Faust audio DSP language for JUCE

Adrien Albouy, Stéphane Letz

Faust audio DSP language for JUCE

Adrien ALBOUY and **Stéphane Letz**
GRAME
11, cours de Verdun (GENSOUL)
69002 LYON,
FRANCE,
{adrien.albouy, letz}@grame.fr

Abstract

FAUST [Functional Audio Stream] is a functional programming language specifically designed for real-time signal processing and synthesis [1]. It consists of a compiler that translates a FAUST program into an equivalent C++ program, taking care of generating the most efficient code. JUCE is an open-source cross-platform C++ application framework developed since 2004, and bought by ROLI[1] in November 2014, used for the development of desktop and mobile applications. A new feature to the FAUST environnement is the addition of architectures files to provide the glue between the FAUST C++ output and the JUCE framework. This article presents the overall design of the architecture files for JUCE.

Keywords

JUCE, FAUST, Domain Specific Language, DSP, real-time, audio

1 Introduction

From a technical point of view FAUST[2] (*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 fullscale audio applications. It can be easily embedded as it is selfcontained and does not depend of any DSP library or runtime system. Moreover it has a very deterministic behavior and a constant memory footprint.

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 [2]. 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.), and JUCE is now supported.

The aim of JUCE[3] is to allow software to be written such that the same source code will compile and run identically on Windows, Mac OS X, Linux platforms for the desktop devices, and on Android and iOS for the mobile ones. A notable feature of JUCE when compared to other similar frameworks is its large set of audio functionality. Those services, the user-interface possibilities and the multi-platform exportability position JUCE as a great framework for FAUST to get exported on, to have in the future less code to maintain up-to-date, and simpler utilization.

In section 2, the idea and the use of the `GUI` architecture file will be introduced. In section 3, the JUCE `Component` hierarchy will be presented without going into many details. Section 4 is the main one, explaining in detail the graphical architecture file for JUCE. MIDI and OSC architecture files are introduced in Section 5. Section 6 will treat of the "glue" between JUCE audio layers and FAUST ones. Section 7

[1] `https://roli.com/`
[2] `http://faust.grame.fr`

presents the `faust2juce` script. Section 8 is a quick tutorial on how to use JUCE for FAUST.

2 FAUST 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 a `UI` class, containing active widgets, passive widgets, layout widgets, and metadata.

2.1 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`, `RadioButton`, `Menu`, `VerticalSlider`, `HorizontalSlider`, `Knob` and `NumEntry`.

A `UI` architecture must implement a method `addxxx (const char* name, float* zone, ...)` for each active widget. Additional parameters are available to `Slider`, `Knob`, `NumEntry`, `RadioButton` and `Menu`: the init value, the min and max values and the step (`RadioButton`, `Menu` and `Knob` being special kind of `Sliders`, cf. subsection 2.4, Metadata).

2.2 Passive widget

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`, `Led`, `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. (`NumDisplay` and `Led` are a special kind of `BarGraph`, cf. Subsection 2.4).

2.3 Widget layout

Generally, a `UI` 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.

2.4 Metadata

The FAUST language allows widget labels to contain metadata enclosed in square brackets. These metadata are handled at `UI` 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*)`. Metadata can also declare a DSP as polyphonic, with a line looking like `declare nvoices "8"` for 8 voices. This will always output a polyphonic DSP, either you use the polyphonic option of the compiler or not. This number of voices can be changed with the compiler (cf. Section 7).

For instance, if the program needs a `Slider` to be a `Knob`, those lines are written:

```
declare(&fVslider0, "style", "knob");
addVerticalSlider("Vol", &fVslider0,...);
```

The `style` can be a `knob`, `menu`, etc... depending on the program.

Multiple aspects of the items can be described with the metadata, such as the type of the item just as seen before, the tooltip of the item, the unit, etc...

3 JUCE Component class

To implement a complete program, the graphical elements described in the previous section need to be combined with JUCE classes. In the JUCE Framework, the component class is the base-class for all JUCE user-interface objects. The following section explains the relationship between FAUST GUI architecture files, and the JUCE mechanics.

3.1 Parent and child mechanics

As most frameworks have, JUCE has a hierarchy of `Component` objects, organized in a tree structure. The common way to set a `Component` as child of another component is to do `parent->addAndMakeVisible(child);`.

