Interfacing Pure Data with Faust

Albert GRÄF
Dept. of Music Informatics, Johannes Gutenberg University
55099 Mainz, Germany,
ag@muwiinfa.geschichte.uni-mainz.de

Abstract

This paper reports on a new plugin interface for Grame's functional DSP programming language Faust. The interface allows Faust programs to be run as externals in Miller Puckette's Pd (Pure Data), making it possible to extend Pd with new audio objects programmed in Faust. The software also includes a script to create wrapper patches around Faust units which feature "graph-on-parent" GUI elements to facilitate the interactive control of Faust units. The paper gives a description of the interface and illustrates its usage by means of a few examples.

Keywords

Computer music, digital signal processing, Faust programming language, functional programming, Pd, Pure Data

1 Introduction

Faust is a modern-style functional language for programming digital signal processing algorithms being developed at Grame [1; 2], which was already presented in-depth at last year's Linux Audio Conference [3]. Faust provides an executable, high-level specification language for describing block diagrams operating on audio signals. Signals are modelled as functions of (discrete) time and DSP algorithms as higher-order functions operating on signals. The main advantages of this approach over using graphical block diagrams is that the building blocks of signal processing algorithms can be combined in much more flexible ways, and that Faust can also serve as a formal specification language for signal processing units.

Faust programs are compiled to efficient C++ code which can be used in various environments, including Jack, LADSPA, Max/MSP, SuperCollider, VST and the Q programming language. This paper reports on Faust's plugin interface for Miller Puckette's *Pure Data* a.k.a. *Pd* (`http://puredata.info`). The new interface allows audio developers and Pd users to run Faust programs as Pd externals, in order to test Faust programs using Pd's convenient graphical environment, or to extend Pd with new custom audio objects. The package also includes a Q script `faust2pd` which can create wrapper patches featuring "graph-on-parent" GUIs around Faust externals, which further facilitates the interactive control of Faust units in the Pd environment.

The software described in this paper is free (GPL'ed). It is already included in recent Faust releases (since version 0.9.8.6), and is also available as a separate package `faust2pd` from `http://q-lang.sf.net`, which includes the `puredata.cpp` Faust architecture file, the `faust2pd` script and supporting Pd abstractions, as well as a bunch of examples. You can also try Faust interactively, without having to install the Faust compiler, at Grame's Faust website (`http://faust.grame.fr`).

Because of lack of space we cannot give an introduction to Faust and Pd here, so the paper assumes a passing familiarity with both. More information about these systems can be found in the documentation available at `http://faust.grame.fr` and `http://puredata.info`.

2 Building Faust externals

Faust Pd plugins work in much the same way as the well-known `plugin~` object (which interfaces to LADSPA plugins), except that each Faust DSP is compiled to its own Pd external. Under Linux, the basic compilation process is as follows (taking the freeverb module from the Faust distribution as an example):

```
# compile the Faust source to a C++ module
# using the "puredata" architecture
faust -a puredata.cpp freeverb.dsp
  -o freeverb.cpp
# compile the C++ module to a Pd plugin
g++ -shared -Dmydsp=freeverb freeverb.cpp
  -o freeverb~.pd_linux
```

By these means, a Faust DSP named `XYZ` with N audio inputs and M audio outputs becomes a Pd object `XYZ~` with $N+1$ inlets and $M+1$ outlets. The leftmost inlet/outlet pair is for control messages only. This allows you to inspect and change the controls the unit provides, as detailed below. The remaining inlets and outlets are the audio inputs and outputs of the unit, respectively. For instance, `freeverb.dsp` becomes the Pd object `freeverb~` which, in addition to the control inlet/outlet pair, has 2 audio inputs and outputs.

When creating a Faust object it is also possible to specify, as optional creation parameters, an extra unit name (this is explained in the following section) and a sample rate. If no sample rate is specified explicitly, it defaults to the sample rate at which Pd is executing. (Usually it is not necessary or even desirable to override the default choice, but this might occasionally be useful for debugging purposes.)

