Original version by Jorge Herrera (jorgeh@ccrma.stanford.edu), April 2012

1 VST version

In this class we'll be using VST 2.4 (not the latest VST 3.x) mainly because most VST hosts handle VST2, but VST3 support is still not adopted by some hosts.

If you are curious about the new features included in VST3 you can check http://www.steinberg.net/en/company/technologies/vst3.html

2 VST Host

To test/run your VSTs you'll need a *VST Host*. There are many commercial and open source hosts. Some of them are cross-platform (e.g. Audacity), but most are platform specific.

To use VSTs in Audacity you will probably have to install the VST enabler (http://web.audacityteam.org/vst/). Although Audacity is a good alternative, you should be aware of a big limitation: when using VSTs, it doesn't allow for real-time parameter changes. You can perfectly use it for this class, but the option of real-time control can be very useful, specially when debugging your VST.

Another cross-platform solution is Ardour. Through Audacity's VST enabler it allows to run 32-bit VSTs (I've tested it in OS X 10.7 and works really nicely). Some pointers on how to use VST with Ardour can be found at http://ardour.org/node/3849

Here are some other hosts you can use:

- Linux
  - Qtractor
  - JOST
- OS X
  - Ableton Live
- Windows
  - Cubase
  - Nuendo
  - Ableton Live

A more comprehensive list can be found at http://en.wikipedia.org/wiki/Virtual_Studio_Technology#VST_hosts

3 VST SDK

Because of licensing issues, we are not allowed to distribute the VST SDK. You'll have to download it yourself. To do so, you should create a free account at Steinberg's website (`http://www.steinberg.net/nc/en/company/developer/sdk_download_portal.html`) and download the VST 2.4 SDK to your computer.

4 VST architecture

4.1 Sample Code

In 4.1.1 and 4.1.2 the two files that you'll need to modify to implement different VSTs.

4.1.1 Header File

```
//-------------------------------------------------------------------------------
// VST Effect Plug-in
//
// Filename     : Gain.h
// Created by   : jorgeh@ccrma.stanford.edu
// Company      : CCRMA
// Description  :
//
// Date         : April, 2012
//-------------------------------------------------------------------------------

#ifndef __Gain__
#define __Gain__

#include "public.sdk/source/vst2.x/audioeffectx.h"


enum
{
  // Parameters Tags
  kGain = 0,
  kPan,

  kNumParams
};


//-------------------------------------------------------------------------------
class Gain : public AudioEffectX
{
public:
  Gain(audioMasterCallback audioMaster);
  ~Gain();

  // Processing
  virtual void processReplacing( float** inputs,
                                 float** outputs,
                                 VstInt32 sampleFrames);
//  virtual void processDoubleReplacing ( double** inputs,
//                                        double** outputs,
//                                        VstInt32 sampleFrames);
```

```
42
43    // Program
44    virtual void setProgramName(char* name);
45    virtual void getProgramName(char* name);
46
47    // Parameters
48    virtual void setParameter(VstInt32 index, float value);
49    virtual float getParameter(VstInt32 index);
50    virtual void getParameterLabel(VstInt32 index, char* label);
51    virtual void getParameterDisplay(VstInt32 index, char* text);
52    virtual void getParameterName(VstInt32 index, char* text);
53
54      // Effect basics
55    virtual bool getEffectName(char* name);
56    virtual bool getVendorString(char* text);
57    virtual bool getProductString(char* text);
58    virtual VstInt32 getVendorVersion();
59
60    // VST type (effect, synth, analysis, etc.) see aeffectx.h for other options
61    virtual VstPlugCategory getPlugCategory() { return kPlugCategEffect; }
62
63  protected:
64
65      // Actual Parameters
66    float fGain;
67    float fPan;
68
69      // Programs (settings)
70    char programName[kVstMaxProgNameLen + 1];
71  };
72
73  #endif
```

Listing 1: "Gain.h"

All the definitions in the header file are important, but most of them are pretty much the same for any VST you implement, at least in this class. The main changes you'll need to make are the parameter "definition" (`enum` at the top) and the internal attributes to keep track of the values of the parameters needed by your effect (int his case `fGain` and `fPan`), an possibly other state variables that are invisible to the user, but required in the implementation.

Note that in the presented header file, there's a commented out method definition (`processDoubleReplacing`). If you want to process the audio data using double floating precission you should uncomment this definition and implement it in the `.cpp` file.

