A JUCE Layout Builder

J.J. Girardot - jj@girardot.name

Abstract

This document presents «*LayoutBuilder*», a tool to build a (quick and dirty) GUI for your plugins made with JUCE. Basically, you describe the various elements (sliders, buttons, labels, etc.) that you need for your plugin, arranged in rows and columns, as you want them to appear in your interface.

This tool uses the juce module: `ff_layout` by *Daniel Walz*, available from `https://ffAudio.github.io/`, and the JUCE class «`AudioProcessorValueTreeState`» to manage the values of the plugin parameters.

1 Introduction

An important part of the time you spend building a plugin is spend in the design of the user interface, specially if you are in a prototyping phase and don't know in advance how many items you will finally be using. The idea of the proposed layout builder is to describe in a very simple way a GUI layout for the various graphical controlers you need, and let the system create a «usable» interface. Using this approach, it takes litteraly less than a minute to add a slider or make small changes to the layout of the GUI.

The resulting layout is quite appropriate for a prototype, even if you may wish to later completely redesign the GUI for a commercial plugin...

Note This is the first release of the tool so there are potentially dozen of bugs left, and many things not working as I'd like...

The tool itself is subject to the following copyright notice:

2 The concepts

We adopt here a «simplified» view of the various concepts related to a plugin:

the parameters are float values, that can be used and modified by the AudioProcessor part of the plugin, can be controlled by graphical controllers (sliders, buttons, etc.) that are visible in the GUI of the plugin, can be automated by the DAW host, and, together, can be saved and loaded as «presets». Each parameter is linked to a single controller.

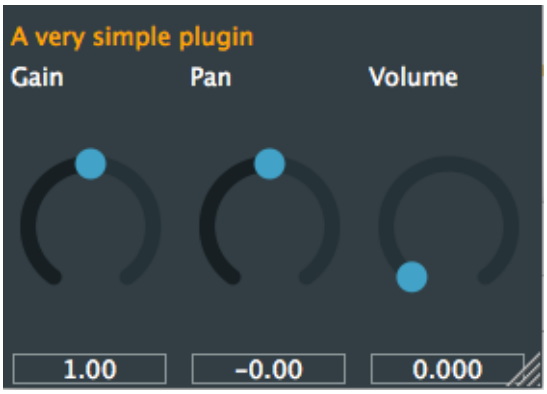

Fig. 1: *A simple plugin example*

the GUI is a geometric arrangement of various components, that can be graphical controllers (sliders, buttons, combo box, etc), graphical items (labels, pictures or just spaces), or containers that can themselves contain components (such as plain Components, GroupComponents, wave form displays, etc.). Inside a component, items are arranged in rows or columns, that can themselves contain rows or columns.

Stretching items The juce module: `ff_layout` by *Daniel Walz*, uses the concept of «stretch»: vertical and horizontal stretch factors are associated to each item, and the module tries to best fit all elements respecting these constraints.

The GUI and the parameters are described with a simple and terse formalism, that lets you express a large part of the possibilities offered by JUCE. This description is written in standard C++, and doesn't imply any external tool.

3 A first example

The GUI As a first example, imagine we need a GUI with three rotary sliders and the name of our plugin above them. Here is the full description of the layout, and a screen capture of the resulting GUI:

```
{
    // The main window
    startCmpt(pluginEditorWindow),
        aCol(),
            aRow(),
                aLabel(lab0), pad(bottom, 15), colorize(c_orange,c_transparentBlack),
            endRow(), setStretch(1.0, 0.35),
            aRow(),
                aSlider(sl1,rotVD), withLabel(lab1),
                aSlider(sl2,rotVD), withLabel(lab2),
                aSlider(sl3,rotVD), withLabel(lab3),
            endRow(),
        endCol(),
    endCmpt(pluginEditorWindow)
};
```

What this description, which is a kind of bytecode or pseudocode, basically says is:

- we are describing here the main editor window content (the «pluginEditorWindow» in `startCmpt()` and `endCmpt()`), other inner container components can be described the same way.
- this component contains a unique column (`aCol()`, `endCol()`). Of course, more columns can be added to the component.
- this column is made of two rows (`aRow()`, `endRow()`).
- the first row contains only a label ; «`lab0`» is a reference to the text of the label. For the sake of presentation, we add some padding under the label, and choose «orange» for the text, and «transparentBlack» for the background.
- the second row contains three sliders, described here left to right. A slider declaration references the number of the corresponding plugin parameter and specifies its geometry (here rotary sliders with vertical drag). By default, its text box is positionned under the slider, and we add here a label (defined by its number) above the slider.
- by default, all columns have the same width, all rows have the same height. However, we can change this by giving a stretch factor, and since the height of the label is less than the height of the sliders, we reduce its vertical stretch factor from 1.0 to 0.35.

- finally, note that the window is resizable, which lets you quickly determine the most appropriate size for it.

This layout is not an exotic external description, but just a vector of ints which can be directly built (thanks to the C++ compiler *constexpr*) inside your C++ code with no external tool involved.