In addition, there is also a Q script named `faust2pd`, described in more detail below, which allows you to create Pd abstractions as "wrappers" around Faust units. The wrappers generated by `faust2pd` can be used in Pd patches just like any other Pd objects. They are much easier to operate than the "naked" Faust plugins themselves, as they also provide "graph-on-parent" GUI elements to inspect and change the control values.

Note that, just as with other Pd externals and abstractions, the compiled `.pd_linux` modules and wrapper patches must be put somewhere where Pd can find them. To these ends you can either move the files into the directory with the patches that use the plugin, or you can put them into the `lib/pd/extra` directory or some other directory on Pd's library path for system-wide use.

3 The control interface

Besides the DSP algorithm itself, Faust programs also contain an abstract "user interface" definition from which the control interface of a Faust plugin is constructed. The Faust description of the user interface comprises various abstract GUI elements such as buttons, checkboxes, number entries and (horizontal and vertical) sliders as well as the initial value and range of the associated control values, which are specified in the Faust source by means of the builtin functions `button`, `checkbox`, `nentry`, `hslider` and `vslider`. Besides these "active" elements which are used to input control values into the Faust program, there are also "passive" elements (`hbargraph`, `vbargraph`) which can be used to return control values computed by the Faust program to the client application.

It is also possible to specify a hierarchical layout of the GUI elements by means of appropriate "grouping" elements which are implemented by the Faust functions `hgroup`, `vgroup` and `tgroup` (`hgroup` and `vgroup` are for horizontal and vertical layouts, respectively, whereas `tgroup` is intended for "tabbed" layouts). Each GUI element (including the grouping elements) has an associated label (a string) by which the element can be identified in the client application. More precisely, each GUI element is uniquely identified by the path of labels in the hierachical layout which leads up to the given element. For further details we refer the reader to the Faust documentation [4].

To implement the control interface on the Pd side, the control inlet of a Faust plugin understands a number of messages which allow to determine the available controls as well as change and inspect their values:

- The `bang` message reports all available controls of the unit on the control outlet. The message output for each control contains the type of control as specified in the Faust source (`checkbox`, `nentry`, etc.), its (fully qualified) name, its current value, and its initial, minimum, maximum and stepsize values as specified in the Faust source.
- The `foo 0.99` message sets the control `foo` to the value 0.99, and outputs nothing.
- Just `foo` outputs the (fully qualified) name and current value of the `foo` control on the control outlet.

Control names can be specified in their fully qualified form (giving the complete path of a control, as explained above), like e.g. `/gnu/bar/foo` which indicates the control `foo` in the subgroup `bar` of the topmost group `gnu`, following the hierarchical group layout defined in the Faust source. This lets you distinguish between different controls with the same name which are located in different groups. To find out about all the controls of a unit and their fully qualified names, you can bang the control inlet of the unit as described above, and connect its control outlet to a `print` object, which will cause the descriptions of all controls to be

printed in Pd's main window. (The same information can also be used, e.g., to initialize Pd GUI elements with the proper values. Patches generated with `faust2pd` rely on this.)

You can also specify just a part of the control path (like `bar/foo` or just `foo` in the example above) which means that the message applies to *all* controls which have the given pathname as the final portion of their fully qualified name. Thus, if there is more than one `foo` control in different groups of the Faust unit then sending the message `foo` to the control inlet will report the fully qualified name and value for each of them. Likewise, sending `foo 0.99` will set the value of all controls named `foo` at once.

Concerning the naming of Faust controls in Pd you should also note the following:

- A unit name can be specified at object creation time, in which case the given symbol is used as a prefix for all control names of the unit. E.g., the control `/gnu/bar/foo` of an object `baz~` created with `baz~ baz1` has the fully qualified name `/baz1/gnu/bar/foo`. This lets you distinguish different instances of an object such as, e.g., different voices of a polyphonic synth unit.
- Pd's input syntax for symbols is rather restrictive. Therefore group and control names in the Faust source are mangled into a form which only contains alphanumeric characters and hyphens, so that the control names are always legal Pd symbols. For instance, a Faust control name like `"meter #1 (dB)"` will become `meter-1-dB` which can be input directly as a symbol in Pd without any problems.
- "Anonymous" groups and controls (groups and controls which have empty labels in the Faust source) are omitted from the path specification. E.g., if `foo` is a control located in a main group with an empty name then the fully qualified name of the control is just `/foo` rather than `//foo`. Likewise, an anonymous control in the group `/foo/bar` is named just `/foo/bar` instead of `/foo/bar/`.