4.1.2 Implementation file

```
1  //-------------------------------------------------------------------------------
2  // VST Effect Plug-in
3  //
4  // Filename     : Gain.cpp
5  // Created by   : jorgeh@ccrma.stanford.edu
6  // Company      : CCRMA
7  // Description  :
8  //
9  // Date         : April, 2012
10 //-------------------------------------------------------------------------------
```

```
11
12 #include "Gain.h"
13
14 //-------------------------------------------------------------------------------
15 AudioEffect* createEffectInstance(audioMasterCallback audioMaster)
16 {
17   return new Gain (audioMaster);
18 }
19
20 //-------------------------------------------------------------------------------
21 Gain::Gain(audioMasterCallback audioMaster)
22 : AudioEffectX (audioMaster, 1, 2)  // 1 program, 1 parameter only
23 {
24   setNumInputs(2);         // stereo in
25   setNumOutputs(2);        // stereo out
26   setUniqueID('Gain');     // Unique identifier.
27                              // You must get this from Steinberg's site:
28                              // http://service.steinberg.de/databases/plugin.nsf/plugIn?
                                   openForm)
29   canProcessReplacing();  // supports replacing output
30
31   fGain = 1.f;             // default to 0 dB
32   fPan = 0.5f;
33   vst_strncpy (programName, "Default", kVstMaxProgNameLen); // default program name
34 }
35
36 //-------------------------------------------------------------------------------
37 Gain::~Gain ()
38 {
39   // nothing to do here
40 }
41
42 //-------------------------------------------------------------------------------
43 void Gain::setProgramName(char* name)
44 {
45   vst_strncpy (programName, name, kVstMaxProgNameLen);
46 }
47
48 //-------------------------------------------------------------------------------
49 void Gain::getProgramName(char* name)
50 {
51   vst_strncpy (name, programName, kVstMaxProgNameLen);
52 }
53
54 //-------------------------------------------------------------------------------
55 void Gain::setParameter(VstInt32 index, float value)
56 {
57   switch (index)
58   {
59     case kGain:
60             fGain = value;
61             break;
62     case kPan:
63             fPan = value;
64             break;
65   }
66 }
67
68 //-------------------------------------------------------------------------------
69 float Gain::getParameter(VstInt32 index)
70 {
71   switch (index)
72   {
73     case kGain:
```

```
74              return fGain;
75              break;
76      case kPan:
77              return fPan;
78              break;
79    }
80      return 0;
81  }
82
83  //------------------------------------------------------------------------
84  void Gain::getParameterName(VstInt32 index, char* label)
85  {
86    switch (index)
87    {
88      case kGain:
89              vst_strncpy (label, "Gain", kVstMaxParamStrLen);
90              break;
91      case kPan:
92              vst_strncpy (label, "Pan", kVstMaxParamStrLen);
93              break;
94    }
95  }
96
97  //------------------------------------------------------------------------
98  void Gain::getParameterDisplay(VstInt32 index, char* text)
99  {
100   switch (index)
101   {
102     case kGain:
103             dB2string (fGain, text, kVstMaxParamStrLen);
104             break;
105     case kPan:
106             float2string(2.0*fPan-1.0, text, kVstMaxParamStrLen);
107             break;
108   }
109 }
110
111 //------------------------------------------------------------------------
112 void Gain::getParameterLabel(VstInt32 index, char* label)
113 {
114   switch (index)
115   {
116     case kGain:
117             vst_strncpy (label, "dB", kVstMaxParamStrLen);
118             break;
119     case kPan:
120             vst_strncpy (label, " ", kVstMaxParamStrLen);
121             break;
122   }
123 }
124
125 //------------------------------------------------------------------------
126 bool Gain::getEffectName(char* name)
127 {
128   vst_strncpy (name, "Gain", kVstMaxEffectNameLen);
129   return true;
130 }
131
132 //------------------------------------------------------------------------
133 bool Gain::getProductString(char* text)
134 {
135   vst_strncpy (text, "Gain", kVstMaxProductStrLen);
136   return true;
137 }
```

```
138
139 //-----------------------------------------------------------------------
140 bool Gain::getVendorString(char* text)
141 {
142   vst_strncpy (text, "CCRMA", kVstMaxVendorStrLen);
143   return true;
144 }
145
146 //-----------------------------------------------------------------------------
147 VstInt32 Gain::getVendorVersion()
148 {
149   return 1000;
150 }
151
152 //-----------------------------------------------------------------------------
153 void Gain::processReplacing(float** inputs, float** outputs, VstInt32 sampleFrames)
154 {
155     float* in1  =  inputs[0];
156     float* in2  =  inputs[1];
157     float* out1 = outputs[0];
158     float* out2 = outputs[1];
159
160     float left = 1.0;
161     float right = 1.0;
162
163     if (fPan < 0.5) {
164         right = 2.0*fPan;
165     } else {
166         left = 2.0*(1.0 - fPan);
167     }
168
169     while (--sampleFrames >= 0)
170     {
171         (*out1++) = (*in1++) * fGain * left;
172         (*out2++) = (*in2++) * fGain * right;
173     }
174 }
```

Listing 2: "Gain.cpp"

In the implementation file, the bulk of the work is done in the `processReplacing()` method (or `processDobleReplacing()` if you want to use double floating point). This method takes an buffer of input samples (`inputs`) and outputs the processed buffer (`outputs`).