This function sets the child component as visible too, because it's not by default. Multiple functionalities are accessible to run through this `Component` tree, with methods that give the child `Component` at index `i`, or give the parent. There's even a function allowing to get the parent of a `Component` with a specific type, this type being a derived class of `Juce::Component`. However, this function does not exist for the child, and imply that `dynamic_cast` has to be done if you want to get a child of a certain type.

3.2 Component setup mechanics

First of all, a `Component` is drawn if it's visible, and its parent too. If a `Component` is not visible, its child and all of its children, etc... will not be visible, but as `addAndMakeVisible` function is used most of the time, this should not be a problem. A `Component` has a `Rectangle<int> boundsRelativeToParent`, containing its x and y coordinates, and its width and height. As the variable name implies, the bounds of a `Component` is relative to its parent, and not absolute in the window ; it is very important in the architecture files for FAUST, as will be demonstrated in subsection 4.4.

3.3 Drawing mechanics

A `Component` has two virtual functions[3] that are the main tools to handle a dynamic layout, the `void resized()` and `void paint(Graphics& g)` functions. The `resized` one is called each time a `Component` bounds are changed, and the `paint` one when the `Component` flag indicates that it needs to be repainted. The mouse cursor being on top of it, a mouse click, the `Component` bounds being changed, or one or multiple of its child needing to be repainted indicates that it needs to be repainted for example.

There is a design class called `LookAndFeel` that allows customization of the interface. The `LookAndFeel` objects defines the appearance of all the JUCE widgets, and subclasses can be used to apply different 'skins' to the application.

There is obviously a lot more to the `Juce::Component` class, but that's the basics, or at least what the architecture files need.

4 JuceGUI architecture file

To summarize what has been seen before, the system of widgets and boxes of FAUST needs to be adapted to the `Juce::Component` mechanics in an architecture file called JuceGUI.h. The following section discusses annotated examples.

4.1 Two different kinds of objects

There are two kinds of object used in the adaptation:

- `uiComponent`, which are basically any items of the FAUST program, like sliders or buttons.
- `uiBox`, which is container component, and so can contain a `uiComponent` or some others `uiBox`.

Both are derived classes of a `uiBaseComponent` class, which is itself a derived class of `Juce::Component`.

The `uiBaseComponent` class regroups methods shared by both `uiBox` and `uiComponent`, like `void setRatio()`, `int getTotalWidth()`, etc.... This way, too many `dynamic_cast` in our code are avoided. Here's what the `uiBaseComponent` class contains:

```
float fHRatio, fVRatio;
int fTotalWidth, fTotalHeight;
int fDisplayRectHeight,
    fDisplayRectWidth;
String fName;

uiBaseComponent(int totWidth,
          int totHeight, String name);

int getTotalHeight() ;
int getTotalWidth();
virtual void setRatio();
float getHRatio();
float getVRatio();
String getName();
void setHRatio();
void setVRatio()
void setBaseComponentSize
                (Rectangle<int> r);
void mouseDoubleClick
  (const MouseEvent &event) override;

virtual void writeDebug() = 0;
virtual void setCompLookAndFeel
  (LookAndFeel* laf) = 0;
```

The `mouseDoubleClick` function is a JUCE overridable function, which is called every time a `Component` is double-clicked. Here it's used

[3] placeholder functions which programmer must implement

to call the `writeDebug` function, showing different characteristics of the double clicked `uiBox` or `uiComponent`.

The two pure virtual functions are defined to have their own behavior for both `uiBox` and `uiComponent`, not being the same obviously.

The `virtual void setRatio();` function is virtual because there is a special case with the `uiBox`, which is setting her own ratio, and need to be asking its child to set their ratios too, in a recursive way.

