FAUST ARCHITECTURES DESIGN AND OSC SUPPORT.

D. Fober, Y. Orlarey, S. Letz

Grame
Centre national de création musicale
Lyon, France
`{fober,orlarey,letz}@grame.fr`

ABSTRACT

FAUST [Functional Audio Stream] is a functional programming language specifically designed for real-time signal processing and synthesis. It consists in a compiler that translates a FAUST program into an equivalent C++ program, taking care of generating the most efficient code. The FAUST environment also includes various *architecture* files, providing the glue between the FAUST C++ output and the host audio and GUI environments. The combination of architecture files and FAUST output gives ready to run applications or plugins for various systems, which makes a single FAUST specification available on different platforms and environments without additional cost. This article presents the overall design of the architecture files and gives more details on the recent OSC architecture.

1. INTRODUCTION

From a technical point of view FAUST [1] (*Functional Audio Stream*) is a functional, synchronous, domain specific language designed for real-time signal processing and synthesis. A unique feature of Faust, compared to other existing languages like Max, PD, Supercollider, etc., is that programs are not interpreted, but fully compiled.

One can think of FAUST as a *specification language*. It aims at providing the user with an adequate notation to describe *signal processors* from a mathematical point of view. This specification is free, as much as possible, from implementation details. It is the role of the FAUST compiler to provide automatically the best possible implementation. The compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code. The compiler offers various options to control the generated code, including options to do fully automatic parallelization and take advantage of multicore machines.

The generated code can generally compete with, and sometimes even outperform, C++ code written by seasoned programmers. It works at the sample level, it is therefore suited to implement low-level DSP functions like recursive filters up to full-scale audio applications. It can be easily embedded as it is self-contained and doesn't depend of any DSP library or runtime system. Moreover it has a very deterministic behavior and a constant memory footprint.

From a syntactic point of view FAUST is a textual language, but nevertheless block-diagram oriented. It actually combines two approaches: *functional programming* and *algebraic block-diagrams*. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a *block-diagram algebra* of five composition operations (`: , ~ <: :>`) [1, 2].

We don't have the space to describe the language in details but as an example here is how to write a pseudo random number generator r in Faust [2] :

```
r = +(12345)~*(1103515245);
```

This example uses the recursive composition operator ~ to create a feedback loop as illustrated figure 1.

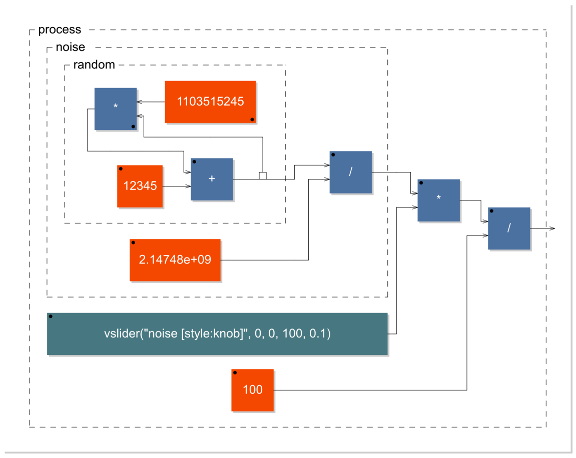

Figure 1: Block-diagram of a noise generator. This image is produced by the FAUST compiler using the `-svg` option.

Being a specification language the FAUST code says nothing about the audio drivers or the GUI toolkit to be used. It is the role of the architecture file to describe how to relate the dsp code to the external world. This approach allows a single FAUST program to be easily deployed to a large variety of audio standards (Max-MSP externals, PD externals, VST plugins, CoreAudio applications, Jack applications, etc.). In the following sections we will detail this architecture mechanism and in particular the recently developed OSC architecture that allows FAUST programs to be controlled by OSC messages.

2. FAUST SIGNAL PROCESSORS

A FAUST program denotes a signal processor implemented as an instance of a `dsp` class, defined as follows:

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

[2] Please note that this expression produces a signal $r(t) = 12345 + 1103515245 * r(t-1)$ that exploits the particularity of 32-bits integer operations

```
class dsp {
 public:
           dsp() {}
  virtual ~dsp() {}
  virtual int getNumInputs()              = 0;
  virtual int getNumOutputs()             = 0;
  virtual void buildUserInterface(UI* ui)  = 0;
  virtual void init(int samplingRate)      = 0;
  virtual void compute(int len, float** in,
                              float** out)= 0;
};
```

The `dsp` object is central to the FAUST architectures design:

- `buildUserInterface` creates the user interface,
- `compute` is called by the audio architecture for the signal processing,
- `getNumInputs, getNumOutputs` provides information about the signal processor,
- `init` is called to initialize the sampling rate, which is typically done by the audio architecture.