Last but not least, there is also a special convenience control named `active` which is generated automatically. The default behaviour of this control is as follows:

- When `active` is nonzero (the default), the unit works as usual.
- When `active` is zero, and the unit's number of audio inputs and outputs match, then the audio input is simply passed through.
- When `active` is zero, but the unit's number of audio inputs and outputs do *not* match, then the unit generates silence.

The `active` control frequently alleviates the need for special "bypass" or "mute" controls in the Faust source. However, if the default behaviour of the generated control is not appropriate you can also define your own custom version of `active` explicitly in the Faust program; in this case the custom version will override the default one.

4 Basic example

Let's take a look at a simple example to see how these Faust externals actually work in Pd. The patch shown on the right of Figure 1 features a Faust external `tone~` created from the Faust source shown on the left of the figure. The Faust program implements a simple DSP, a sine oscillator with zero audio inputs and stereo (i.e., two) audio outputs, which is controlled by means of three control variables `vol` (the output volume), `pan` (the stereo panning) and `pitch` (the frequency of the oscillator in Hz). Note that in the patch the two audio outlets of the `tone~` unit are connected to a `dac~` object so that we can listen to the audio output produced by the Faust DSP.

Several messages connected to the control inlet of the `tone~` object illustrate how to inspect and change the control variables. For instance, by sending a `bang` to the control inlet, we obtain a description of the control parameters of the object printed in Pd's main window, which in this case looks as follows:

```
print: nentry /faust/pan 0.5 0.5 0 1 0.01
print: nentry /faust/pitch 440 440 20 20000 0.01
print: nentry /faust/vol 0.3 0.3 0 10 0.01
```

Clicking the `vol 0.1` message changes the `vol` parameter of the unit. We can also send the message `vol` to show the new value of the control, which is reported as follows:

```
print: /faust/vol 0.1
```

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

// control variables

vol     = nentry("vol", 0.3, 0, 10, 0.01);
pan     = nentry("pan", 0.5, 0, 1, 0.01);
pitch   = nentry("pitch", 440, 20, 20000, 0.01);

// simple sine tone generator

process = osci(pitch)*vol : panner(pan);
```

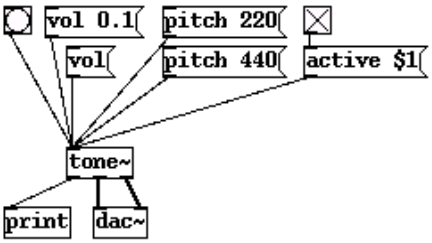

Figure 1: Basic Faust example

In the same fashion we can also set the `pitch` control to change the frequency of the oscillator. Moreover, the `active` control (which is not defined in the Faust source, but created automatically by the Pd-Faust plugin interface) allows to switch the unit on and off. The example patch allows this control to be operated by means of a toggle button.

5 Wrapping Faust DSPs with faust2pd

Controlling bare Faust plugins in the way sketched out in the preceding section can be a bit cumbersome, so the `faust2pd` package also provides a Q script `faust2pd.q` which can generate "wrapper" patches featuring additional graph-on-parent GUIs. Most of the sample patches in the `faust2pd` package were actually created that way. To use the script, you'll also need the Q interpreter available from `http://q-lang.sf.net`. The `faust2pd` package contains instructions on how to install the script and the supporting Pd abstractions on your system.

The graph-on-parent GUIs of the wrapper patches are *not* created from the Faust source or the compiled plugin, but from the XML descriptions (`dsp.xml` files) Faust generates when it is run with the `-xml` option. Such an XML file contains a readable description of the complete hierarchy of the control elements defined in the Faust program, and includes all necessary information to create a concrete rendering of the abstract user interface in the Faust source. The `faust2pd` script is able to read this XML description and create the corresponding Pd GUI along with the necessary control logic.

The script is run as `faust2pd filename.dsp.xml`; this will create a Pd patch named `filename.pd` from the Faust XML description in `filename.dsp.xml`. The `faust2pd` program understands a number of options which affect the layout of the GUI elements and the contents of the generated patch; you can also run `faust2pd -h` for information about these additional options.

On Linux, the compilation of a Faust DSP and creation of the Pd patch typically involves the following steps (again taking the freeverb module from the Faust distribution as an example):

