Polyphony, sample-accurate control and MIDI support for Faust using combinable architecture files

Stéphane Letz
LAC 2017

Purpose

- **revisit** Faust architecture file model
- **share some techniques** we use to design and code more reusable components
- further **develop the Faust ecosystem**
- **get around** some (current…) limits of the compiler

How do people usually use Faust generated code ?

- **hack the generated code** ? (Guitarix) using external tools (Python…)
- **copy part of it** in DSP plugins ?
- this « could » possibly be done by **properly using** the proposed architecture model…

Agenda

- the Faust **architecture files** model
- explain how new files **to be combined** can be developed
- some examples: **polyphony, sample-accurate control, MIDI**
- **perspectives**: refine polyphonic class, extend the control/DSP model…

Combined DSP and control (1)

- described in a **same source file**
- **abstract UI** that can be later on connected to **standard UI** (buttons sliders..), **MIDI**, **OSC**, and **various sensors** (accelerometers…)

Combined DSP and control (2)

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

vol             = hslider("volume [unit:dB][midi:ctrl 7]", 0, -96, 0, 0.1) : ba.db2linear : si.smoo ;
freq            = hslider("freq [unit:Hz]", 1000, 20, 24000, 1);

process         = vgroup("Oscillator", os.osc(freq) * vol);
```

- « **hslider** » later on connected to **UI slider** or **OSC, MIDI**…
- **metadata** for finer control (alternate controller, units…)

Deployment

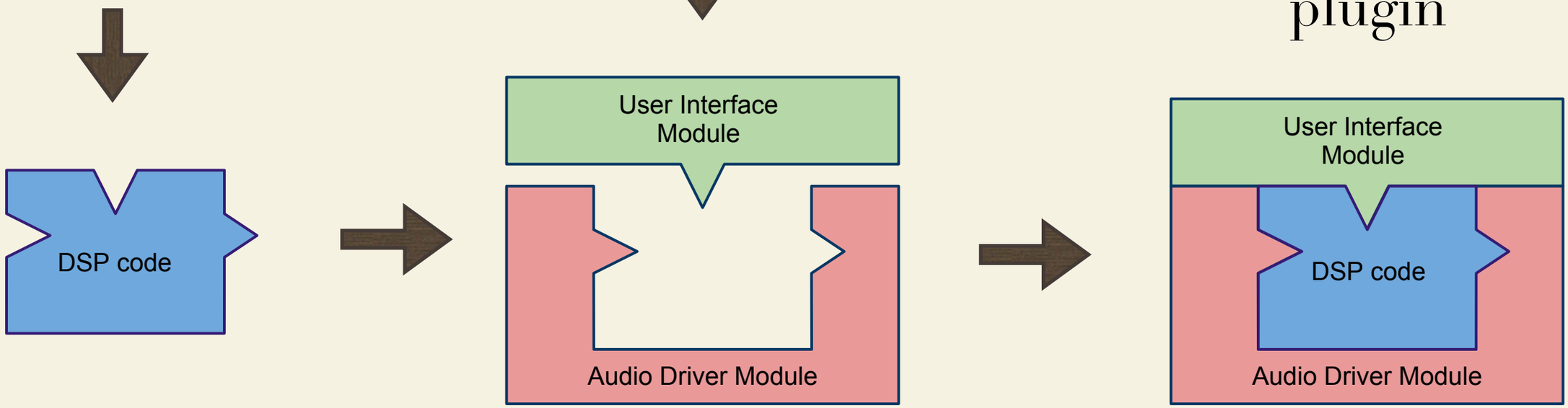
- the compiler generates the **DSP computation as a C++ class**: generating or transforming samples
- an **architecture file** connects the computation to **audio layer** and **control/UI interface**

Architecture model

Faust compiler

Architecture file

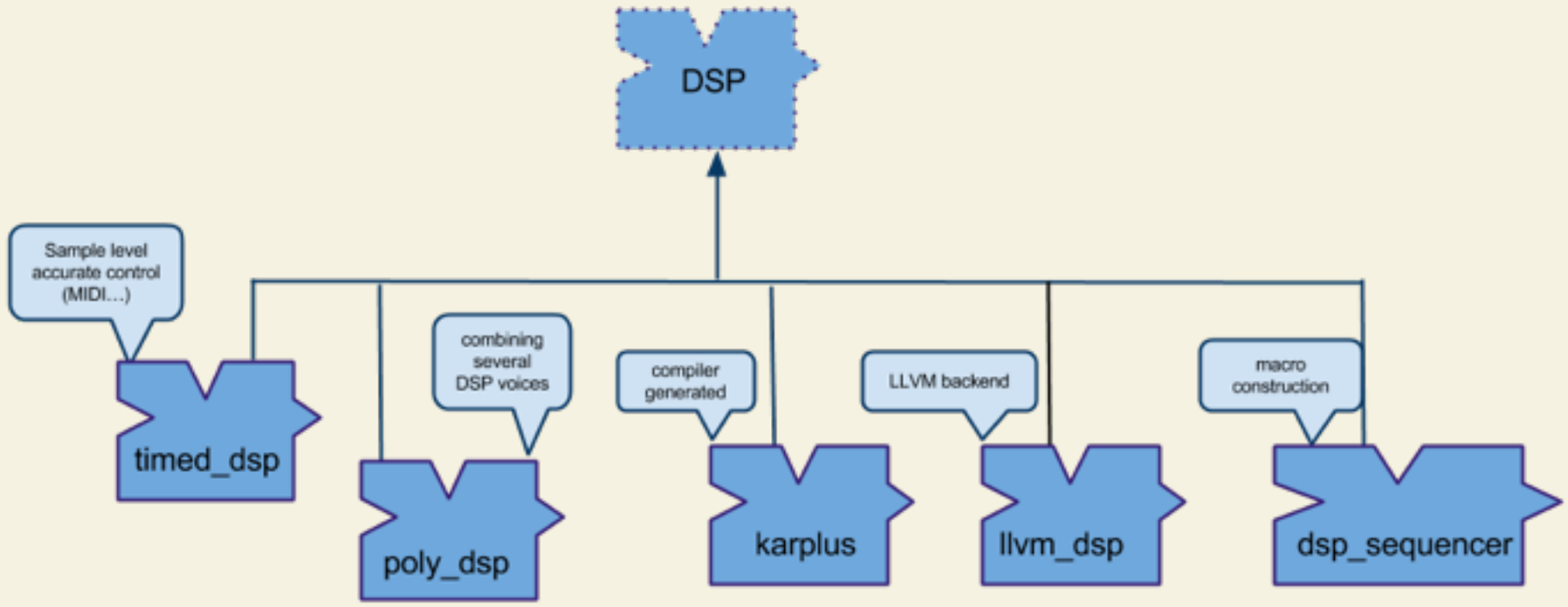
Application or plugin



DSP class definition

