button("gate");
gain = hslider("gain[unit:dB][style:knob]", -10, -30, +10, 0.1) : db2linear : smooth(0.999);
freq = nentry("freq[unit:Hz]", 440, 20, 20000, 1);
prevfreq = nentry("prevfreq[unit:Hz]", 440, 20, 20000, 1);
portamento = vslider("[5] Portamento [unit:sec] [style:knob] [tooltip: Portamento (frequency-glide)
time-constant in seconds]", 0.1,0.01,0.3,0.001);
pitchbend = vslider("pitchbend", 0, -1, 1, 0.01);

start_time = latch(freq == freq', time);
dt = time - start_time;
expo(tau) = exp(0-dt/(tau*SR));
mix(tau, f, pf) = f*(1 - expo(tau)) + pf*expo(tau);
bended_freq = freq + pitchbend * 20;
sfreq = mix(portamento, bended_freq, prevfreq) : min(20000) : max(20);

x = sawtooth(sfreq : smooth(0.999));
process = x * gain * (gate);
```

Setting note identifier control in addition to frequency

Currently the pitch is set by a "freq" control, used by the FAUST code to determine the frequency. The "freq" control value is set by the architecture according to the note identifier received in the MIDI Note-On event. Sometimes it is more useful to have the note identifier or piano key identifier. For instance, there are existing FAUST synthesizers that take the key as input. A percussion synthesizer that produces a different sound for every key would possibly use a key identifier instead of note frequency. It would be therefore a welcome addition to the vsti-poly architecture to set the value of a note identifier control on each Note-On event.

Extended note history

We currently save only the previously played frequency enabling the implementation of the portamento-slide. Synthesizers that produce chords or arpeggios may require information about more previously played notes. This would be enabled by extending the saved note history. Passing these values to the FAUST code would require instantiation of multiple note or frequency controls.

Single FAUST VST architecture file

Currently there are several FAUST architecture files related to VST: `vst.cpp, vst2p4.cpp, vsti-mono.cpp and vsti-poly.cpp`. While theses have been kept side-by-side to not interfere with other users during development of each new architecture file, they are redundant and should be consolidated into a single `vsti.cpp` architecture file.

Shared signals among multiple voices

Many synthesizers offer modulation sources that affect multiple voices simultaneously. For example, an LFO can modulate pitch or waveform on all voices in a polyphonic synth. In the future it would be beneficial if shared signal support was provided to the synthesizer designer.

Enhanced GUI support

Other architectures within the FAUST ecosystem have more features in their GUI layout capabilities. The grouping of controls into subsections and providing specification of knobs vs. sliders would provide better flexibility and organization comparable to hand coded VSTi plugins.

Further Host-Plugin integration

One simple yet useful feature to implement is the `Bypass` capability, enabling the user to turn off a plugin from the host.

More information provided to the plugin by the host includes time and tempo. This can be useful for implementing arpeggio instruments, or audio effects dependent on tempo, such as

gating or synchronized echo.

Consolidation of various VST related architecture files

At the time of writing, FAUST code contains multiple architecture variants pertaining to VST: `vst.cpp` and `vst2p4.cpp` for effects, `vsti-mono.cpp` for monophonic instruments and `vsti-poly.cpp`, introduced by this work, supporting effects, polyphonic instruments and other features discussed in the paper. We suggest there should be one architecture encorporating all mentioned functionality. Meanwhile, since portamento is very relevant to monophonic instruments, we added the necessary modifications to support the effect in `vsti-mono.cpp`, as well as support for pitch-bend.

5 Conclusion

We presented the `vsti-poly.cpp` FAUST architecture file and its new features: polyphony, pitch-bend and note-history. We used these features in the implementation of a polyphonic sawtooth synthesizer with pitch-bend and portamento slide support, and demonstrated it in a short musical loop, recorded in a popular DAW. We also suggest ideas for further development of VSTi support in FAUST which will contribute to easier implementation of common synthesizer features. The ideas presented here are not limited to the VSTi architecture but could also serve as a reference for implementing FAUST architectures for other plugin formats.

References

MIDI Manufacturers Association. MIDI messages. `http://www.midi.org/techspecs/-midimessages.php`.

Albert Gräf. Interfacing Pure Data with FAUST. In *Proc. 5th Int. Linux Audio Conf. (LAC-07), TU Berlin, http://www.kgw.tu-berlin.de/~lac2007/-proceedings.shtml*, 2007. `http://www.grame.fr/ressources/-publications/lac07.pdf`.

Albert Gräf. Creating LV2 plugins with Faust. In *Proc. 11th Int. Linux Audio Conf. (LAC-13), Graz, http://lac.linuxaudio.org/*, 2013. `http://wiki.faust-lv2.googlecode.com/-hg/faust-lv2-lac13-full.pdf`.

MuTools. Mulab, `http://www.mutools.com/`. `http://www.mutools.com/-mulab-product.html`.

Yann Orlarey, Dominique Fober, and Stephane Letz. Faust: an efficient functional approach to DSP programming. *New Computational Paradigms for Computer Music*, 2009.

Yann Orlarey, Albert Gräf, and Stefan Kersten. DSP programming with FAUST, Q and SuperCollider. In *Proc. 4th Int. Linux Audio Conf. (LAC-06), ZKM Karlsruhe, http://lac.zkm.de/2006/proceedings.shtml*, pages 39–40, 2006. `http://lac.zkm.de/-2006/proceedings.shtml#orlarey_et_al`.

Renoise. Renoise, `http://www.renoise.com`.

Wikipedia. Virtual studio technology.