The resources list This description of the GUI layout is just a part of the story. This is the declaration of the resources that we will be needing:

```
//-- Active components
static constexpr int sliderCount {3};  // we use 3 sliders
static constexpr int tbuttonsCount {0}; // no text buttons
static constexpr int tgbuttonsCount {0};  // no toggle buttons
static constexpr int comboBoxCount {0};  // no combo box
//-- Labels are graphic elements, but not active components
static constexpr int labelCount {4};
//-- Containers components hold active components, labels components, pictures, etc.
// There is at least one container, the Editor window, which is number "0"
static constexpr int componentCount {1};
static constexpr int pluginEditorWindow {0};

//------ Nothing needs to be changed after this comment
//-- All parameters are used through an array of pointers to float, which I call
// the "value array" with such a declaration: float * pParams[parCount];
// Containers and labels do not appear in the "value array"
// Total count of "active components" (or parameters)
static constexpr int parCount {sliderCount+tbuttonsCount+tgbuttonsCount+comboBoxCount};
//-- Positions of active components in the "value array"
// we arrange here in the order: all sliders, then text buttons, then toggle
// buttons, etc., which simplifies designation of items in the code
static constexpr int sliderFirst {0};
static constexpr int tbuttonsFirst {sliderFirst+sliderCount};
static constexpr int tgbuttonsFirst {tbuttonsFirst+tbuttonsCount};
static constexpr int comboBoxFirst {tgbuttonsFirst+tgbuttonsCount};
// Others components
static constexpr int labelFirst {comboBoxFirst+comboBoxCount};
```

As a matter of fact, we just indicate here that we use 3 sliders, no other active component, and 4 labels.

We then give symbolic names to the components, for using them in the algorithmic part of the plugin.

```
// Components identifications
enum cmptsIdts {
    // sliders
    sl1=sliderFirst, sl2, sl3,
    // text buttons
    // toggle buttons
    // labels
    lab0=labelFirst, lab1, lab2, lab3,
    //  others...
};
```

The resources description Finally, we describe as vectors of structs the various character strings (the «*stringSet*»), and parameters (the «*paramSet*»).

```
//--  Labels and strings used by the plugin
{
    { lo::none, "Error 80", "error" },
    { lab0, "A very simple plugin" },
    { lab1, "Gain"},
    { lab2, "Pan"},
    { lab3, "Volume" },
};
//--  Definition of the plugin parameters associated to the various sliders,
buttons, etc. The first number (an integer) is the item identification, and the others
values are what is required to declare the parameter: the range (typically,
```

```
    min value, max value, step, skew), and the default value
    */
    {
        { sl1, {0.0f, 2.0f, 0.01f}, 1.0f },
        { sl2, {-1.0f, 1.0f, 0.01f}, 0.0f },
        { sl3, {0.0f, 100.0f, 0.001f, 0.25f}, 0.0f},
    };
```

This last structure lets us describe the various characteristics of the parameters associated to the sliders. In particular, the values for the range will determine the behaviour of the sliders: min and max values, number of decimal digits displayed in the value box (deduced from the step value), and skew factor in the case of the last slider.

These structs reference the identifications that we choose for the sliders and the labels («sl1», «sl2», «lab3», etc). Let's note than none of the identifiers «pParams», «sl1», «lab1», etc, are imposed, and you can choose any name you want.

All these declarations can be inserted in the «pluginProcessor.h» file, or put in a separate header, which is then «#included» in the «pluginProcessor.h» file.

4 The whole plugin

You need other instructions to be added in your plugin, but these are always the same, whatever the design of the GUI is. These changes are described here, using the usual pluginProcessor/pluginEditor and .h/.cpp dichotomy.

4.1 The pluginProcessor.h

The file will include the resources list (see. 3 on the preceding page), which is used both by the AudioProcessor and the AudioEditor. One way is to include here the two modules that we use:

```
#include <ff_layout/ff_layout.h>
#include <jj_lbuilder/jj_lbuilder.h>
```

Before the class declaration, we put the ressource liste described in 3 on the previous page.

In the private section of the AudioProcessor class, we add:

```
AudioProcessorValueTreeState parameters;
lo::LayoutBuilder lBuilder; // the layout builder, unique for all components
float * pParams[parCount]; // for accessing the parameters values
std::vector<int> codeLout; // The layout description byte code
std::vector<struct lo::AppStringsDesc> allStrs; // strings used
std::vector<struct lo::AppValuesDesc> allVals; // parameters description
```

The «AudioProcessorValueTreeState» will manage the parameters of the plugin, and their links with the GUI. For example, the value of the first slider, referenced by «sl1», can be obtained by «*pParams[sl1]».

Again, let's note that the names «parameters», «lBuilder», «pParams», «parCount», «codeLout», etc. are not imposed, and you can choose better identifier as far as they a used coherently in the source code. As an exemple, «parCount», which is the number of parameters used in the plugin, should be the same as the identifier used as the result of the *«Total count of "active components"»* mentioned in the resource list.

4.2 The pluginProcessor.cpp

4.2.1 The constructor

In the «pluginProcessor.cpp» file, the essential work is to be done in the constructor. This one should first initialize the «AudioProcessorValueTreeState» and then the «LayoutBuilder», by:

```
: AudioProcessor (...), parameters (*this, nullptr), lBuilder (&parameters)
```

We declare the resources description (*stringSet*, *paramSet* and *code*) in the AudioPlugin constructor. Doing so let us keep the description private, should we use more than one instance of the plugin. Here is a possible way to do so:

```
{
    using namespace lo;
    allStrs = * new std::vector<struct AppStringsDesc>
    {
        { none, "Error 80", "error" },
        ...
    };
```

```
        allVals = * new std::vector<struct AppValuesDesc>
        {
            { sl1, {0.0f, 2.0f, 0.01f}, 1.0f },
            ...
        };
        codeLout = * new std::vector<int>
        {
            startCmpt(pluginEditorWindow),
            ...
        };
    };  // end using "lo"
```

Then, inside the constructor, insert:

```
lBuilder.setProcessor (this);
lBuilder.setCode (codeLout);
lBuilder.setStringsSet (allStrs);
lBuilder.setValuesSet (allVals);
lBuilder.setPAddresses (pParams, parCount);
lBuilder.setSliderCount (sliderCount, sliderFirst);
lBuilder.setTButtonCount (tbuttonsCount, tbuttonsFirst);
lBuilder.setTgButtonCount (tgbuttonsCount, tgbuttonsFirst);
lBuilder.setCmbBoxCount (comboBoxCount, comboBoxFirst);
// All parameters are ready to be created.
lBuilder.buildParameters();
parameters.state = ValueTree (Identifier ("MyPlugin"));
```

These different «*setX()*» methods give the builder all the information it needs to manage the declaration of the parameters, which is done in the «buildParameters» method.