```
/**
* Signal processor definition.
*/

class dsp {

    public:

        dsp() {}
        virtual ~dsp() {}

        virtual int getNumInputs() = 0;

        virtual int getNumOutputs() = 0;

        virtual void buildUserInterface(UI* ui_interface) = 0;

        virtual int getSampleRate() = 0;

        virtual void init(int samplingRate) = 0;

        virtual void instanceInit(int samplingRate) = 0;

        virtual void instanceConstants(int samplingRate) = 0;

        virtual void instanceResetUserInterface() = 0;

        virtual void instanceClear() = 0;

        virtual dsp* clone() = 0;

        virtual void metadata(Meta* m) = 0;

        virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT** outputs) = 0;

        virtual void compute(double date_usec, int count, FAUSTFLOAT** inputs, FAUSTFLOAT** outputs)
        {
            compute(count, inputs, outputs);
        }

};
```

DSP classes hierarchy

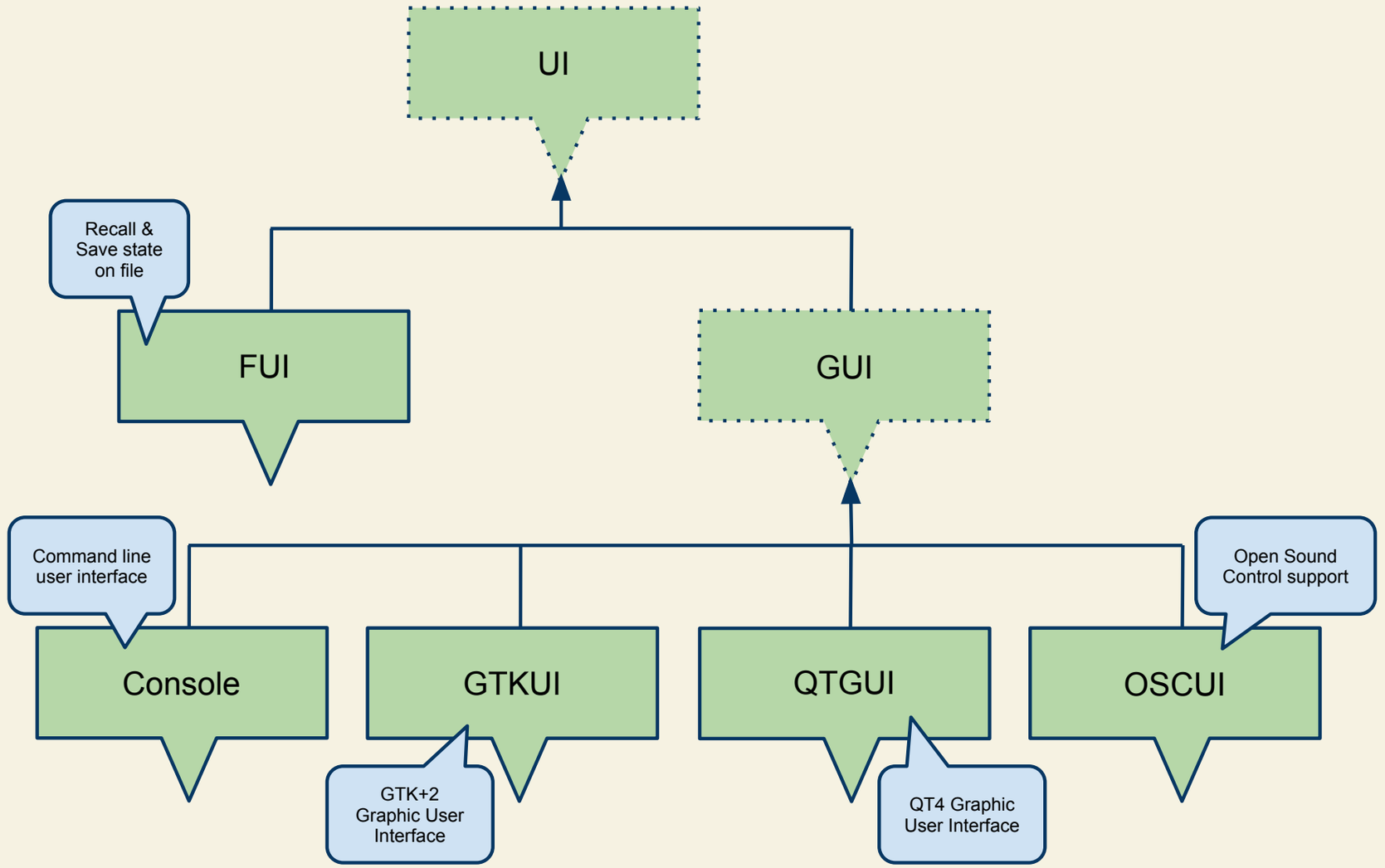


- compiler **statically generated** (like « C++ **karplus** »)
- compiler **dynamically generated** (like LLVM **llvm_dsp**)
- « **decorator » like class:** poly_dsp, timed_dsp…
- **« macro construction »: dsp_sequencer, dsp_parallelizer…**

UI class definition