```
# compile the Faust source and generate
# the xml file
faust -a puredata.cpp -xml freeverb.dsp
  -o freeverb.cpp
# compile the C++ module to a Pd plugin
g++ -shared -Dmydsp=freeverb freeverb.cpp
  -o freeverb~.pd_linux
# generate the Pd patch from the xml file
faust2pd freeverb.dsp.xml
```

Just like the Faust plugin itself, the generated patch has a control input/output as the leftmost inlet/outlet pair, and the remaining plugs are signal inlets and outlets for each audio input/output of the Faust unit. However, the control inlet/outlet pair works slightly different from that of the Faust plugin. Instead of being used for control replies, the control outlet of the patch simply passes through its control input (after processing messages which are understood by the wrapped plugin). By these means control messages can flow along with the audio signal through an entire chain of Faust units. Moreover, when generating a polyphonic synth patch using the `-n` a.k.a. `--nvoices` option there will actually be two control inlets, one for note messages and one for ordinary control messages. (This is illustrated by the examples in the following section.)

The generated patch also includes the necessary GUI elements to see and change all (active and passive) controls of the Faust unit. Faust control elements are mapped to Pd GUI elements in an obvious fashion, following the hori-

zontal and vertical layout specified in the Faust source. The script also adds special buttons for resetting all controls to their defaults and to operate the special `active` control.

This generally works very well, but you should be aware that the control GUIs generated by `faust2pd` are somewhat hampered by the limited range of GUI elements available in a vanilla Pd installation:

- There are no real "button" widgets as required by the Faust specification, so "bangs" are used instead. There is a global delay time for switching the control from 1 back to 0, which can be changed by sending a value in milliseconds to the `faust-delay` receiver. If you need interactive control over the switching time then it is better to use checkboxes instead, or you can have `faust2pd` automatically substitute checkboxes for all buttons in a patch by invoking it with the `-f` a.k.a. `--fake-buttons` option.
- Sliders in Pd do not display their value in numeric form so it may be hard to figure out what the current value is. Therefore `faust2pd` has an option `-s` a.k.a. `--slider-nums` which causes it to add a number box to each slider control. (This flag also applies to Faust's passive bargraph controls, as these are implemented using sliders, see below.)
- Pd's sliders also have no provision for specifying a stepsize, so they are an awkward way to input integral values from a small range. On the other hand, Faust doesn't support the "radio" control elements which Pd provides for that purpose. As a remedy, `faust2pd` allows you to specify the option `-r MAX` (a.k.a. `--radio-sliders=MAX`) to indicate that sliders with integral values from the range `0..MAX-1` are to be mapped to corresponding Pd radio controls.
- Faust's "bargraphs" are emulated using sliders. Note that these are passive controls which just display a value computed by the Faust unit. A different background color is used for these widgets so that you can distinguish them from the ordinary (active) slider controls. The values shown in passive controls are sampled every 40 ms by default. You can change this value by sending an appropriate message to the global `faust-timer` receiver.
- Since Pd has no "tabbed" (notebook-like) GUI element, Faust's "tgroups" are mapped to "hgroups" instead. It may be difficult to present large and complicated control interfaces without tabbed dialogs, though. As a remedy, you can control the amount of horizontal or vertical space available for the GUI area with the `-x` and `-y` (a.k.a. `--width` and `--height`) options and `faust2pd` will then try to break rows and columns in the layout to make everything fit within that area.
- You can also exclude certain controls from appearing in the GUI using the `-X` option. This option takes a comma-separated list of shell glob patterns indicating either just the names or the fully qualified paths of Faust controls which are to be excluded from the GUI. For instance, the option `-X 'volume,meter*,faust/resonator?/*'` will exclude all `volume` controls, all controls whose names start with `meter`, and all controls in groups matching `faust/resonator?`.
- Faust group labels are not shown at all, since there seems to be no easy way to draw some kind of labelled frame in Pd.

Despite these limitations, `faust2pd` appears to work rather well, at least for the kind of DSPs found in the Faust distribution. Still, for more complicated control surfaces and interfaces to be used on stage you'll probably have to edit the generated GUI layouts by hand.

6 Faust2pd examples

Figure 2 shows the Faust program of a simple chorus unit. On the right side of the figure you see the corresponding object generated with `faust2pd` with its graph-on-parent area, as it is displayed in a parent patch. The object has three inlet/outlet pairs, one for the control messages and two for the stereo input and output signals. For this abstraction, we ran `faust2pd` with the `-s` a.k.a. `--slider-nums` options so that each `hslider` control in the Faust source is represented by a pair of horizontal slider and number GUI elements in the Pd patch.

If you open the `chorus` object inside Pd you can have a closer look at the contents of the patch (Figure 3). Besides the graph-on-parent area with the GUI elements, it contains the `chorus~` external itself along with inlets/outlets and receivers/senders for the control and audio

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

level   = hslider("level", 0.5, 0, 1, 0.01);
freq    = hslider("freq", 2, 0, 10, 0.01);
dtime   = hslider("delay", 0.025, 0, 0.2, 0.001);
depth   = hslider("depth", 0.02, 0, 1, 0.001);

tblosc(n,f,freq,mod)
        = (1-d)*rdtable(n,waveform,i&(n-1)) +
          d*rdtable(n,waveform,(i+1)&(n-1))
with {
  waveform        = time*(2.0*PI)/n : f;
  phase           = freq/SR : (+ : decimal) ~ _;
  modphase        = decimal(phase+mod/(2*PI))*n;
  i               = int(floor(modphase));
  d               = decimal(modphase);
};

chorus(d,freq,depth)    = fdelay(1<<16, t)
with { t = SR*d/2*(1+depth*tblosc(1<<16, sin, freq, 0)); };

process                 = vgroup("chorus", (c, c))
with { c(x) = x+level*chorus(dtime,freq,depth,x); };
```

