Audio Effects Theory & Design

1 Use the Projucer to Create a New Audio Plugin

I. Settings (see “Software Introduction” Slides)

A. Manufacturer Code: Mu48

B. Manufacturer: MUS483

C. AU, VST3

II. Compile Plugin using IDE (i.e. Xcode) and Test in DAW (i.e. Logic)

2 Create a User Parameter in the PluginEditor

Yellow highlights below means this is new code you will ADD.

Non-highlighted code serve as clues as to WHERE you should look to add your code.

Grayed out “note:” are simply notes about what the code is doing and where code should go. Do not copy.

III. Create a pointer for your slider in the class definition (in PluginEditor.h)

```
private:
   juce::Slider volume;
```

```
note: slider is an object JUCE allows you to implement
```

IV. Create a listener to the slider in the class header (PluginEditor.h)

```
class StarterCodeAudioProcessorEditor  : public AudioProcessorEditor,
                                         public juce::Slider::Listener
```

```
note: a Listener enables you to “listen” for events/time and update an object
```

V. Instantiate a new slider in the PluginEditor Constructor (in PluginEditor.cpp)

```
{

volume.setRange(0.0, 100.0, 1.0);
    volume.setSliderStyle(juce::Slider::LinearBarVertical);
```

```
    volume.setTextValueSuffix("linear gain");
    volume.setTextBoxStyle(juce::Slider::TextBoxBelow, false, 60, 20);
    volume.addListener(this);
    addAndMakeVisible(volume);
```

note: comes before setSize() and inside the curly braces of the function{}!

VI. Specify how to draw your slider inside the resized() method (in PluginEditor.cpp)

```
    void StarterCodeAudioProcessorEditor::resized()
    {
    int w = 50; int h = 200; int x = 20; int y = 60;
     volume.setBounds(x, y, w, h);

    }
```

note: located in resized() function. goes between curly braces. alternatively, you can do this without variables, like …. volume.setBounds(20, 60, 50, 200);

---

3 Make the PluginEditor respond to parameter changes from the UI and automation

I. Add a Timer object, TimerCallback function and SliderValueChanged function (PluginEditor.h)

```
class StarterCodeAudioProcessorEditor  : public AudioProcessorEditor,
                                         public juce::Slider::Listener, public
juce::Timer

public: …
    void resized() override;
    void sliderValueChanged(juce::Slider*) override;
     void timerCallback() override;
```

note: a Timer allows parameters to update their value if the host changed them using parameter automation. valueChange() allows parameters to update their value whenever the user moves a Slider*.

II. Create a function that updates the parameter when the slider is moved. (PluginEditor.cpp)

```
void StarterCodeAudioProcessorEditor::sliderValueChanged(juce::Slider* slider)
{
    // make an additional "else if" for each slider
    // &volume is a reference to our Slider* variable
    if (slider == &volume) {
        *audioProcessor.volumeUParam = (float) volume.getValue();
    }
}
```

note: add this code at the very bottom of PluginEditor.cpp

note: we haven't created the variable volumeUParam yet. This will be done in PluginProcessor and create a connection between the UI and the audio processor.

```
&volume is a reference to our Slider* variable. https://www.geeksforgeeks.org/pointers-vs-references-cpp/

note: “StarterCodeAudioProcessorEditor” will throw “Use of undeclared identifier” error. This is based on your Plugin name and is not generic. Just look at the other “void PluginNameProcessorEditor” to see specific naming.
```

III. Make it so that the slider gets updated if the DAW host changes the parameter via parameter automation. (PluginEditor.cpp)

```
void StarterCodeAudioProcessorEditor::timerCallback()
{
    //Update the value of each slider to match the value in the Processor
    volume.setValue(*audioProcessor.volumeUParam, juce::dontSendNotification);

}


note: add this code at the very bottom of PluginEditor.cpp

note: we haven’t created the variable volumeUParam yet. Parameter automation only updates the PluginProcessor but not the PluginEditor. So we create a timer callback function, which gets called periodically, and which updates the sliders to the current state of the user params in the PluginProcessor

note: “StarterCodeAudioProcessorEditor” will throw “Use of undeclared identifier” error. This is based on your Plugin name and is not generic. Just look at the other “void PluginNameProcessorEditor” to see specific naming.
```

IV. Initialize the timer (PluginEditor.cpp)

```
setSize (400, 300);
startTimer(100); //right after setSize(), based in ms

 note: timerCallback gets called every X ms based upon our value in startTimer(). So every 100 ms this is called. Placed inside the constructor function, right after setSize()
```

---

4 In the PluginProcessor, use the user parameter data to calculate algorithm parameter values

I. Create any member variables you may need, including any algorithm parameters (PluginProcessor.h)

```
 public: …
 //Add a public pointer for each user parameter, do this at the BOTTOM of public, right before private:
juce::AudioParameterFloat* volumeUParam;

 private: …
```

```
 //Create private variables for algorithmic parameters
float volumeAParam;
```

