Towards (even more) practical Faust: Polyphony, Portamento and Pitch Bend in Faust VSTi-s

Music 420B project, Yan Michalevsky, Student ID: 05768838

June 12, 2013

Abstract

Faust is a musical signal processing language. The Faust compiler and the toolset provided along with it enable generating standalone synthesizers and plug-ins for various architectures. We noticed that while being a very useful tool for sound synthesis its VSTi plug-ins lack several critical features for practical usage in combination with music production software and digital audio workstations (DAW). We focus on the VST architecture as one that has been used traditionally and is supported by many tools and add several important features: polyphony, note history and pitch-bend support. The contribution of polyphony is a step towards making Faust a practical tool for real-wold music production.

Introduction

Faust [4] is a language for describing musical signal processing blocks and the toolset that comes with it enables generation of standalone synthesizers as well as plug-ins for different architectures. Feeling that Faust is a convenient tool and a fast way for prototyping and even creating production level sound effects and synthesizers we want to use Faust in combination with real-world music production tools and DAWs (Digital Audio Workstations).

We believe that it is necessary to support working with tools like Cubase, Logic Pro, Ableton or other DAWs providing a similar level of user experience and similar features. In the past ten years those tools shifted from relying on built-in PC sound-blaster or external MIDI-controlled modules to a plug-in based architecture. Plug-ins are used to generate sound and apply audio effects. Several common plug-in architectures exist: VST (Virtual Studio Technology by Steinberg), Apple's Audio Unit, LV2 (the successor of LADSPA under the Linux OS) and DSSI. Steinberg's VST is a particularly common format supported by many older and newer tools.

In order for Faust to be a practical tool for generating plug-ins it has to support most of the features expected from such plug-ins. Some of the features expected from instrument plug-ins are

Figure 1: VST plug-in generated by Faust as it appears in MuLab. Using predefined control names "freq", "gain" and "gate" automatically maps the controls to MIDI event parameters.

Figure 2: VST plug-in generated by Faust as it appears in Renoise tracker.

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

Faust Generated VST plug-ins currently did not support polyphony which prevented them from being useful in common audio workstations. They were also lacking pitch-bend support and note history which prevent having effects such as portamento slide and creating arpeggiators. Also, Faust generated VST plug-ins seem incompatible with some popular workstations such as Ableton live. The workstations that we found to support Faust generated VSTs are MuLab by MuTools [3] and Renoise [1].

This project focuses on adding polyphony, portamento slide and pitch-bend support to VST plug-ins generated by Faust.

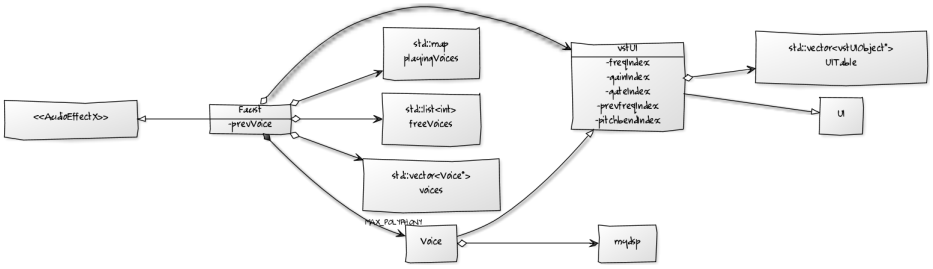

Figure 3: Faust VSTi design

Related work

Useful resources for understanding how polyphony and MIDI events can be implemented in an architecture layer are the section about MIDI plug-ins in [2] and the DSSI architecture source code under the Faust project (dssi.cpp). vsti-mono.cpp is was very useful as a basis to our extended Faust VSTi architecture.

Design

Currently, the VST architecture file implements functionality for recognizing the "freq", "gate" and "gain" keywords to set the note and velocity upon MIDI Note-On events (0x90) and to set the gate to 0 upon MIDI Note-Off event (0x80). One approach for adding polyphony to VST architecture is to do it in a manner similar to the way it is done for DSSI plug-ins. The "freq", "gate" and "gain" are mapped to the controls multiple times which enables playing simultaneously a predefined maximum number of notes. Another approach would be to simply create multiple controls for frequency, gate and gain and map each one of them to a different voice. For supporting effects such as portamento or arpeggio we have to keep a history of recently pressed notes and a list of currently pressed notes.

We chose to implement those features combining the approaches taken in vsti-mono.cpp and dssi.cpp. Figure 3 shows a UML diagram describing our design. The VST host interacts with the VST plug-in through the *AudioEffectX* interface. The *Faust* class implements this interface and defines the functionality of the plug-in. *mydsp* class is the piece that performs the signal processing and synthesis, that is the code that is produced by the Faust compiler. There is an instance of *mydsp* for each voice (*Voice* class).

The VST plug-in 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 plug-in parameter to all *Voice* instances.

Handling MIDI events

Faust VSTi architecture handles MIDI events delegated by the VST host. The host send the events to the plug-in by calling *processEvents*. An event type of *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. If a free voice is found its frequency is set according to the note number, gain parameter is set according to the note velocity and 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.

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 and the voice index is added to the *freeVoices* list.

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 ignoring or handling the pitch-bend MIDI event according to the desired behavior.

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 to 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

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

```
declare name "Bastard-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(gate, time); dt = time - start_time;
expo(tau) = exp(0-dt/(tau*SR));
shmooz(tau, f, pf) = f*(1 - expo(tau)) + pf*expo(tau);
bended_freq = freq + pitchbend * 20;
sfreq = shmooz(portamento, bended_freq, prevfreq) : min(20000) : max(20);

x = sawtooth(sfreq);
process = x * gain * (gate) : _ <: _,_;
```

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 in http://stanford.edu/∼yanm2/music/faustloop.mp3.

This short loop was produced using Faust generated VSTi-s with the only exception of drums.

Future Work

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

Inherent portamento slide support

Portamento-slide is common to many synths and as such it might be a good idea to incorporate the support for it into the architecture. It 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 it is done with other controls recognized by the architecture.

Inherent pitch-bend support

Pitch-bending is also common to many synths and requires a change of frequency. The change in frequency can be done by the architecture prior to calling *mydsp::compute*. This of course requires the synth to use a "freq" control and not only note identifier as we suggest in the next paragraph.

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 definitely use a key identifier and 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. We might think of synths that produce chords or arpeggios and require information about more previously played notes. Extending the saved note history would enable that. Passing these values to the Faust code would require instantiation of multiple note or frequency controls.

References

[1] Renoise, http://www.renoise.com.

[2] Albert Gräf. Creating lv2 plugins with faust.

[3] MuTools. Mulab, http://www.mutools.com/mulab-product.html.

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