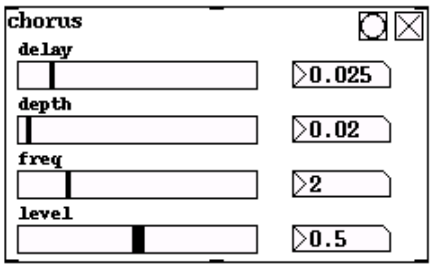

Figure 2: Faust chorus patch

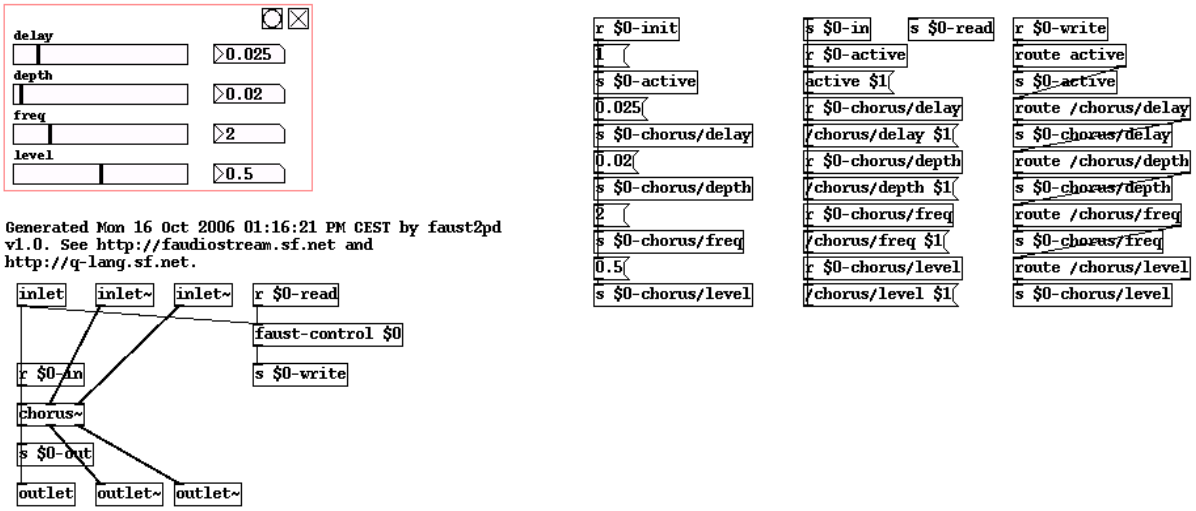

Figure 3: Inside the chorus patch

inputs and outputs (on the left side of the figure, below the GUI area) and the control logic for the GUI elements (on the right side). Of course, the generated contents of the patch can also be edited manually as needed.

It is also possible to generate polyphonic synth patches. Figure 4 shows a simple example, an additive synthesizer. On this Faust program we invoked `faust2pd` with the `-n` a.k.a. `--nvoices` option which specifies the desired number of voices. The generated abstraction then contains as many instances of the Faust external as given with the `-n` option. The resulting patch does not have any audio inputs, but *two* control inputs instead of one. While the right control inlet takes Faust control messages which are sent to all the Faust objects (a.k.a. "voices") simultaneously, the left inlet takes triples of numbers consisting of a voice number, a note number and a velocity value and translates these to the appropriate `freq`, `gain` and `gate` messages for the corresponding voice.

(At this time, the names of the three special voice controls are hard-wired into the `faust2pd`

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

// control variables

vol     = hslider("vol", 0.3, 0, 10, 0.01);
pan     = hslider("pan", 0.5, 0, 1, 0.01);
attack  = hslider("attack", 0.01, 0, 1, 0.001);
decay   = hslider("decay", 0.3, 0, 1, 0.001);
sustain = hslider("sustain", 0.5, 0, 1, 0.01);
release = hslider("release", 0.2, 0, 1, 0.001);

// voice controls

freq    = nentry("freq", 440, 20, 20000, 1);
gain    = nentry("gain", 0.3, 0, 10, 0.01);
gate    = button("gate");