4.2.2 Other Methods

There is a small change in the method creating the Editor, as you need to transmit the «AudioProcessorValueTreeState» and the «LayoutBuilder» which are owned by the AudioProcessor:

```
AudioProcessorEditor* MyPluginAudioProcessor::createEditor()
{
    return (AudioProcessorEditor*)
        new MyPluginAudioProcessorEditor (*this, parameters, lBuilder);
}
```

To have the «AudioProcessorValueTreeState» manage the saving and loading of presets, define the following methods:

```
void MyPluginAudioProcessor::getStateInformation (MemoryBlock& destData)
{
    MemoryOutputStream stream(destData, false);
    parameters.state.writeToStream (stream);
}
void MyPluginAudioProcessor::setStateInformation (const void* data, int sizeInBytes)
{
    ValueTree tree = ValueTree::readFromData (data, size_t (sizeInBytes));
    if (tree.isValid()) { parameters.state = tree; }
}
```

4.3 The pluginEditor.h

The «pluginEditor» class needs to inherit from «TypedComponent», and has to define the various graphical items used in the plugin. Here is the typical declaration:

```
class MyPluginAudioProcessorEditor  : public AudioProcessorEditor,
                                      public virtual TypedComponent
{
  public:
    MyPluginAudioProcessorEditor (MyPluginAudioProcessor&,
                                  AudioProcessorValueTreeState&,
```

```
                                    lo::LayoutBuilder&);
    ~MyPluginAudioProcessorEditor();
    void paint (Graphics&) override;
    void paintOverChildren (Graphics& g) override;
    void resized() override;
    void buildIt() override;
  private:
    typedef AudioProcessorValueTreeState::SliderAttachment SliderAttachment;
    typedef AudioProcessorValueTreeState::ButtonAttachment ButtonAttachment;
    typedef AudioProcessorValueTreeState::ComboBoxAttachment ComboBoxAttachment;
    //
    MyPluginAudioProcessor& processor;
    AudioProcessorValueTreeState& valueTreeState;
    Layout layout;
    lo::LayoutBuilder& lBuilder;
    // The local ressources:
    Label allLabels [labelCount];
    // interaction objects [some arrays empty...]
    Slider allSliders [sliderCount];
    TextButton allTButtons[tbuttonsCount];
    ToggleButton allTgButtons[tgbuttonsCount];
    ComboBox allComboBox[comboBoxCount];
    // The actual components list
    Component * comArray[componentCount];
    // Finally, the various attachments, so they are deleted first
    std::unique_ptr<SliderAttachment> slAttachment[sliderCount];
    std::unique_ptr<ButtonAttachment> tbAttachment[tbuttonsCount];
    std::unique_ptr<ButtonAttachment> tgAttachment[tgbuttonsCount];
    std::unique_ptr<ComboBoxAttachment> cmbBAttachment[comboBoxCount];
    JUCE_DECLARE_NON_COPYABLE_WITH_LEAK_DETECTOR (MyPluginAudioProcessorEditor)
};
```

In the private section, we just need to declare the various resources: sliders, labels, text buttons, toggle butons, etc.

The editor inherits of the «TypedComponent» and has to define the method «`buildIt`» to initialise the component (here the AudioProcessorEditor) so that it can be managed by the «Layout» manager.

4.4 The pluginEditor.cpp

4.4.1 The constructor

Some work is to be done in the Editor constructor. Here is what we have to add:

The constructor header looks like:

```
MyPluginAudioProcessorEditor::MyPluginAudioProcessorEditor
                              (MyPluginAudioProcessor& p,
                               AudioProcessorValueTreeState& vts,
                               lo::LayoutBuilder& lB)
    : AudioProcessorEditor (&p)
    , processor (p)
    , valueTreeState (vts)
    , layout (LayoutItem::LeftToRight, this)
    , lBuilder(lB)
{
    ...
};
```

The next two paragraphs describe what we need to insert inside the Editor constructor. Note that the code is «generic» in the sense it is the same whatever the layout of the GUI is, or the number of controls we use.