```
/*******************************************************************************
 * UI : Faust User Interface
 * This abstract class contains only the method that the faust compiler can
 * generate to describe a DSP interface.
 ******************************************************************************/

class UI
{

    public:

        UI() {}

        virtual ~UI() {}

        // -- widget's layouts

        virtual void openTabBox(const char* label) = 0;
        virtual void openHorizontalBox(const char* label) = 0;
        virtual void openVerticalBox(const char* label) = 0;
        virtual void closeBox() = 0;

        // -- active widgets

        virtual void addButton(const char* label, FAUSTFLOAT* zone) = 0;
        virtual void addCheckButton(const char* label, FAUSTFLOAT* zone) = 0;
        virtual void addVerticalSlider(const char* label, FAUSTFLOAT* zone, FAUSTFLOAT init, FAUSTFLOAT min, FAUSTFLOAT max, FAUSTFLOAT step) = 0;
        virtual void addHorizontalSlider(const char* label, FAUSTFLOAT* zone, FAUSTFLOAT init, FAUSTFLOAT min, FAUSTFLOAT max, FAUSTFLOAT step) = 0;
        virtual void addNumEntry(const char* label, FAUSTFLOAT* zone, FAUSTFLOAT init, FAUSTFLOAT min, FAUSTFLOAT max, FAUSTFLOAT step) = 0;

        // -- passive widgets

        virtual void addHorizontalBargraph(const char* label, FAUSTFLOAT* zone, FAUSTFLOAT min, FAUSTFLOAT max) = 0;
        virtual void addVerticalBargraph(const char* label, FAUSTFLOAT* zone, FAUSTFLOAT min, FAUSTFLOAT max) = 0;