As said before, `uiComponent` inherits from those `uiBaseComponent` functions, and is itself a mother class for plenty of different widgets. Here's the inheritance diagram:

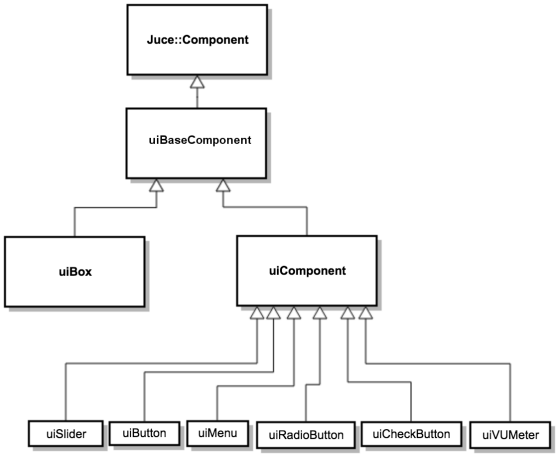

Figure 1: Inheritance diagram

A `uiComponent` subclasses can handle multiple "type" of items.

For instance, `uiSlider` groups every kind of sliders: `HorizontalSlider`, `VerticalSlider`, `NumEntry` and `Knob`.

4.2 The main window

The user interface cannot be shrunk infinitely in order to be always lisible and clear, so a minimal window size is defined. That implies that instead of a basic `Component` in a `DocumentWindow` (a resizable window with a title bar and maximise, minimise and close buttons), a `Viewport` in a `DocumentWindow` is used, which displays scrollbars when the window gets lower dimensions than the minimal size of the FAUST DSP program, allowing to have full access to the user interface even in the lower dimensions.

This `Viewport` can either contains a `uiBox` as presented before, or a `uiTabs` if the program requires tabs.

4.3 uiTabs class

The `uiTabs` class inherits of `Juce::TabbedComponent`, which is a `Juce::Component` with a `TabbedButtonBar` on one of its size. It just needs a `Juce::Component` for each tab, and a tab name, and it will display them.

A tab layout is needed when the `buildUserInterface` starts with a `openTabBox` call. In this, a boolean `tabLayout` is set to true, to know that it's a tab layout.

While parsing the `buildUserInterface`, a `uiBox` is given to the `uiTabs` every time the current tab is "closed". To do that, a variable called `order` keeps track of the "level" of the current box. The order starts at 0, is incremented when a new box is opened, and decremented when a box is closed. If the order is 0 in a `closeBox()` call, then a tab is being closed, and so the current box is added to the `uiTabs`, using the `TabbedComponent::addTab` function.

Once all the tabs are closed, the `tabBox` is closed too, the `order` is now at -1, and it triggers the initialization function of `uiTabs`, `uiTabs::init()`. It'll be described it in the next subsection.

4.4 Initialization of the layout

First of all, while parsing the `buildUserInterface` lines, which are listing the different boxes and items that need to be displayed, the tree is getting built. It's done using the `Juce::Component` mechanics of `addAndMakeVisible`. The different `uiBaseComponent` are added as child of different `uiBox`, and `uiBox` display rectangle size and total size are calculated every time a box is closed in the `buildUserInterface` (i.e. when `closeBox()` is called).

The `uiBox` ***display rectangle*** size is the sum of his child width and the maximum of his child height, and the contrary depending on its orientation. But margins are added to our display rectangle width and height, 4 pixels per child, for a margin of 2 pixels on the top, left, bottom and right, and the `uiBox` ***total size*** is obtained. This is to avoid an overlapping effect, having two items touching each other. Following the same spirit, 12 pixels are added to the height of the box if its name needs to be displayed, 12 pixels being the space needed to display its name.

Here's the `buildUserInterface` that display this program:

```
ui_interface->openHorizontalBox("TITLE1");
```

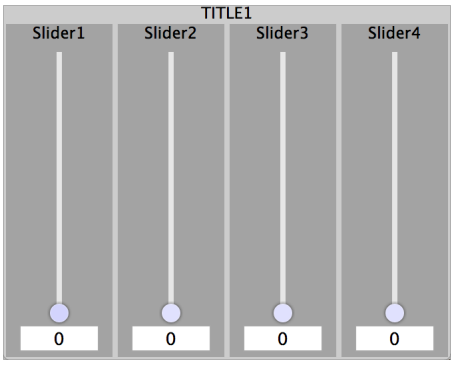

Figure 2: Representation of the display rectangle size and the total size of a box with four child

```
ui_interface->addVerticalSlider("Slider1",
      &fVslider0, 0.0f, 0.0f, 6.0f, 1.0f);
ui_interface->addVerticalSlider("Slider2",
      &fVslider1, 0.0f, 0.0f, 6.0f, 1.0f);
ui_interface->addVerticalSlider("Slider3",
      &fVslider2, 0.0f, 0.0f, 6.0f, 1.0f);
ui_interface->addVerticalSlider("Slider4",
      &fVslider3, 0.0f, 0.0f, 6.0f, 1.0f);
ui_interface->closeBox();
```

In Figure 2, the difference between the ***display rectangle size*** and the ***total size*** can be easily seen. The ***total size*** of the box here named "TITLE1" is the lighter gray, and the ***display rectangle size*** would be the four darker gray rectangle stick together. The layout is not aligned seamlessly because of the margin that is implemented to avoid the overlapping of the components.

The space left on the top of the box is for its title, and this margin is included in the ***total size***.

$$h = \sum_{i=0}^{n-1} (c_i.H) \tag{1}$$

$$w = \max_{i \in [0,n-1]} c_i.W \tag{2}$$

$$H = h + 4 * n \tag{3}$$

$$W = w + 4 \tag{4}$$

In those equations, H is the ***total height***, W the ***total width***, h the ***display rectangle height***, and w ***display rectangle width*** ; c_i being the nth child component of the current box.

H might get incremented by 12 pixels, depending on the need to display the box name.

4 pixels for each child component are added on a dimension to have margins between each of them, because they will be placed aside of each other in this dimension, and simply 4 pixels added to the other dimension to have 2 pixels separating parent and child box on each side.

Once `buildUserInterface` is done, the last box is closed, and the user interface initialized. This last box, that will be called the "main box" is initiated with ratios of 1 and 1, even if they are needed, because it'll take the window size. Here's how the UI is initialized:

- Setting the actual rendering size for the main box, because the total size is set here, but not the `Juce::Component` bounds. That's done through the `void setBaseComponentSize (Rectangle<int> r)` methods, which sets the size of the components, and especially position them right. Concretely, a 30 pixels offset is needed on the height for a tab layout, 30 pixels being the height taken by the tab bar. Only the main box needs to be set with an offset, because other boxes will be positioned depending on its parents coordinates.
- After that, the ratios are calculated for the whole tree, from root to leaves. The horizontal ratio is the component total width divided by its parent display rectangle width, same for the height. This way, it avoids to have the margins to mess with our ratios, and to have a sum of ratio equals to 1 instead of one approaching 1, but not being 1 exactly.
- Last step is to set the `LookAndFeel` for all `uiComponents`, which are for all of them the leaves of the trees. So the tree is fully parsed there, root to leaves.

The only possible change in the initialization of the program, is in a case of a tab layout. The `uiTabs::init()` method just calls the `uiBox::setRatio()` and the `uiBox::setCompLookAndFeel(LookAndFeel*)` for every of its tab component.

While going through all the tabs, the algorithm keeps track of the minimal size of the `uiTabs` component to be displayed. Its minimal dimensions being the maximum width and the maximum height of all its tabs.

There, the tree is built, the total size has been initiated, display rectangle size and the ratios for all components, all the `uiBox` and `uiComponent`.

4.5 Dynamic Layout

At that point, the user interface is displayed at his original size, but it needs to adapt to the potential resizing of the window. To do that, the `uiBoxes` are used to layout all the items. A `uiBox` item has a `void arrangeComponents(Rectangle<int> functionRect)` function, which is the main tool to organize the layout. It's called whenever the `resized()` function of the main box is called.

In this function, the initial rectangle given as argument, that is basically the window size, will propagate through all the child `uiBox` and `uiComponent`, in a recursive way [4].

At the beginning, it checks if the name needs to be displayed, and as no child components should be displayed there, it cuts 12 pixels from the top of the `functionRect`, given as argument.

After that, the margins are sets, so 2 pixels are cut on the left, top, right and bottom side. This way, overlapping components are avoided. Once it's done, it goes through all the child, to give them the right space to occupy and the right position of course.