4.4.2 The editor is a «TypedComponent»

We need to initialize some values used by the «TypedComponent» interface:

```
layer = & lBuilder;
lgcNum = 0;  // The editor is always "component 0"
```

```
lcmp = lo::type_main; // its type is "main window"
playout = & layout;
```

4.4.3 The data for the Builder

We send the rest of needed information to the Builder:

```
lBuilder.setEditor (this); // Declare the Editor
lBuilder.setCompDesc (comArray, componentCount); // provide the component array
lBuilder.registerComponent (this); // Register the AudioEditor as component 0
lBuilder.setSliderList (allSliders, slAttachment, sliderCount); // sliders
lBuilder.setTButtonList (allTButtons, tbAttachment, tbuttonsCount); // text buttons
lBuilder.setTgButtonList (allTgButtons, tgAttachment, tgbuttonsCount); // toggle buttons
lBuilder.setCmbBoxList(allComboBox, cmbBAttachment, comboBoxCount); // declare the combo b
lBuilder.setLabelList (allLabels, labelCount); // labels
```

Then, we build the code for all the components (in this simple example, there is only one):

```
for (int i = 0; i < componentCount; i++)
{
    TypedComponent * tc = dynamic_cast<TypedComponent *> (comArray[i]);
    if (tc != nullptr)
    {
        lBuilder.clearError();
        tc->buildIt(); // This calls the builder for this component
    }
}
```

Finally, we declare the size of the Editor window, making it resizable if we wish:

```
setResizable (true, true);
setResizeLimits (160, 120, 1500, 900);
setSize (300, 200);
```

4.4.4 Other methods

The «`resized`» method is the one which calls the Layout manager:

```
void MyPluginAudioProcessorEditor::resized()
{
    layout.updateGeometry ();
};
```

If no other work is needed, the «`paint`» method can just paint the background:

```
void MyPluginAudioProcessorEditor::paint (Graphics& g)
{
    g.fillAll (getLookAndFeel().findColour (ResizableWindow::backgroundColourId));
};
```

The «`buildIt`» method is invoked once, during the GUI construction:

```
void MyPluginAudioProcessorEditor::buildIt()
{
    layer->setLayoutDesc(playout, this);
    layer->buildLayout(lgcNum);
};
```

Note that there is an unique instance of the LayoutBuilder, but a separate instance of the Layout manager for each component.

5 Managing other components

The LayoutBuilder can manage various types of components:

1. The plugin Editor itself,

2. «LayedComponent(s)» which are simple containers that contain active components such as sliders, buttons, labels, etc.
3. «LayedGroupComponent(s)» which are grouped components (see Juce *GroupComponent*), and provide a frame and a title.
4. «Canvas» which are typically used for creating graphical representations. For each instance of a canvas, you can provide specific «paint» and «resized» functions.

These components inherit from «TypedComponent». To use such a component, you need to:

- declare the component in the Editor class, like «`LayedGroupComponent frameEQ;`»
- initialise the component in the header of the Editor constructor: «`frameEQ(&lBuilder, 1),`» where «1» is the component number
- in the Editor constructor declare the component to the LayoutBuilder: add «`lBuilder.registerComponent(&frameEQ);`»
- describe the layout of this component inside the layout bytecode.
- insert the component itself as an entry (with `theCmpt(1)`) in the bytecode description of the main window.

Canvas For the canvas elements, you may wish to declare «paint» and «resized» functions. For example, define:

```
static void paintItBlack(LayedCanvas * can, int idt, Graphics& g)
{
    g.setColour (can->bgColor);
    g.fillRect (can->getLocalBounds());
    g.setFont (Font (16.0f));
    g.setColour (Colours::white);
    g.drawText (String(idt), can->getLocalBounds(), Justification::centred, true);
}
```

and then in the Editor constructor associate this paint function with a specific canvas component (for example, `frame5`):

```
frame5.providePaintMethod(paintItBlack);
```

Similarily, define somewhere the resized function:

```
static void isResized(LayedCanvas * can, int idt) { ... }
```

Declare the function in the Editor constructor:

```
frame5.provideResizedMethod(isResized);
```

Note that both methods receive a reference to the component, and the identification of this component.

6 A second example

The picture 2 on the facing page shows a slightly more complex example, designed to give you an idea of the various possibilities of the builder. Here is the code for this example:

```
[1]     constexpr int stdsSlColors { colorize(c_orange, c_green, c_grey) };
        codeLout = * new std::vector<int> {
            // Global declarations
[2]         setSldPdef(dfSl), setTbtPdef(dfTB), setTgbPdef(dfTB),
            // First define some common declarations
            startDcl(),
[3]             setColor(c_A, c_orange), // chose a default main color for the frames
            endDcl(),
            //The main window contains a label, a combo box and 5 components
            startCmpt(pluginEditorWindow),
                aCol(),
                    aRow(),
[4]                     aLabel(lab0), fontFace(font2), fontStyle(0, 0, j_centred),
                            colorize(c_orange,c_transparentBlack), setStretch(3.5),
[5]                     aCmbBox(cb1, cb1itms), pad(vertical, 5), setStretch(1.5),
                    endRow(), setStretch(1.0, 0.1),
                    aRow(),
[6]                     theCmpt(1),
```