        // -- metadata declarations

        virtual void declare(FAUSTFLOAT*, const char*, const char*) {}
};
```

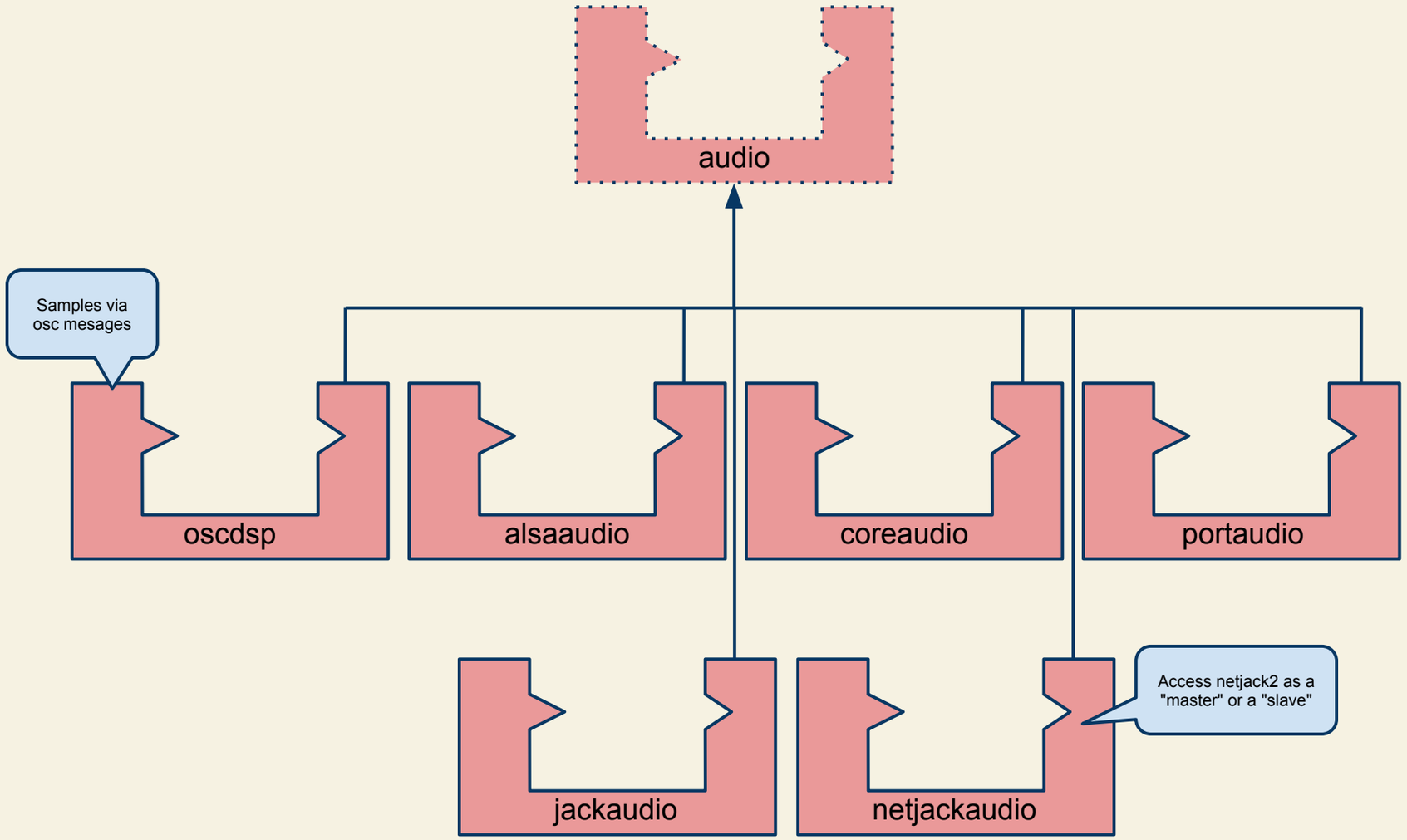
UI classes hierarchy



Audio class definition

```
typedef void (* shutdown_callback)(const char* message, void* arg);

class audio {

 public:
                audio() {}
    virtual ~audio() {}

    virtual bool init(const char* name, dsp*)                   = 0;
    virtual bool start()                                        = 0;
    virtual void stop()                                         = 0;
    virtual void shutdown(shutdown_callback cb, void* arg)  {}

    virtual int getBufferSize() = 0;
    virtual int getSampleRate() = 0;

    virtual int getNumInputs() = 0;
    virtual int getNumOutputs() = 0;

    virtual float getCPULoad() { return 0.f; }
};
```

Audio class hierarchy



« karplus » DSP source

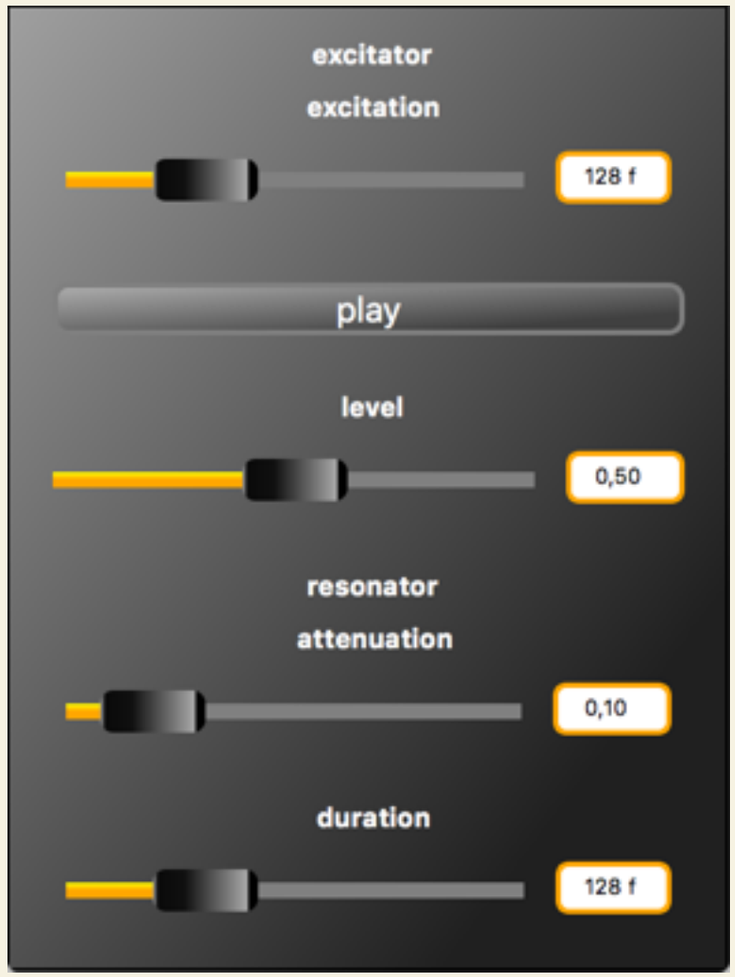
```
// Excitator
//-----------