II. Initialize your variables, both in the PluginProcessor constructor, and in PrepareToPlay( ) (PluginProcessor.cpp)

```
  note: inside StarterCodeAudioProcessor(){}, inside the curly braces of the
function, look for
 #endif {<code goes here between curly braces>}

addParameter (volumeUParam = new juce::AudioParameterFloat (
   "volume", // parameterID
   "Volume", // parameter name
    juce::NormalisableRange<float> (0.0f, 100.0f), //parameter range
    80.0f)); // default value


inside prepareToPlay() {}, scroll down to find the function and put inside the
curly braces.

    // Use this method as the place to do any pre-playback
    // initialisation that you need..
    volumeAParam = *volumeUParam;


note: * is a pointer to the variable (points to place in memory). https://
www.geeksforgeeks.org/pointers-vs-references-cpp/
```

III. Create any methods you will need to calculate the algorithm parameters from the user parameters (PluginProcessor.h). Typically these are private methods.

```
private:
        void calcAlgorithmParams();
```

IV. Use these methods to calculate the algorithm parameter values (PluginProcessor.cpp)

```
void StarterCodeAudioProcessor::calcAlgorithmParams()
{
    volumeAParam = 0.1 * *volumeUParam;
}

note: I typically place this function RIGHT above processBlock() function since
this function gets called within the function processBlock()

note: "StarterCodeAudioProcessor" will throw "Use of undeclared identifier"
error. This is based on your Plugin name and is not generic. Just look at the
other "void PluginNameAudioProcessor" to see specific naming.
```

5 Use your algorithm parameters to modify the audio

I. Make sure these methods get called at the appropriate time (PluginProcessor.cpp).

```
note: inside ::processBlock() but BEFORE the buffer/sample loop…

    calcAlgorithmParams();
     const int numSamps = buffer.getNumSamples();

     // This is the place where you'd normally do the guts of your plugin's
     // audio processing...
     // Make sure to reset the state if your inner loop is processing
     // the samples and the outer loop is handling the channels.
     // Alternatively, you can process the samples with the channels
     // interleaved by keeping the same state.
     for (int channel = 0; channel < totalNumInputChannels; ++channel)
     {
         auto* channelData = buffer.getWritePointer (channel);

         // ..do something to the data...
         for (int i = 0; i < numSamps; ++i)
         {
             channelData[i] = volumeAParam * channelData[i];
         }
     }

note: The actual audio processing takes place in the PluginProcessor's
processBlock( ) method (PluginProcessor.cpp). The "inner loop" is where each
output audio sample is calculated from each input audio sample. This is where
every audio sample is processed! Pretty cool!
```

6 Compile, Test, and Debug

Anytime you rebuild your plugin, you may wish to quit and reopen your DAW (i.e. Logic). This will refresh the plug-in state and minimize errors.

STOP!

You should compile your plugin and check your work. Once your plugin has compiled, remember, you need to quit Logic or your DAW and reopen before you'll see the plugin. And if you don't remember to check the folders

~/Library/Audio/Plug-ins OR /Library/Audio/Plug-ins

7 Saving Your Plug-In State (Closing and opening of DAW session)

I. Get the parameter's state from memory and store in XML (PluginProcessor.cpp)

```
note: inside ::getStateInformation(), inside the curly braces!

    std::unique_ptr<juce::XmlElement> xml (new juce::XmlElement ("VolumeParams"));
    xml->setAttribute ("volume", (double) *volumeUParam);
    copyXmlToBinary (*xml, destData);
```

```
note: You only need to establish the XmlElement once. When you add other parameters,
you only need to add another line (xml->setAttribute) specific to your user parameter.
For example if we have a low shelf filter with a user parameter called
“filterLowShelfParam” we just add one line before copyXml. e.g.
xml->setAttribute ("lowshelf", (double) *filterLowShelfParam);
```

II. Restore the parameter’s state from memory upon open

```
note: inside ::setStateInformation(), inside the curly braces

    std::unique_ptr<juce::XmlElement> xmlState (getXmlFromBinary (data,
sizeInBytes));
    if (xmlState.get() != nullptr)
        if (xmlState->hasTagName ("VolumeParams"))
            *volumeUParam = xmlState->getDoubleAttribute ("volume", 1.0);
```

```
note: Similarly, you only need to get the single xmlState. If you have other user
params, just add your user params at the end.
e.g. *filterLowShelfParam = xmlState->getDoubleAttribute ("lowshelf", 100.0);
```

---

8 Automation Modes (Touch, Latch, Write)

I. Enable gesture changes on the user parameter (PluginEditor.cpp)

```
    Look for the sliderValueChanged function to add our new lines of code

    void StarterCodeAudioProcessorEditor::sliderValueChanged(Slider* slider)
    {
        if (slider == &volume) {
        //notify HOST for plugin automation (touch, latch, etc)
        audioProcessor.volumeUParam->beginChangeGesture(); //adds automation
capability!
        *audioProcessor.volumeUParam = (float) volume.getValue();
        audioProcessor.volumeUParam->endChangeGesture();  //adds automation
capability!
        }
    }
```

```
note: JUCE enables you to manually write automation onto user parameters; however, we
have yet to add code to enable the recording of automation using Touch, Latch, and
Write automation modes. To enable these automation modes we have to wrap our
SliderValueChanged params with two lines of code.
```

9 GUI and Other Ideas

I. Start to notice the design of your plugin. Where are patterns in code that might address the look and feel of your plugin? How do you think you may alter them?

Version notes:

This walkthrough document has been tested in JUCE 6.0 and JUCE 6.0.8

Author:

Jon Bellona, https://jpbellona.com