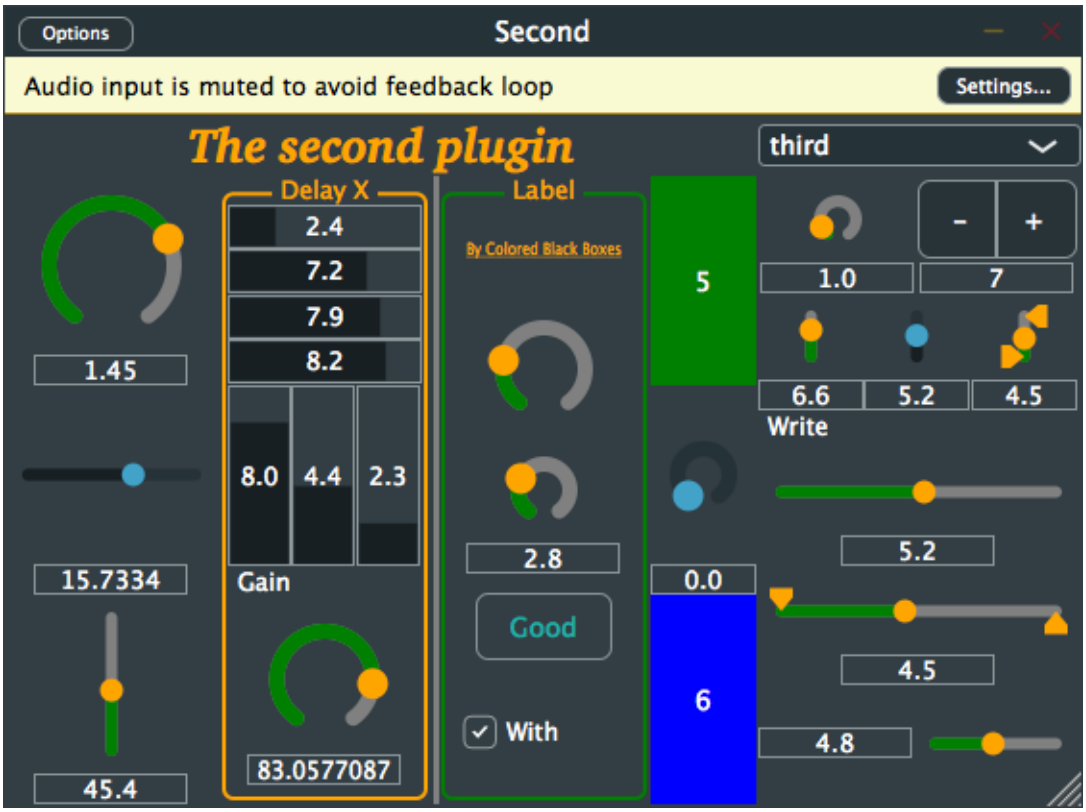

Fig. 2: *A second plugin example*

```
                        theCmpt(2),
[7]                     aSplitter(0.4),
                        theCmpt(3),
                        theCmpt(7), setStretch(0.5),
                        theCmpt(4), setStretch(1.5),
                    endRow(),
                endCol(),
            endCmpt(pluginEditorWindow),
            // Component 1
[8]         startCmpt(1, topDown),
                aSlider(b1s1,0), stdsSlColors,
                aSlider(b1s2,linH),
                aSlider(b1s3,linV), stdsSlColors,
            endCmpt(1),
            // Component 2
[9]         startCmpt(2),
                aCol(),
                    aCol(),
                        aSlider(b2s1a,barH), stdsSlColors,
                        aSlider(b2s1b,barH), stdsSlColors,
                        aSlider(b2s1c,barH), stdsSlColors,
                        aSlider(b2s1d,barH), stdsSlColors,
                    endCol(),
                    aRow(),
                        aSlider(b2s2a,barV), stdsSlColors,
                        aSlider(b2s2b,barV), stdsSlColors,
                        aSlider(b2s2c,barV), stdsSlColors,
                    endRow(),
                    aSpace(), setHeight(20,20),
                    aSlider(b2s3,rot,2), withLabel(lab7), stdsSlColors,
[10]            endCol(), pad(15,10,5,5),
[11]            frameLab(lab2, c_orange), // Build the frame and label
            endCmpt(2),
            // Component 3
```

```
[12]        startCmpt(3),
                aCol(),
[13]                aLabel(lab1), setHeight(35,55), pad(vertical, 10),
                        colorize(c_orange,c_transparentBlack),
                        fontStyle(9.5, s_underlined),
[14]              aSlider(b3s1,rotVD,noTB), stdsSlColors, setHeight(50,150),
                        setWidth(45,140),
                    aSlider(b3s2,rotHD), stdsSlColors,
[15]                aTButton(tbt1,tgB), setHeight(25,55), pad(allDirections, 10),
                        colorize(c_lightseagreen),
[16]                aTgButton(tgbt1), setHeight(35,55), colorize(c_lightseagreen),
[17]            endCol(), pad(allDirections, 10),
[18]            frameLab(lab6, c_orange, c_green),
            endCmpt(3),
            // Component 4
            startCmpt(4, topDown),
                aRow(),
                    aSlider(b4s1,rotHVD), stdsSlColors,
                    aSlider(b4s2,incDec), stdsSlColors,
                endRow(),
                aRow(),
                    aSlider(b4s3,linV), stdsSlColors,
                    aSlider(b4s4,linV),
[19]                aSlider(b4s5,lin3V), stdsSlColors,
                endRow(),
                aSpace(), setHeight(20,20),
                aSlider(b4s6,linH), stdsSlColors, withLabel(lab5),
[19]            aSlider(b4s7,lin3H), stdsSlColors,
                aSlider(b4s8,linH,tBLeft,2), stdsSlColors,
            endCmpt(4),
            // Component 7
            startCmpt(7, topDown),
[20]            theCmpt(5),
                aSlider(b1s4,0),
[20]            theCmpt(6),
            endCmpt(7),
        };
```

Some comments about this code:

1. This first line declares a «code», `stdsSlColors`, that we will be using when we want to change the set of colors used for a control so it reflects a specific «look & feel».

2. Here we use `setSldPdef(dfSl)` to declare a default entry (here, `dfSl`) for any slider parameter for which no specific description is available in the *paramSet*. The other instructions on this line have similar effects for text buttons and slider buttons.