upfront(x)  = (x-x') > 0.0;
decay(n,x)  = x - (x>0.0)/n;
release(n)  = + ~ decay(n);
trigger(n)  = upfront : release(n) : >(0.0);

size        = hslider("excitation [unit:f]", 128, 2, 512, 1);
```

```
// resonator
//------------

dur         = hslider("duration [unit:f]", 128, 2, 512, 1);
att         = hslider("attenuation", 0.1, 0, 1, 0.01);
average(x)  = (x+x')/2;

resonator(d, a) = (+ : de.delay(4096, d-1.5)) ~ (average : *(1.0-a)) ;

process = no.noise * hslider("level", 0.5, 0, 1, 0.01)
        : vgroup("excitator", *(button("play"): trigger(size)))
        : vgroup("resonator", resonator(dur, att));
```

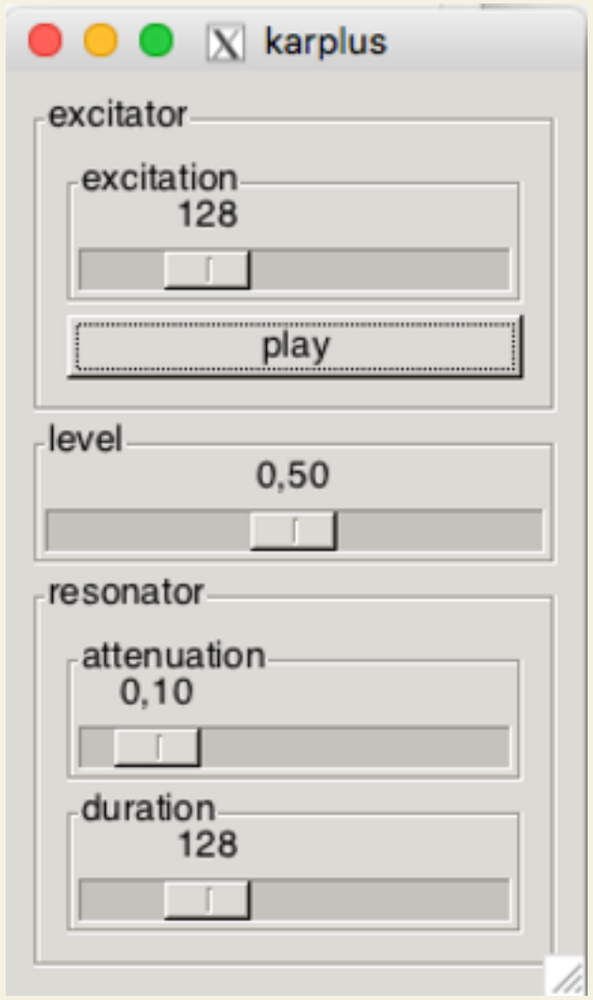
« karplus » class definition

```
virtual void buildUserInterface(UI* ui_interface) {
    ui_interface->openVerticalBox("0x00");
    ui_interface->openVerticalBox("excitator");
    ui_interface->declare(&fHslider2, "unit", "f");
    ui_interface->addHorizontalSlider("excitation", &fHslider2, 128.0f, 2.0f, 512.0f, 1.0f);
    ui_interface->addButton("play",&fButton0);
    ui_interface->closeBox();
    ui_interface->addHorizontalSlider("level", &fHslider1, 0.5f, 0.0f, 1.0f, 0.00999999978f);
    ui_interface->openVerticalBox("resonator");
    ui_interface->addHorizontalSlider("attenuation", &fHslider0, 0.100000001f, 0.0f, 1.0f, 0.00999999978f);
    ui_interface->declare(&fHslider3, "unit", "f");
    ui_interface->addHorizontalSlider("duration", &fHslider3, 128.0f, 2.0f, 512.0f, 1.0f);
    ui_interface->closeBox();
    ui_interface->closeBox();
}

virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT** outputs) {
    FAUSTFLOAT* output0 = outputs[0];
    float fSlow0 = (0.5f * (1.0f - float(fHslider0)));
    float fSlow1 = (4.65661287e-10f * float(fHslider1));
    float fSlow2 = float(fButton0);
    float fSlow3 = (1.0f / float(fHslider2));
    int iSlow4 = int(min(4096.0f, max(0.0f, (float(fHslider3) + -1.5f))));
    for (int i = 0; (i < count); i = (i + 1)) {
        iRec1[0] = ((1103515245 * iRec1[1]) + 12345);
        fVec0[0] = fSlow2;
        fRec2[0] = ((fRec2[1] + float(((fSlow2 - fVec0[1]) > 0.0f))) - (fSlow3 * float((fRec2[1] > 0.0f))));
        fVec1[(IOTA & 511)] = ((fSlow0 * (fRec0[1] + fRec0[2])) + (fSlow1 * float((iRec1[0] * (fRec2[0] > 0.0f)))));
        fRec0[0] = fVec1[((IOTA - iSlow4) & 511)];
        output0[i] = FAUSTFLOAT(fRec0[0]);
        iRec1[1] = iRec1[0];
        fVec0[1] = fVec0[0];
        fRec2[1] = fRec2[0];
        IOTA = (IOTA + 1);
        fRec0[2] = fRec0[1];
        fRec0[1] = fRec0[0];
    }
}
```

« karplus » generated GUI



faust2caqt: QT



faust2jack: GTK

Polyphonic support (1)

- **polyphonic support** : one voice is coded in Faust with the « **freq/gate/gain** » and possibly **declare nvoices « 8 »;** naming conventions
- work has already been done by Albert Graef in **LV2 and VST plugin** (quite sophisticated actually…)

Polyphonic support (1)

- the idea here is to recode this behavior a **more generic** and **reusable class**
- so define a **poly_dsp subclass of dsp**
- **voices allocation** and **note handing** done by the **poly_dsp** class
- then connection to **MIDI events** (or OSC...)

Polyphonic support (2)

- adapted **buildUserInterface** method:
  - all voices added in **a tab group**
  - one **master voice** to control them all in the first tab
  - a **panic** button

Polyphonic support (3)