The algorithm works that way: if the current box is vertical, then it needs to give its child a vertical part `functionRect`, and a horizontal one for a horizontal box of course.The amount of vertical or horizontal size of the child is calculated, still depending on the vertical nature of the current box. This size is the box current height or width, minus the margins, multiplied by the horizontal or vertical ratio. Concrete example: the current box is a horizontal display, and has 2 child components, one having a horizontal ratio of 0.7 and the other one of 0.3. The box display size is here 1000x500 pixels, and it's total size 1008x504 (2 items and it's a horizontal box, so $2 * (2 * margin) = 8$ on the width, and $2 * margin = 4$ on the height).

Let's say the size of the window almost doubled, and it's now 2008x1004 (arbitrary simple values). It will calculate that the first item get a $0.7*(2008-2*4) = 0.7*2000 = 1400$ pixels wide space and the second one $0.3*(2008-2*4) = 600$ pixels. First item bounds will be 1400x1000 and the second one 600x1000, height being kept the same, without the margins of course.

On top of that, to keep track of where to place our components, the `functionRect` get cut off little by little every time a `uiBaseComponent` is given a rectangle to be displayed in [5]. Basically, every rectangle that is given to child is removed from the original `functionRect`, and this allow us the keep track of the good `x` and `y` coordinates to give to the child component, with the margin added. It's done over and over again for each child component, cutting from the left or the top of the box`Rectangle<int> rect` depending on its orientation.

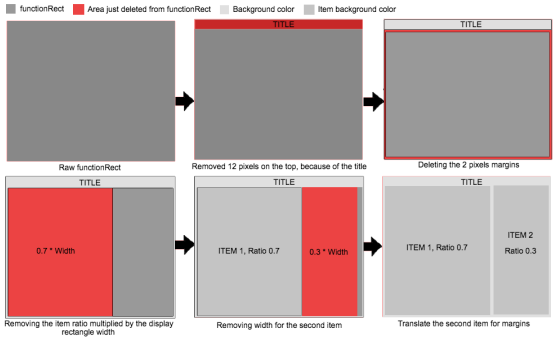

Figure 3: Representation of the layout algorithm

4.6 The MainContentComponent class

In the adapted `MainContentComponent` class, there is plenty of FAUST libraries, that are indispensable for the FAUST program. There are some optionals includes, for OSC, MIDI and polyphonic mode, that depends on the compilation options that the user sets.

The `MainContentComponent` class is the `Juce::Component` contained in our `Viewport`, and contains itself a `JuceGUI` object, that is a subclass of `Juce::Component`, FAUST `GUI` class and `MetaDataUI`. The minimal things to do is:

```
addAndMakeVisible(juceGUI);
fDSP = new mydsp();
fDSP->buildUserInterface(&juceGUI);
recommendedSize = juceGUI.getSize();
setSize (recommendedSize.getWidth(),
         recommendedSize.getHeight());
setAudioChannels (fDSP->getNumInputs(),
                  fDSP->getNumOutputs());
[...]

private:
   JuceGUI juceGUI;
```

A simple `buildUserInterface` call is needed, set the size of the `MainContentComponent`, and set the amount of audio channels. Following the

same spirit, there is optional code in case of a MIDI, OSC or polyphonic mode.

5 Other FAUST architecture files

Just a GUI architecture file isn't enough to run a FAUST program on JUCE, adaptations for different kind of control are also needed, such as OSC and MIDI.

5.1 OSC

OSC integration has been done by developing a new `JuceOSCUI` class, subclass of the base `UI` class. Two send and receive ports are defined. Input OSC messages are decoded by subclassing the JUCE `OSCReceiver` class, and implementing its `OSCReceiver::oscMessageReceived` method. Output OSC messages are sent by using the `OSCSender::send` method.

The special "hello" message allows to retrieve several parameters of the FAUST applications: its root OSC port, IP address, input and output port. The "get" message allows to retrieve the current, min and max values for a given parameter. Finally a float value received on a given path will allow to change the parameter value in real-time.

An application wanting to be controlled by OSC messages has to use an instance of the `JuceOSCUI` class, to be given to the DSP `buildUserInterface` method.

5.2 MIDI

MIDI messages handling is done by using the `MidiInput` and `MidiOutput` JUCE classes. A new `juce_midi` class subclassing the `MidiInputCallback` and implementing the required `MidiInputCallback::handleIncomingMidi Message` method has been defined. MIDI messages coming from the JUCE layer are decoded and sent to the corresponding application controllers. MIDI messages produced by the application controllers are encoded and sent using a `MidiOutput` object.