3. Another way to «customize» is to change «user colors» and use them...

4. instructions of these two lines are related to defining our main label. We first add a label, defined as entry «lab0» in the *stringSet*, then change the Font used with the `fontFace` function. That one uses entry «font2» in the *stringSet*, where the entry specifies as strings the font name, the font type, and the font size. Then there is the `fontStyle` function, with parameters specifying the font size (set to 0 to mean «unchanged»), the font style (bold, italic, etc., also set to 0 to mean «unchanged»), and the text justification, here «j_centred», for centered. When both functions `fontFace` and `fontStyle` are used, they should be used in that order, as the second modifies the result of the first. We the use colorize to change the default color of the text and the background, and finally set an horizontal stretch factor of 3.5, so that, combined with the 1.5 factor associated later with the comboBox, the label occupies 7/10 of the width of the plugin.

5. This is a declaration of a *comboBox*. The first parameter in an entry, in both *stringSet*, to set the name of the associated parameter, and in the *paramSet*, to indicate the corresponding range. The second parameter in an entry in the *stringSet*, to specify a vector of strings that will represent the various items of the *comboBox*. We then add a little vertical padding of 5 pixels, above and under the *comboBox*, to make it nice looking, and set an horizontal stretch factor of 1.5.

6. We organize the lower part of the plugin as 5 columns, each one containing a component. This row therefore contains, from left to right, components 1, 2, 3, 7 and 4 (we could of course have associated symbolic identifications, through an enum, to these components). The stretch factor associated tho each item in 1.0 by default, but as we wanted component 7 to be smaller,

and component 4 to be larger, we changed the stretch factor for these two ones. The layouts of these components are described later in the code.

7. Just for the fun, we add a splitter between components 2 and 3. This item appear as a vertical grey line, and can be moved with the mouse to enlarge or reduce the left and right sides of the plugin. the default value is 0.4, which means the 40% of the width of the plugin is allocated to the left part, anf 60% to the right part. No additional constraints are associated to the various items, so by moving the splitter, you can allocate up to 100% of the width to the left or the right side.

8. This is the declaration of component #1, which is a plain component. We add a second parameter to indicate that it is organized *topDown*, as rows, rather than *leftToRight*, as columns, the default value. Note that in this component, some sliders have the `stdsSlColors` command following them, and therefore appear with a color set different from the classical look & feel.

9. Declaration of a second component, which is a group component. This one is also organized topDown, with a column as the first block, containing 4 horizontal slider, a row as the second blocl, containing 3 vertical sliders, a small vertical space, defined by the «aSpace()» construct, with a «setHeight» modifier specifying 20 as both the minimum and the maximum height, and finally a slider with a label attached.

10. As the current component is a group component that will get a frame and a label, it is a good idea to pad its contents with some space. Here, we add 15 pixels at the top, 10 at the bottom, and 5 on each side.

11. Finally, we provide the label «lab2» for the frame, and specify the color to use, «c_orange».

12. Component 3 is also a grouped component in which we will arrange vertically various items.

13. We insert first a label, with modifiers similar to the ones unse for the main label (see 4 on the preceding page). The `setHeight` indicate that the height of the item can vary between 35 and 55 pixels. The size of the font is set to 9.5, and the style to *underlined*.

14. We have then a rotary slider with vertical drag (`rotVD`) and no text box (`noTB`), with specifications for the minimum and maximum values of height and width.

15. Here we declare a text button. The entry «tbt1» defines the text to appear in the button itself, and the second, optional, parameter (`tgB`) indicates that it acts as a toggle button.

16. A toggle button, with «tgbt1» to specify the label.

17. As for the previous component, we add some padding around the contents to manage space for the frame and label. Here, we specify 10 pixels in all directions.

18. Finally, we define the label, and specify orange for the text, green for the frame.

19. We have here two sliders with 3 indexes. Sadly, only the center index is managed by the system.

20. Components 5 and 6 are «canvas» components. We have a paint function associated to them, which fills the backgroup and displays the component number.

7 Constructing the interface

7.1 «Local concepts»

The last example makes references to some *colors*, or *directions*, or *justifications*, etc. All these are JUCE concepts that you manipulate through local conventions, which are described in the next paragraphs.

7.1.1 Colors

The layout builder does not provide access to all the colors defined in the JUCE framework, but only to a subset of 64. 48 of them are colors from the JUCE color table, and are defined with similar names. Here is the set of those colors:

c_transparentBlack, c_black, c_white, c_red, c_green, c_blue, c_yellow, c_grey, c_greenyellow, c_lightblue, c_lightseagreen, c_purple, c_crimson, c_darkblue, c_cyan, c_darkcyan, c_orange, c_darkorange, c_darkred, c_turquoise, c_darkturquoise, c_indianred, c_limegreen, c_orangered, c_springgreen, c_yellowgreen, c_aquamarine, c_beige, c_brown, c_azure, c_coral, c_darkgrey, c_darkslategrey, c_lightgrey, c_lightsteelblue, c_lime, c_slategrey, c_silver, c_tan, c_mediumpurple, c_mediumslateblue, c_forestgreen, c_teal, c_aqua, c_skyblue, c_midnightblue, c_sienna

You can also access 16 definable colors, under these names:

c_A, c_B, c_C, c_D, c_E, c_F, c_G, c_H, c_I, c_J, c_K, c_L, c_M, c_N, c_O, c_P

To define one of these colors, use the «`defColor`» opcode.

7.1.2 Directions and localisations

These are used in the layout specifications. A *direction* is one element of the set: *topDown, bottomUp, leftToRight, rightToLeft*.
A *localisation* is one element of the set: *top, bottom, left, right, horizontal, vertical, allDirections*.
A *text box position* is one of the set: *noTB, tBLeft, tBRight, tBAbove, tBBelow*.