```
void poly_dsp::buildUserInterface(UI* ui_interface)
{
    if (fVoiceTable.size() > 1) {
        ui_interface->openTabBox("Polyphonic");

        // Grouped voices UI
        ui_interface->openVerticalBox("Voices");
        ui_interface->addButton("Panic", &fPanic);
        fVoiceGroup->buildUserInterface(ui_interface);
        ui_interface->closeBox();

        // If not grouped, also add individual voices UI
        if (!fGroupControl) {
            for (int i = 0; i < fVoiceTable.size(); i++) {
                char buffer[32];
                snprintf(buffer, 31, ((fVoiceTable.size() < 8) ? "Voice%d" : "V%d"), i+1);
                ui_interface->openHorizontalBox(buffer);
                fVoiceTable[i]->buildUserInterface(ui_interface);
                ui_interface->closeBox();
            }
        }

        ui_interface->closeBox();
    } else {
        fVoiceTable[0]->buildUserInterface(ui_interface);
    }
}
```

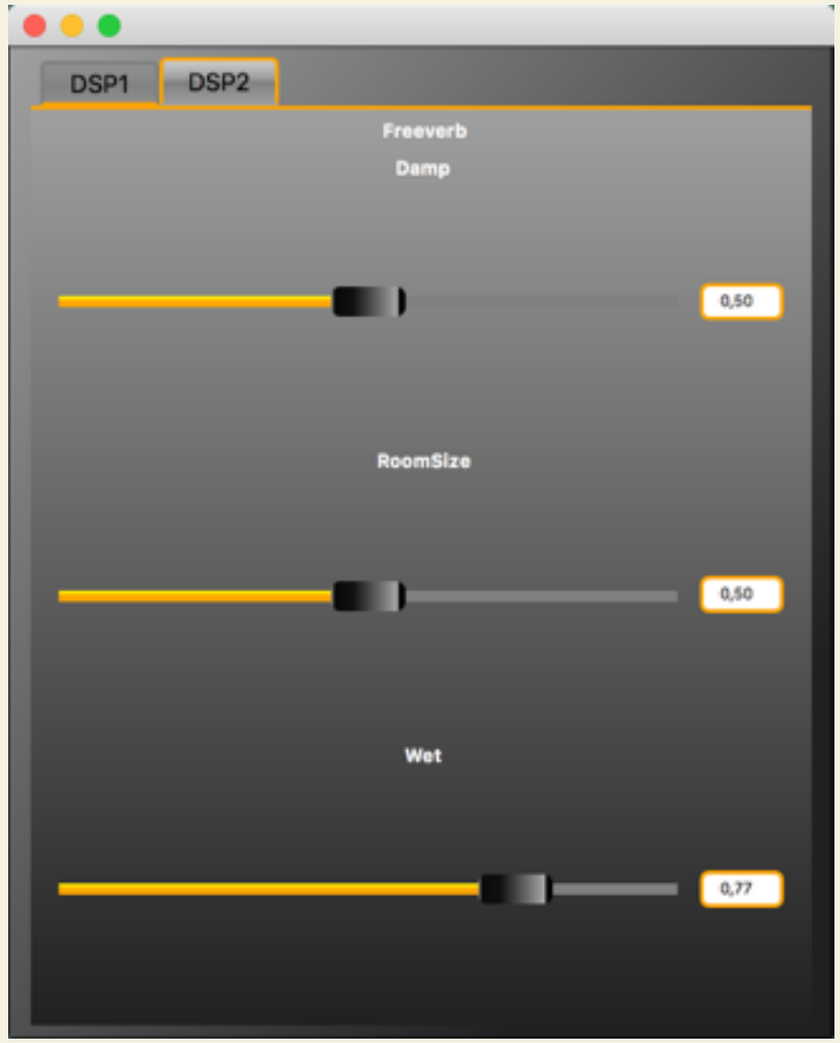
Polyphonic GUI

faust2caqt -midi -nvoices 16 karplus_synth.dsp

Polyphonic OSC paths

```
// Mono mode

/0x00/0x00/vol f -10.0
/0x00/0x00/pan f 0.0

// Polyphonic mode

/Polyphonic/Voices/0x00/0x00/pan f 0.0
/Polyphonic/Voices/0x00/0x00/vol f -10.0
...
/Polyphonic/Voice1/0x00/0x00/vol f -10.0
/Polyphonic/Voice1/0x00/0x00/pan f 0.0
...
/Polyphonic/Voice2/0x00/0x00/vol f -10.0
/Polyphonic/Voice2/0x00/0x00/pan f 0.0
...
```

Polyphonic + effect GUI



faust2caqt -midi -nvoices 16 -effect freeverb karplus.dsp

(using **dsp_sequencer** to connect the synth and the effect)

Sample-accurate control (1)

- control values are kept in **memory zones shared between DSP and UI**
- controllers **write on** this memory zones
- in compute method, control values are « **sampled** » **once per block** and **considered constant** during the block

Example of generated code

```
//-----------------------------------------------
//          Volume control in dB
//-----------------------------------------------

import("music.lib");

smooth(c)   = *(1-c) : +~*(c);
gain        = vslider("[1]", 0, -70, +4, 0.1) : db2linear : smooth(0.999);

process     = *(gain);
```

```
virtual void compute(int count, FAUSTFLOAT** inputs, FAUSTFLOAT** outputs) {
    FAUSTFLOAT* input0 = inputs[0];
    FAUSTFLOAT* output0 = outputs[0];
    float fSlow0 = (0.00100000005f * powf(10.0f, (0.0500000007f * float(fVslider0))));
    for (int i = 0; (i < count); i = (i + 1)) {
        fRec0[0] = (fSlow0 + (0.999000013f * fRec0[1]));
        output0[i] = FAUSTFLOAT((float(input0[i]) * fRec0[0]));
        fRec0[1] = fRec0[0];
    }
}
```

Sample-accurate control (2)

- the compute method sees the **last value of each control zone**
- some values **may be lost…**
- enough for most cases…

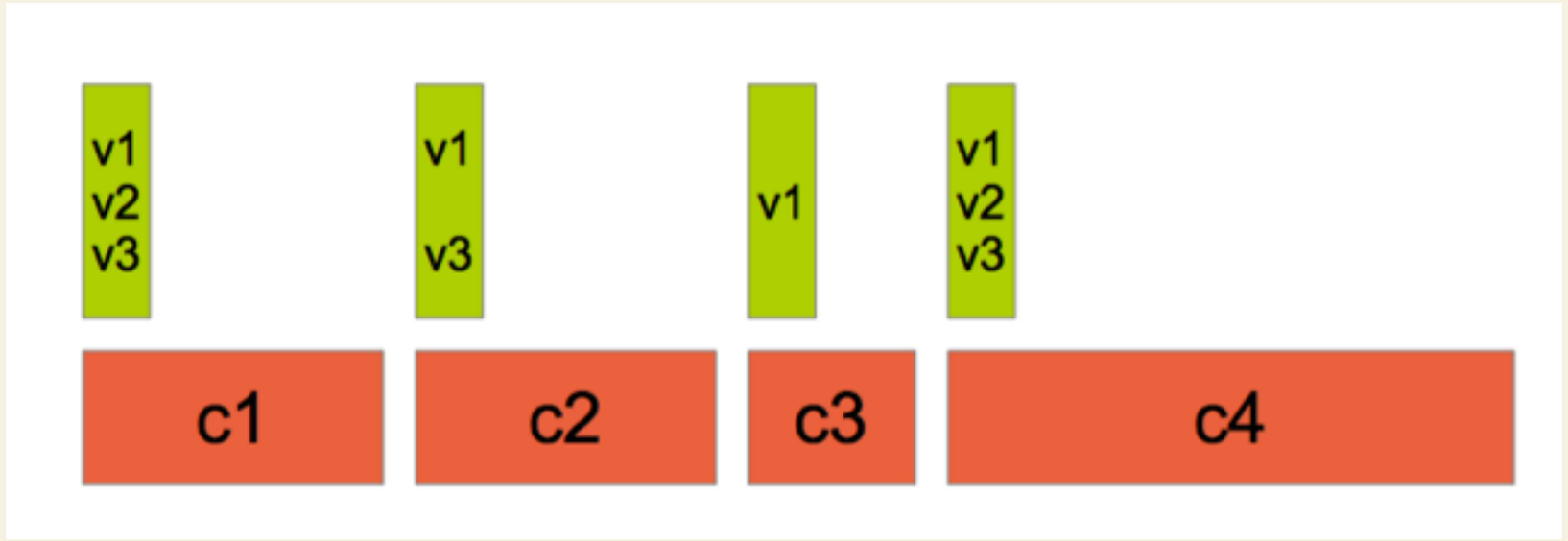
Sample-accurate control (3)

- but not always…
- for instance for **MIDI clock synchronization**
- all clock messages **have to be kept** at their **precise sample position**

Sample-accurate control (3)

- using the **timed_dsp** DSP decorator and **uiTimedItem** based GUI items
- when received, all control values are **time-stamped** and **kept in a queue**