An application wanting to be controlled by MIDI messages has to use an instance of the `MidiUI` class, created with a `juce_midi` handler, to be given to the DSP `buildUserInterface` method.

6 Audio integration

To be connected to the external world, a given FAUST DSP has to be connected to an audio driver and a User Interface definition. JUCE framework already contains an abstract audio layer connected to a set of native audio drivers on all development platforms. JUCE developers can choose to deploy their code as standalone audio applications or audio plugins. A standalone application has to subclass the abstract `AudioAppComponent` class and implement the `prepareToPlay`, `getNextAudioBlock` and `releaseResources` methods:

- `prepareToPlay` is called just before audio processing starts with a sample rate parameter. The FAUST DSP is initiated with this sample rate value, and input/output channels number is possibly adapted to match the capabilities of the used native layer (that can a different number of input/output channels than the DSP).
- `getNextAudioBlock` is called every time the audio hardware needs a new block of audio data. Audio buffers presented as a `AudioSourceChannelInfo` data type are retrieved and adapted to be given to the FAUST DSP compute method.
- `releaseResources` is called when audio processing has finished. Nothing special has to be done at the FAUST level.

7 The faust2juce script

There are many scripts availiable in the FAUST ecosystem allowing to generate a ready to use binary, project file, or compiled file from a simple DSP file. They are labeled `faust2xxx`.

Following the same spirit, a `faust2juce` script has been implemented, that allows to create a JUCE project directory from a simple DSP file. The command is used as follow:

```
faust2juce [-options] dspFile.dsp
```

This will create a folder containing a .jucer file, and a "Source" folder containing the `Main.cpp` and the `MainComponent.h`. This folder is self contained, all needed FAUST includes are in the `MainComponent.h`, including the compiled DSP.

There are the options available at this moment for `faust2juce`:

- `-nvoices x`: produces a polyphonic self-contained DSP with x voices, ready to be used with MIDI events
- `-midi`: activates MIDI control
- `-osc`: activates OSC control

- `-help`: shows the different options available

As described in subsection 2.4, a number of voices can be hardcoded for a polyphonic DSP, but you can change it with the `nvoices` option. It has the priority over the metadata declaration. In the case of a non-hardcoded polyphonic DSP, it will just make it a polyphonic one with this compiler option. Some others options will be added later, it's still in development.

8 How to use JUCE architecture files

Using JUCE to export a FAUST DSP program file is easy: create the project folder with `faust2juce [-options] dspFile.dsp` and drag & drop the created folder named after the DSP to the "example" folder contained in the JUCE git folder.

Simply execute the .jucer file, and select "Save Project and Open in IDE...", the first time at least, to generate the JUCE header files, etc... And it's ready to execute your program on whatever export platform you chose.

9 Conclusion

The FAUST audio DSP language implementaion is now possible with JUCE, and can theoretically be exported to every platform that JUCE supports. It has been tested on OS X and iOS, both work correctly, and has a close performance to already available options, such as `faust2caqt` for OS X and `faust2ios`, for iOS.

MIDI control, polyphonic mode, and OSC control are implemented, more features are in progress of development, to permit a full compatibility with the whole FAUST library.

JUCE offers two types of "audio project", standalone applications or plug-in. Currently the FAUST architecture files are limited to describe standalone applications, but we are looking forward to adapt our code for plug-ins.

References

[1] Orlarey, Y., Fober, D., and Letz, S. (2009), "FAUST: an efficient functional approach to DSP programming." New Computational Paradigms for Computer Music, 290.

[2] D. Fober, Y. Orlarey, and S. Letz, "FAUST Architectures Design and OSC Support", *IRCAM*, (Ed.): Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-11), pp. 231-216, 2011.

[3] JUCE online documentation `https://www.juce.com/doc/classes`

[4] JUCE "Tutorial: Advanced Rectangle techniques" `https://www.juce.com/doc/tutorial_rectangle_advanced`

[5] J. Storer "Developing Graphical User Interfaces with JUCE", JUCE Summit 2015 `https://www.youtube.com/watch?v=xsCZoE1s_uw`