7.1.3 Font styles

Used by the `fontStyle` opcode. A *font style* is *s_plain*, or an additive combination of *s_bold*, *s_italic*, and *s_underlined*.

7.1.4

Used for labels. A *text justification* is one of *j_none, j_left, j_right, j_horizontallyCentred, j_top, j_bottom, j_verticallyCentred, j_horizontallyJustified, j_centred, j_centredLeft, j_centredRight, j_centredTop, j_centredBottom, j_topLeft, j_topRight, j_bottomLeft, j_bottomRight*.

7.2 The «opcodes»

You provide the layout description bytecode as a vector of *ints*, using a set of predefined *constexp* values and functions that you include, with «#include "jj_lbuilder.h"» in your «PluginProcessor.h» file. Some opcodes make reference to a «*color*», which, as explained in paragraph 7.1.1 on the preceding page, is not a «standard» JUCE color, but an entry number in a table of 64 predefined colors.

Note that the byte-code operations are internally defined as *constexpr functions* returning *ints*, and are used to define the values of a vector of *ints*. The calls to these functions must therefore be separated by comas.

The set of bytecodes can be classified as layout instructions, active components, graphical elements, modifiers and setting options.

7.2.1 Layout instructions

startCmpt(int value, int mode=leftToRight) indicates the beginning of a *container component*. value is the number of the component, 0 being the plugin Editor window. The second parameter lets you indicate the layout mode for the inner items, as *leftToRight*, *rightToLeft*, *topDown* or *bottomUp*. The description of the component must end with an *endCmpt(value)* with the same number as parameter.

endCmpt(int value) closes the corresponding (same parameter) component start.

aRow(int dir=leftToRight) marks the beginning of a new row. The parameter indicates the layout mode for the inner items, and can be *leftToRight* or *rightToLeft*.

endRow() closes the corresponding row start.

aCol(int dir=topDown) marks the beginning of a new column. The parameter indicates the layout mode for the inner items, and can be *topDown* or *bottomUp*.

endCol() closes the corresponding column start.

startDcl() marks the beginning of a set of common declarations. There must be only one *Dcl* block.

endDcl() closes the «startDcl()».

7.2.2 Active components

aSlider(int idt, int par2=0, int par3=0) insert in this position a new slider.

- The first parameter is the slider number, and should be in the range defined by $[sliderFirst, sliderFirst + sliderCount)$, which is the case if you define the slider references in an *enum* beginning at *sliderFirst*. This number is the plugin parameter number corresponding to the slider, and must also be a key in the *valuesSet* structure. However, if the key is not found in the *valuesSet*, a default entry is used to provide appropriate values for the parameter.
- The second parameter is the slider type, which is one of {*linH, linV, barH, barV, rot, rotVD, rotHD, rotHVD, incDec, lin2H, lin2V, lin3H, lin3V*}, which stand for the traditional { *Slider::LinearHorizontal, Slider::LinearVertical, Slider::LinearBar, Slider::LinearBarVertical, Slider::Rotary, Slider::RotaryVerticalDrag, Slider::RotaryHorizontalDrag, Slider::RotaryHorizontalVerticalDrag, Slider::IncDecButtons, Slider::TwoValueHorizontal, Slider::TwoValueVertical, Slider::ThreeValueHorizontal, Slider::ThreeValueVertical* }. Note that the system currently doesn't know how to handle two and three value sliders (and neither do I).
- The third parameter indicates the position of the slider textbox. Use { *noTB, tBLeft, tBRight, tBAbove, tBBelow* } for the corresponding { *Slider::NoTextBox, Slider::TextBoxLeft, Slider::TextBoxRight, Slider::TextBoxAbove, Slider::TextBoxBelow* }. Not all positions are pertinent for all sliders types.

aTButton(int idt, int par2=0, int par3=0) insert in this position a Text Button.

- The first parameter is the button number, and should be in the range defined by $[tbuttonsFirst, tbuttonsFirst + tbuttonsCount)$, which is the case if you define the button references in an *enum* beginning at *tbuttonsFirst*. This number is the plugin parameter number corresponding to the button, and must also be a key in the *valuesSet* structure. However, if the key is not found in the *valuesSet*, a default entry is used to provide appropriate values for the parameter.
- The second parameter is a string number, and defines the string used to decorate the button. The default value «0» indicates that no text should be put on the button.
- The third parameter may be left out. You can use here the value «`tgB`» to indicate that the text button works as a toggle button, with two visible states, *on* and *off*.

aTgButton(int idt, int par2=0, int par3=0) insert in this position a Toggle Button. The parameters have the same meaning as those of the Text Button.

aCmbBox(int idt, int par2=0, int par3=0) insert in this position a ComboBox.

- The first parameter is the combo box number, and should be in the range defined by $[comboBoxFirst, comboBoxFirst + comboBoxCount)$, which is the case if you define the button references in an *enum* beginning at *comboBoxFirst*. This number is the plugin parameter number corresponding to the combo box, and must also be a key in the *valuesSet* structure.
- The second parameter is an entry in the stringSet, which is a vector of strings describing all the items of the combo box.
- The third parameter is currently unused.

theCmpt(int idt) insert here the component numbered «idt», described inside the corresponding (startCmpt, endCmpt) pair.