3. AUDIO ARCHITECTURE FILES

A FAUST audio architecture is a glue between the host audio system and a FAUST module. It is responsible to allocate and release the audio channels and to call the FAUST `dsp::compute` method to handle incoming audio buffers and/or to produce audio output. It is also responsible to present the audio as non-interleaved float data, normalized between -1. and 1.

A FAUST audio architecture derives an *audio* class defined as below:

```
class audio {
 public:
           audio() {}
  virtual ~audio() {}
  virtual bool init(const char* name, dsp*) = 0;
  virtual bool start()                      = 0;
  virtual void stop()                       = 0;
};
```

The API is simple enough to give a great flexibility to audio architectures implementations. The `init` method should initialize the audio. At `init` exit, the system should be in a safe state to recall the `dsp` object state.

Table 4 gives the audio architectures currently available for various operating systems.

4. GUI ARCHITECTURE FILES

A FAUST UI architecture is a glue between a host control layer and a FAUST module. It is responsible to associate a FAUST module parameter to a user interface element and to update the parameter value according to the user actions. This association is triggered by the `dsp::buildUserInterface` call, where the `dsp` asks a UI object to build the module controllers.

Since the interface is basically graphic oriented, the main concepts are *widget* based: a UI architecture is semantically oriented to handle active widgets, passive widgets and widgets layout.

A FAUST UI architecture derives an *UI* class (defined in appendix 10.1).

Audio system	Operating system
Alsa	Linux
Core audio	Mac OS X, iOS
Jack	Linux, Mac OS X, Windows
Portaudio	Linux, Mac OS X, Windows
OSC (see section 5.2)	Linux, Mac OS X, Windows
VST	Mac OS X, Windows
Max/MSP	Mac OS X, Windows
CSound	Linux, Mac OS X, Windows
SuperCollider	Linux, Mac OS X, Windows
PureData	Linux, Mac OS X, Windows
Pure[3]	Linux, Mac OS X, Windows

Table 1: FAUST audio architectures

4.1. Active widgets

Active widgets are graphical elements that control a parameter value. They are initialized with the widget name and a pointer to the linked value. The widget currently considered are `Button`, `ToggleButton`, `CheckButton`, `VerticalSlider`, `HorizontalSlider` and `NumEntry`.
A GUI architecture must implement a method
`addxxx (const char* name, float** zone, ...)`
for each active widget. Additional parameters are available to Slider and NumEntry: the init value, the min and max values and the step.

4.2. Passive widgets

Passive widgets are graphical elements that reflect values. Similarly to active widgets, they are initialized with the widget name and a pointer to the linked value. The widget currently considered are `NumDisplay`, `TextDisplay`, `HorizontalBarGraph` and `VerticalBarGraph`.
A UI architecture must implement a method
`addxxx (const char* name, float** zone, ...)`
for each passive widget. Additional parameters are available, depending on the passive widget type.

4.3. Widgets layout

Generally, a GUI is hierarchically organized into boxes and/or tab boxes. A UI architecture must support the following methods to setup this hierarchy :

```
openTabBox (const char* label)
openHorizontalBox (const char* label)
openVerticalBox (const char* label)
closeBox (const char* label)
```

Note that all the widgets are added to the current box.

4.4. Metadata

The FAUST language allows widget labels to contain metadata enclosed in square brackets. These metadata are handled at GUI level by a `declare` method taking as argument, a pointer to the widget associated value, the metadata key and value:

```
declare(float*, const char*, const char*)
```

Table 2 gives the UI architectures currently available.

UI	Comment
console	a command line UI
GTK	a GTK based GUI
Qt	a multi-platform Qt based GUI
FUI	a file based UI to store and recall modules states
OSC	see section 5.1

Table 2: FAUST UI architectures

5. OSC ARCHITECTURES

The OSC support opens the FAUST applications control to any OSC capable application or programming language. But it also transforms a full range of devices embedding sensors (wiimote, smart phones...) into physical interfaces for FAUST applications control, allowing a direct use as music instrument (which is in phase with the new FAUST physical models library adapted [4] from STK [5]).

The FAUST OSC architecture provides an UI architecture but also an audio architecture. This audio architecture runs at the OSC data stream rate, meaning that it allows to slow the audio computation down, up to frame by frame computation, and thus proposing a new and original way to make digital signal computation.

5.1. OSC GUI architecture

The OSC UI architecture transforms all the UI active widgets additions into an `addnode` call, ignores the passive widgets and transforms containers calls (`openxxxBox, closeBox`) into `opengroup` and `closegroup` calls.

5.1.1. OSC address space and messages

The OSC address space adheres strictly to the hierarchy defined by the `addnode` and `opengroup, closegroup` calls. It supports the OSC pattern matching mechanism.

A node expects to receive OSC messages with a single float value as parameter. This policy is strict for the parameters count, but relaxed for the parameter type: OSC int values are accepted and cast to float.

Two additional messages are defined to provide FAUST applications discovery and address space discoveries:

- the `hello` message: accepted by any module root address. The module responds with its root address, followed by its IP address, followed by the UDP ports numbers (listening port, output port, error port). See the network management section below for ports numbering scheme.
- the `get` message: accepted by any valid OSC address. The `get` message is propagated to every terminal node that responds with its OSC address and current values (value, min and max).

Example:
Consider the *noise* module provided with the FAUST examples:

- it sends `/noise 192.168.0.1 5510 5511 5512` in answer to a `hello` message,
- it sends `/noise/Volume 0.8 0.  1.` in answer to a `get` message.

5.1.2. Network management

The OSC architecture makes use of 3 different UDP port numbers:

- 5510 is the listening port number: control messages should be addressed to this port.
- 5511 is the output port number: answers to query messages are send to this port.
- 5512 is the error port number: used for asynchronous errors notifications.

When the UDP listening port number is busy (for instance in case of multiple FAUST modules running), the system automatically looks for the next available port number. Unless otherwise specified by the command line, the UDP output port numbers are unchanged.

A module sends its name (actually its root address) and allocated ports numbers on the OSC output port on startup.

Ports numbers can be changed on the command line with the following options:

```
[-port | -outport | -errport] number
```

The default UDP output streams destination is `localhost`. It can also be changed with the command line option

`-dest address` where address is a host name or an IP number.

5.2. OSC audio architecture

The OSC audio architecture provides audio input and output using OSC messages. It is not intended for real-time audio transportation due to the overhead introduced by the OSC coding. But, as we will explain, it provides a very useful and powerful mean to analyze and/or debug the behaviour of a Faust application.

Using this architecture, a FAUST module accepts arbitrary data streams on its root OSC address, and handles this stream as input interleaved signals. Each incoming OSC packet addressed to a module root triggers a computation cycle, where as much values as the number of incoming frames are computed.

The output of the signal computation is sent to the OSC output port as non-interleaved data to the OSC addresses `/root/n` where `root` is the module root address and `n` is the output number (indexed from 0). For example, consider a simple FAUST program named *split* and defined by:

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

expected and generated OSC datagrams are illustrated in figure 2.

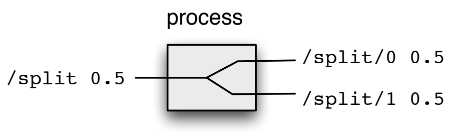

Figure 2: In and out OSC datagrams for the *split* module.

The OSC audio architecture provides a very convenient way to execute a signal processing at an arbitrary rate, allowing even to make step by step computation. Connecting the output OSC signals to Max/Msp or to a system like INScore[3], featuring a powerful dynamic signals representation system, provides a close examination of the computation results.

[3]http://inscore.sf.net

6. OPEN ISSUES AND FUTURE WORKS

Generally, the labeling scheme for a GUI doesn't result in an optimal OSC address space definition. Moreover, there are potential conflicts between the FAUST UI labels and the OSC address space since some characters are reserved for OSC pattern matching and thus forbidden in the OSC naming scheme. The latter issue is handled with automatic characters substitutions. The first issue could be solved using the metadata scheme and will be considered in a future release.

Another issue, resulting from the design flexibility, relies on dynamic aggregation of multiple architectures covering the same domain: for example, it would be useful to embed both a standard and the OSC audio architecture in the same module and to switch dynamically between (for debugging purpose for example). That would require the UI to include the corresponding control and thus a mechanism to permit the UI extension by the UI itself would be necessary.

7. CONCLUSIONS

FAUST is a mature language for the design of signal processors. The FAUST architectures give the developer a handful of various binary outputs without additional cost. The architectures design, made for easy extension and combination, fits very well in the landscape of the evolving technologies. Adaptation to new hardware or software is simple and opens the door to all the existing FAUST programs. The recent OSC addition makes the connection between FAUST modules and the more and more ubiquitous hardware embedding sensors and usable as gestural controllers. FAUST architectures add a practical and ready to run dimension to the powerful FAUST language and additional contributions are welcome.

8. ACKNOWLEDGMENTS

The OSC architecture has been designed in the joint context of the ASTREE project [ANR-08-CORD-003] and the *Parallelism in Interactive Real-Time Signal Processing with* FAUST *and OSC* project made in collaboration with the CNMAT and supported by the France-Berkeley fund.

9. REFERENCES

[1] Y. Orlarey, D. Fober, and S. Letz, "An algebra for block diagram languages," in *Proceedings of International Computer Music Conference*, ICMA, Ed., 2002, pp. 542–547.

[2] Y. Orlarey, D. Fober, and S. Letz, *New Computational Paradigms for Computer Music*, chapter FAUST : an Efficient Functional Approach to DSP Programming, pp. 65–96, Editions DELATOUR FRANCE, 2009.

[3] Albert Graef, "Signal processing in the pure programming language," in *Proceedings of the Linux Audio Conference LAC2009*, 2009.

[4] R. Michon and J. O. Smith, "Faust-stk: a set of linear and nonlinear physical models for the faust programming language.," in *submitted to DAFx 2011*, 2011.

[5] P. Cook, "The synthesis toolkit (stk)," in *Proceedings of the International Computer Music Conference (ICMC)*, Beijing, China, Oct., 1999, pp. 299–304.

10. APPENDIX

10.1. UI class

```
class UI
{
 public:
           UI() {}
  virtual ~UI() {}

  // -- active widgets
  virtual void addButton(const char* label, float* zone)                    = 0;
  virtual void addToggleButton(const char* label, float* zone)              = 0;
  virtual void addCheckButton(const char* label, float* zone)               = 0;
  virtual void addVerticalSlider(const char* label, float* zone,
                              float init, float min, float max, float step) = 0;
  virtual void addHorizontalSlider(const char* label, float* zone,
                              float init, float min, float max, float step) = 0;
  virtual void addNumEntry(const char* label, float* zone, float init,
                                      float min, float max, float step)     = 0;
  // -- passive widgets
  virtual void addNumDisplay(const char* label, float* zone, int precision) = 0;
  virtual void addTextDisplay(const char* label, float* zone,
                                 const char* names[], float min, float max) = 0;
  virtual void addHorizontalBargraph(const char* label, float* zone,
                                                      float min, float max) = 0;
  virtual void addVerticalBargraph(const char* label, float* zone,
                                                      float min, float max) = 0;
  // -- 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;

  // -- metadata declarations
  virtual void declare(float* , const char* , const char* ) {}
};
```

10.2. Available FAUST architectures

Audio system	Environment	OSC support
Linux		
Alsa	GTK, Qt	yes
Jack	GTK, Qt, Console	yes
PortAudio	GTK, Qt	yes
Mac OS X		
CoreAudio	Qt	yes
Jack	Qt, Console	yes
PortAudio	Qt	yes
Windows		
Jack	Qt, Console	yes
PortAudio	Qt	yes
iOS (iPhone)		
CoreAudio	Cocoa	not yet

Table 3: FAUST applications architectures

Name	System
ladspa	LADSPA plugins
csound	CSOUND opcodes
csounddouble	double precision CSOUND opcodes
maxmsp	Max/MSP externals
vst	native VST plugins
w32vst	windows VST plugins
supercollider	Supercollider plugins
puredata	Puredata externals
Q	Q plugins
Pure	Pure plugins

Table 4: FAUST plugins architectures