Other methods you might need to change are:

`Gain()`	class constructor (note this name will change to match the name of the class). Here you should set the initial values for the parameters and state variables of your effect.
`~Gain()`	class destructor. If you use pointers in your class, don't forget to delete them here
`setParameter()`	this is the callback the gets called when the user changes a parameter value in the GUI

`getParameter()`	get the actual value of a parameter (we won't use this much)
`getParameterName()`	the name of the parameter to be displayed in the GUI
`getParameterLabel()`	the units of the parameter to be displayed in the GUI
`getParameterDisplay()`	the parameter value to be presented to the user in the GUI.
`getEffectName()`	some hosts will display this name as your plug-in's name
`getProductString()`	other hosts will display this name as your plug-in's name
`getVendorString()`	some hosts will also display this name as part of the plug-in's name

4.1.3 Knob to value

In general your parameters can be in any arbitrary range. On the other hand, VST knob values will always be in the [0,1] range. You will need to take this into consideration in the `setParameter()` callback. Furthermore, the VST knob always uses a linear scale in the [0,1] range, but sometimes you will want other scales (e.g. logarithmic, squared, to even discrete steps!). You should also consider this when designing/implementing you VSTs.

5 VST GUI

We won't deal with custom GUIs for your VSTs in this class. By default, all the exposed parameters are presented as sliders (faders) to the user. Some hosts may allow other representations (e.g. Ableton Live allows you to pick 2 parameters to become the axis of an XY pad).

You are welcome to try GUIs for your VSTs, but please refrain from including them in your submission, as they might not run in the grader's environment.

6 Debugging a VST

Debugging VST plugins is not easy, as there's a VST host in the middle of the debugging process. While it is possible to debug them in a traditional way (see Section 6.1 for example), there are alternative methods that can be used.

Analogously to the "print out" method in traditional programming, is possible to use the `ProcessReplacing()` outputs as "audio-data print outs". In other words, instead of outputting the actual processed sound is possible to output the value of an internal variable at audio-rate, which then will be plotted by the VST host after processing the input file with your plug-in. Of course this means that you won't be able to necessarily

listen the processed file, but is a *quick-and-dirty* way of checking that the intermediate steps are working as intended.

6.1 Debugging with Xcode

[The following instructions only work in XCode. Other IDE's and OSs might be able to do similar things, but I haven't even tried, as I do all my VST development in XCode]

XCode allows to run an external executable as part of the Run (and Debug) actions. This allows to open a VST host and debug the plug-in while the host is running (you will be able to set breakpoints, etc.)

6.1.1 XCode 3.x

To do that, you will have to add a New Custom Executable (in the Executables branch on the project tree on the left in XCode 3). Specify the VST host (or DAW) you want to use to debug your plug-in.

With this in place, when you Build & Debug the specified VST host will be launched and you can instantiate you plug-in. Then you can, for example, set a break point in the `ProcessReplacing()` method and the execution of the host will be interrupted as soon as you start to play a track that's using your plug-in.

6.1.2 XCode 4.x

The idea is the same as 6.1.1, but in Xcode 4 the executable to be launched when debugging is setup differently. To specify it, using the menu bar go to the `Product` → `Edit Scheme`. Select `Run (Debug)` on the list on the left. Select the `Info` tab and choose the executable you would like to be launched when debugging your VST.

6.1.3 Independently of the XCode version you use

In order for this to work, you should be using the debug version of the compiled plug-in, and not the one that's copied into you `~/Library/Audio/Plug-Ins/VST/` folder. It turns out that Ableton Live has an easy way to specify custom folders to look for VSTs, so you can setup your project's Debug folder as this custom VST folder in Ableton Live. Even though Ableton Live is a commercial app, if you don't own it you can download a demo that will allow you to debug your VSTs (that's what I did). The demo has the limitation that you can't save files, but you can still use and debug your VSTs.

Alternatively, if you don't have/like Able Live, you could change your project settings to compile the debug version in the Plug-Ins folder (so any VST host can "see" it), but I haven't done it myself, so I don't know how :(

7 Changing your VST name

In Xcode, this is a 2 step process.

1. FIrst, you'll need to change the Target name.

   XCode: go to the project's target and search for "Product Name" in the "Build Settings" tab; rename your plugin here. This will change the actual Bundle name that will be copied into the `/Library/Audio/Plug-Ins/VST/` folder.

   Linux: In the provided makefile, change the `TARGET` variable. This step is extremely important when you start modifying starter code and generating different version of your plug-in's, so they won't overwrite each other.

2. Indenpendantly from the OS you are working on, in the plug-in's `.cpp` file, change the plug-in's name inside the `getProductString()` and `getEffectName()` methods. This will change the name displayed by the VST host.