// additive synth: 3 sine oscillators with adsr envelop

process = (osc(freq)+0.5*osc(2*freq)+0.25*osc(3*freq))
  * (gate : vgroup("1-adsr", adsr(attack, decay, sustain, release)))
  * gain : vgroup("2-master", *(vol) : panner(pan));
```

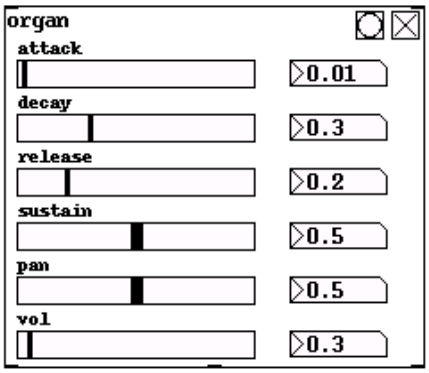

Figure 4: Faust organ patch

script, so Faust programs must follow this standard interface if they are to be used as synth units.)

Both kinds of patches can then easily be arranged to the usual synth-effect chains, as shown in Figure 5. In this example we combined the organ and chorus patches from above with another effect unit generated from the freeverb module in the Faust distribution, and added a frontend which translates incoming MIDI messages and a backend which handles the audio output and displays a dB meter. (The latter two components are just plain Pd abstractions.)

These sample patches can all be found in the `faust2pd` package, along with a bunch of other instructive examples, including the full collection of example DSPs from the Faust distribution, more polyphonic synth examples and a pattern sequencer demo.

7 Conclusion

The Pd-Faust external interface and the `faust2pd` script described in this paper make it easy to extend Pd with new audio processing objects without having to resort to C programming. Faust programs are concise and comparatively easy to write (once the initial learning curve has been mastered), and can easily be ported to other plugin architectures such as LADSPA and VST by simply recompiling the Faust source. Still the efficiency of the generated code can compete with carefully hand-coded C routines, and sometimes even outperform these, because of the sophisticated optimizations applied by the Faust compiler.

The Pd-Faust interface is especially useful for DSPs which cannot be implemented directly in Pd in a satisfactory manner, like the Karplus-Strong algorithm, because of Pd's 1-block minimum delay restriction for feedback loops. But it is also suitable for implementing all kinds of specialized DSP components like filter designs which could also be done directly in Pd but not with the same efficiency. Last but not least, the interface also gives you the opportunity to make use of the growing collection of readily available Faust programs for different audio processing needs.

The Pd-Faust interface is of course only suitable for creating audio objects. However, there is also a companion *Pd-Q* plugin interface for the Q programming language [5], also available at `http://q-lang.sf.net`. Together, the `faust2pd` package and Pd-Q provide a complete functional programming environment for extending Pd with custom audio and control objects.

Future work on Faust will probably concentrate on making the language still more flexible and easy to use, on providing an extensive collection of DSP algorithms for various purposes,

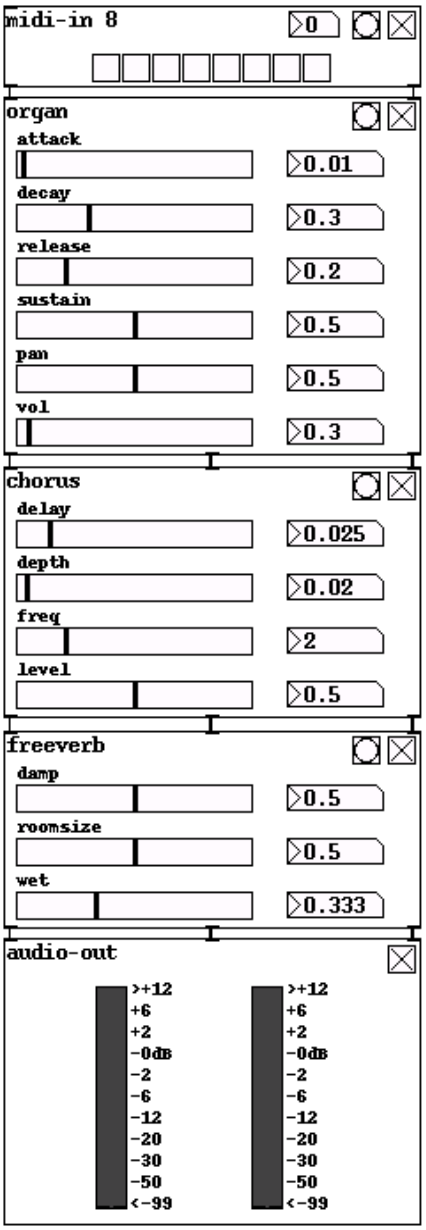

Figure 5: Synth-effects chain

and on adding support for as many target DSP architectures and platforms as possible. Considering the considerable size of these tasks, contributions (especially Faust implementations of common DSP algorithms, and additional plugin architectures) are most welcome. Interested audio developers are invited to join the Faust community at `http://faust.grame.fr`.

References

[1] Y. Orlarey, D. Fober, and S. Letz. An algebra for block diagram languages. In *Proceedings of the International Computer Music Conference (ICMC 2002)*. International Computer Music Association, 2002.

[2] Y. Orlarey, D. Fober, and S. Letz. Syntactical and semantical aspects of Faust. *Soft Computing*, 8(9):623–632, 2004.

[3] Yann Orlarey, Albert Gräf, and Stefan Kersten. DSP programming with Faust, Q and SuperCollider. In *Proceedings of the 4th International Linux Audio Conference (LAC06)*, pages 39–47, Karlsruhe, 2006. ZKM.

[4] Yann Orlarey. Faust quick reference. Technical report, Grame, 2006.

[5] Albert Gräf. Q: A functional programming language for multimedia applications. In *Proceedings of the 3rd International Linux Audio Conference (LAC05)*, pages 21–28, Karlsruhe, 2005. ZKM.