- **timed_dsp** class decorates a given dsp to adapt its **compute** method

Sample-accurate control (4)



- « **slice-based** » call of the compute method
- example here : a complete buffer is **cut in 4 (c1…c4) slices**
- **control changes** between slices

MIDI control

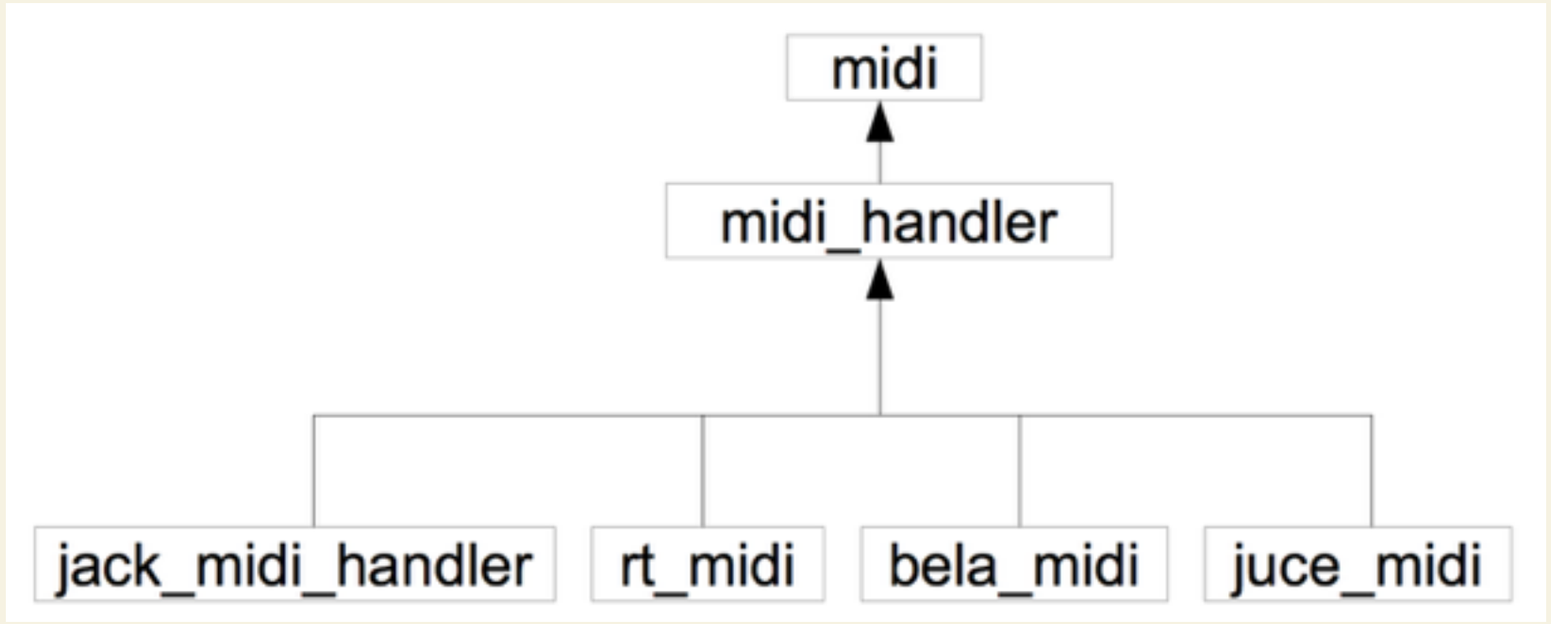
- metadata in UI elements:
  - [**midi:ctrl 7**], [**midi:pgm 12**]....
  - [**midi:start**]: triggers 1 when **start** is received
  - [**midi:stop**]: triggers 0 when **stop** is received
  - [**midi:clock**]: delivers a sequence of 1 and 0 each time a **clock** is received

MIDI clock sync example

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

// square signal (1/0), changing state
// at each received clock
clocker = checkbox("MIDI clock[midi:clock]");

// ON/OFF button controlled
// with MIDI start/stop messages
play = checkbox("On/Off [midi:start][midi:stop]");

// detect front
front(x) = (x-x') != 0.0;

// count number of peaks during one second
freq(x) = (x-x@ma.SR) : + ~ _;

process = os.osc(8*freq(front(clocker))) * play;
```

- MIDI clocks generate a **square control signal**
- converted as a **frequency**, to be used in a oscillator

MIDI classes



- hierarchy of MIDI event handling classes
- **RTMidi** for multi-platform support (Linux, OSX, Windows)
- **specialized support** (JACK, Bela, JUCE…)

Deployment

- **MIDI** and **polyphonic architecture** in:
  - **adapted faust2xx scripts** (**faust2caqt** -midi -nvoices 7 foo.dsp…)
  - in **FaustLive** and **faustgen**~ (so using dynamic LLVM generated llvm_dsp*)
  - in Web tools (asm.js and WebAssembly Faust pages)

Perspectives(1)

- Complete the **poly_dsp** class (MIDI channel support, more MIDI messages handling...)

Perspectives(2)

- Rework the control/DSP model
- having control as an **asynchronous signal:** queues of <time-stamp, value>
- would have to adapt the **DSP code generation** and **architecture model**...

Questions?

- Faust site: **http://faust.grame.fr**
- GitHub: **https://github.com/grame-cncm/faust**