7.2.3 Graphical elements

aLabel(int idt) insert here a Label item, with text given by the corresponding entry in the vector of «AppStringsDesc».

frameLab(int idt, int par2=0, int par3=0) defines the label and frame for the current «LayedGroupComponent». The first parameter is a string number in the «AppStringsDesc» vector, the second and third parameter are the colors of the text and frame. Colors are colors references, as already explained. This command is ignored if the current component is not a «LayedGroupComponent».

aSpace(float width=1.0f, float height=1.0f) this inserts a space, with a given stretch factor for the width and/or the height. The default value is 1.0 for both directions. Values can be set between 0.01f and 40.0f for both directions.

aSplitter(float ratio=0.5f) this inserts a splitter, with an initial given ratio between the left and right part of the current row, or the top and bottom of the current column. The splitter appears as a grey line, and can be moved with the mouse to enlarge or reduce the areas it delimits.

7.2.4 Setting Options

defColor(int n, int r, int g, int b) which defines color «n» (from c_A to c_P) by the values of its RGB components, like `defColor(c_E, 255, 255, 0)`. The alpha channel is set to 0xFF (no transparency).

defColor(int n, int c) which defines color «n» from a 24 bit integer representing the RGB combination, like `defColor(c_E, 0xffff00)`. The alpha channel is set to 0xFF (no transparency).

setSldPdef(int idt) sets a default entry for the sliders parameters. For example, if you use «setSldPdef(int idt)», when you define a slider with number «sl», and this number is not found as an entry in the «AppValuesDesc», then this entry is used instead.

setTbtPdef(int idt) sets a default entry for the text buttons parameters and has an action similar to «setSldPdef».

setTgbPdef(int idt) sets a default entry for the toggle buttons parameters and has an action similar to «setSldPdef».

7.2.5 Modifiers

Modifiers are operations that can appear after the declaration of an item (such as an active component, a label, etc.) and whose purpose is to modify some aspect of the physical rendering of the item. This is the set of modifiers operations:

pad(int where, int val) add a padding (expressed as a number of virtual pixels) around the component. The first parameter indicates where the padding takes place, and the second is an integer value in pixels (in the 1-4095 range). The «where» parameter can be one of the constants: *top, bottom, left, right, horizontal, vertical, allDirections*.

pad(int *top*, int *bottom*, int *left*, int *right*=0) add a padding (expressed as a number of virtual pixels) around the component. Each parameter can be a different integer value (in the 0-63 range).

setHeight(int min, int max=0) specify for the component a minimum and/or a maximum height. Values are in pixel (in the 1-4095 range). A value equal to zero is ignored, and doesn't modify the corresponding characteristic of the item.

setWidth(int min, int max=0) specify for the component a minimum and/or a maximum width. Values are in pixel (in the 1-4095 range). A value equal to zero is ignored, and doesn't modify the corresponding characteristic of the item.

fixSize(int width, int height=0) specify for the component a fixed width and/or a fixed height. Values are in pixel (in the 1-4095 range). A value equal to zero is ignored, and doesn't modify the corresponding characteristic of the item.

setStretch(float width, float height=0.0f) specify for the component a stretch factor for the width and/or the height. The default value is 1.0 for all directions. Values can be set between 0.01f and 40.0f for both directions.

setRatio(float ratio) sets a fixed ratio (width/height) for the component, so its aspect does not change when the main window is resized.

colorize(int c1, int c2=c_transparentBlack, int c3=0, int c4=0) this lets you specify up to 4 colors. Any parameter not equal to 0 specifies a specific color that is applied to the previous item. The result depends on the item:

sliders c1 is the thumb color, c2 the track color, c3 the outline color, and c4 the text box background.

buttons c1 is the «off» text color, c2 the «off» background color, c3 the «on» text color, c4 the «on» background color.

labels c1 is the text color, c2 the background color, c3 the «edit» text color, c4 the outline color.

fontStyle(float size=0.0f, int style, int justif) apply a specific font style and/or font size to the previous item. Font style is one of plain, bold, italic, underlined, and the size can be between 0.01 and 40.0. Justif is one of the keywords defined in section 7.1.4 on page 12.

fontFace(int ent) chooses a new Font, defined as entry «ent» in the *stringSet*. This entry is a vector of three strings, defining the font name, the font style and the font size.

7.3 How the «bytecode» is used

A first «execution» of the bytecode is performed in the AudioProcessor constructor. All the instructions in the bytecode are sequentially executed (or ignored). More precisely, the «buildParameters()» method used from the AudioProcessor ignores all instructions but:

- the «setting» instructions (like «setSldPdef», etc.) which provide some default behavior;
- the «active components» declarations (like «aSlider», etc.) which correspond to the declaration of a parameter.

Inside the AudioProcessorEditor, the bytecode is used a second time. Very precisely, for each container component «i», the «buildIt()» method of the component is called. This method executes in sequence:

- the code between the «startDcl()» and the «endDcl()» bytecode, which provide common settings for all the components;
- the code between the «startCmpt(i)» and «endCmpt(i)», which is specific for component «i».

8 AudioProcessor and Editor programming

The layout builder creates all the parameters cooresponding to the graphical controllers. For any controller (like a slider) with some identifier (like «sl3b2=6»), if there exists an entry «6» in the *stringSet*, the first string of this entry is used as the parameter name, otherwise the parameter is called «slider6». However, in all cases, the corresponding component is named «slider6», where «6» is also the entry in the raw ptr array.

8.1 Audio Parameters

All the audio parameters can be easily accessed and modified through an array of pointers to floats. Use the component number (like «sl3», declared in the «enum cmptsIdts») and the raw ptr array:

```
float x = *pParams[sl3];
```

9 Source code

The source code is available as: