JUCE Audio Plug-Ins

Sean Enderby

Birmingham City University

7th December 2016

Audio Plug-Ins

What is a Plug-In?

- Plug-Ins add extra functionality to an existing application (the host application).
- Audio plug-ins are implemented as shared libraries (.dll, .so).
- These are pieces of compiled code which the host can load and execute.
- In order for the host to be able to execute the code a plug-in must implement a predefined interface.
- These interfaces are defined in the plug-in architecture's Application Programming Interface (API).

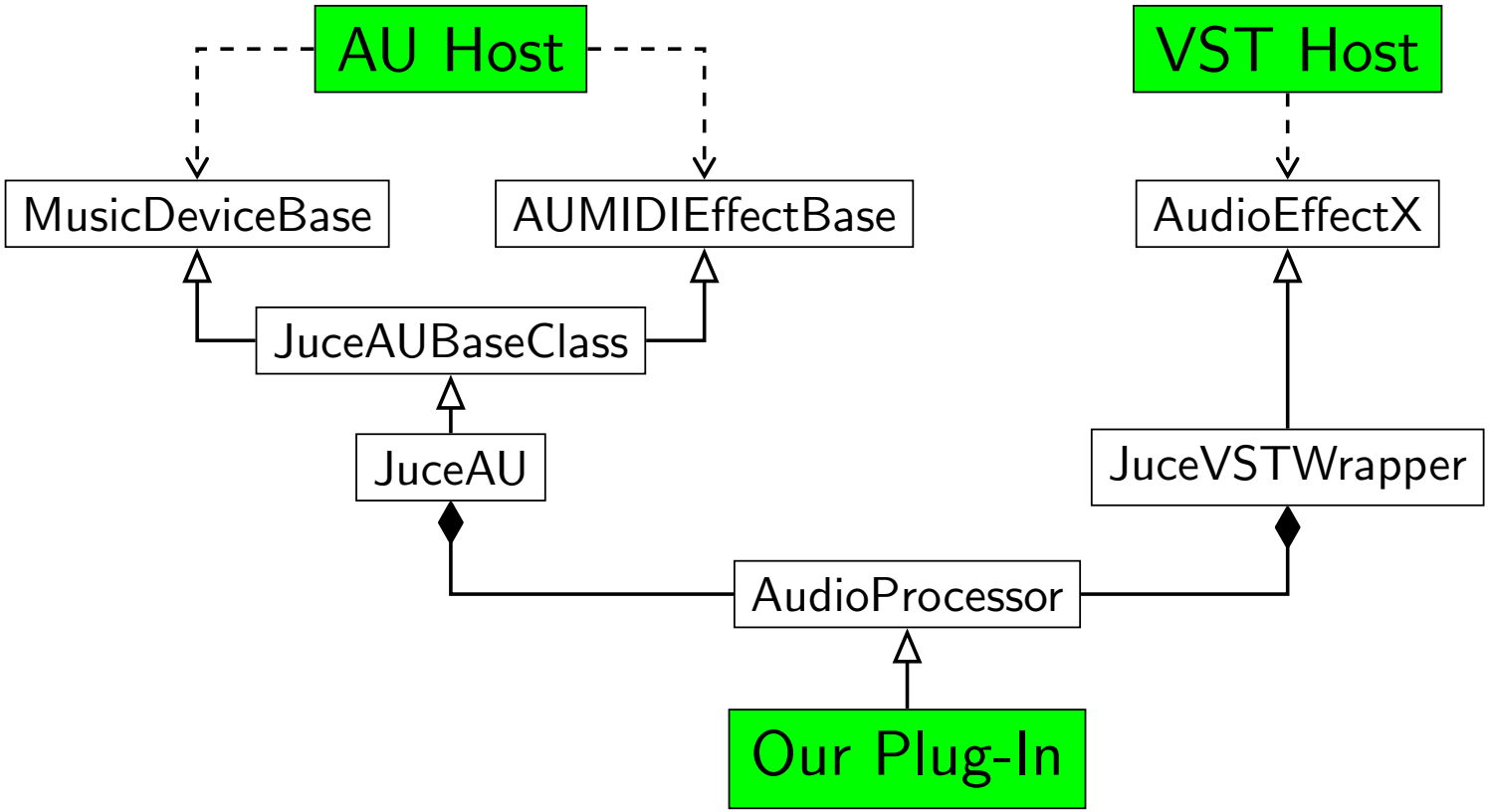
Plug-In APIs

- A plug-in API defines a set of classes and functions.
- When we make a plug-in we define a class that inherits one of these classes and implements the relevant functions.
- A host will know the names of these functions and so will be able to call them from the shared library.
- For example, to process a buffer of audio the host would call a VST's *processReplacing()* function or an AU's *ProcessBufferLists()* function.

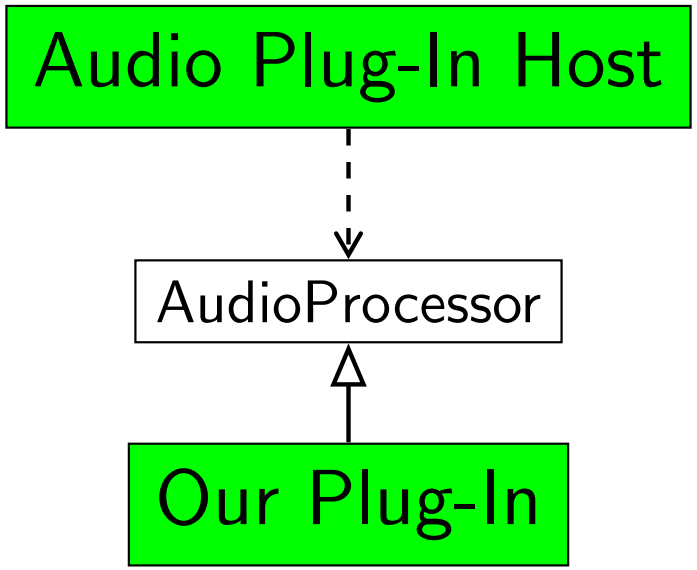
The AudioProcessor Class

- The names of these classes and functions are different for each plug-in architecture which makes writing for multiple formats tedious.
- JUCE to the rescue!
- When making a plug-in with JUCE we inherit the *AudioProcessor* class.
- There are then a series of wrapper classes which encapsulate an *AudioProcessor* object and make it look like a plug-in of a given architecture.

Class Hierarchy

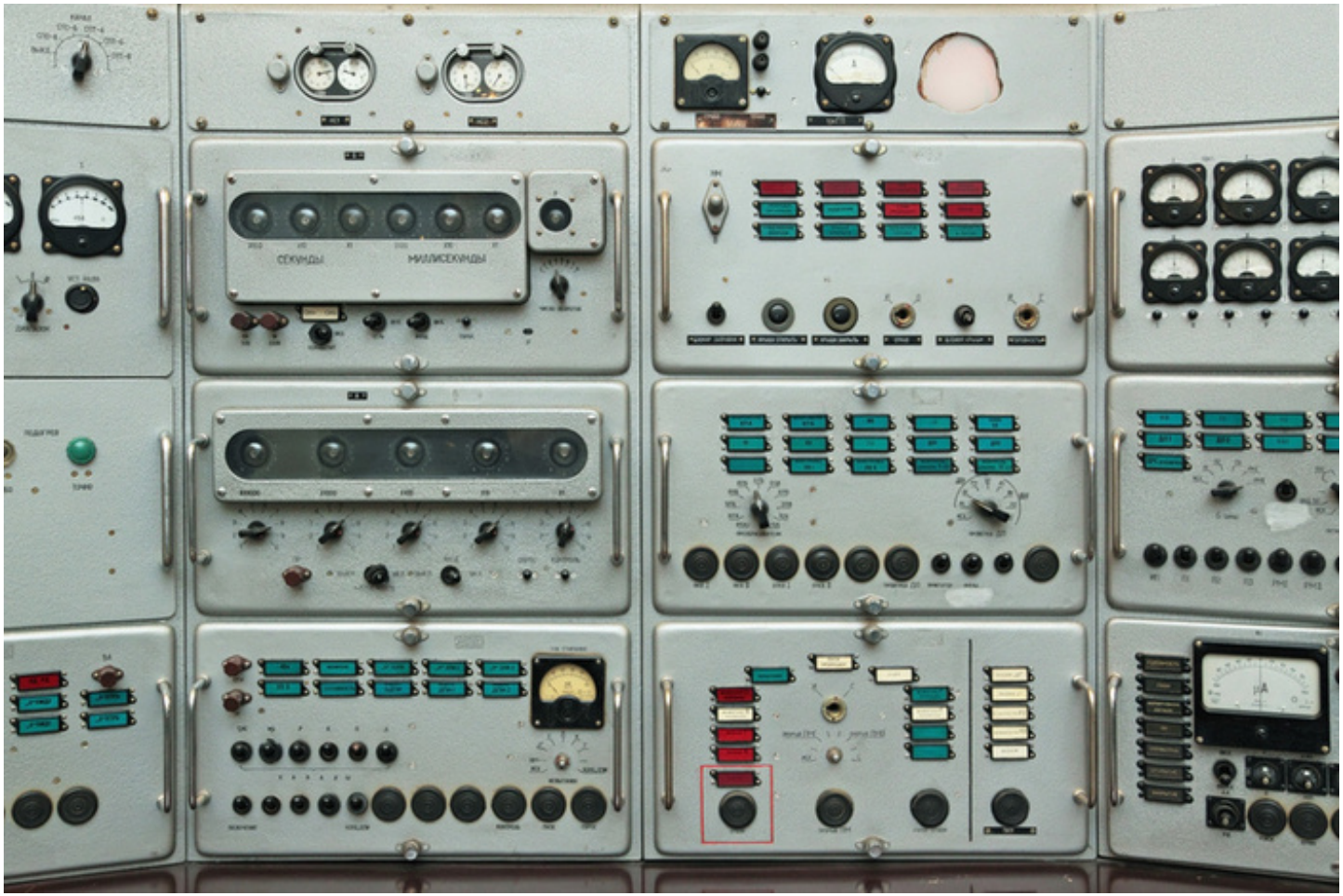


How We Need to Think About It



- JUCE uses witchcraft to hide all the nastyness.

The AudioProcessor Interface



Audio Plug-In Functionality

- An audio plug-in is responsible for a number of things:
  - Reporting information about itself to the host (its name, number of input and outputs and so on).
  - Providing some kind of audio / midi processing or synthesis (obviously).
  - Managing their own parameters.
  - Managing the loading and saving of presets.
  - Provide a GUI (optional).
- A lot of these can be automatically written by JUCE by changing project options in the Projucer.
- In this section we will look at the bits we always need to write.

Processing Functions

- Prior to starting its audio callback a host will inform plugins of the sampling frequency and buffer size it is using.
- As part of its audio callback it passes audio buffers to its active plugins for processing.
- When the audio callback is stopped the host will inform plug-ins of the fact.
- The JUCE *AudioProcessor* class provides the *prepareToPlay()*, *processBlock()* and *releaseResources()* functions in order to allow this.

Processing Functions Cont.

```
// This function tells the plug-in the audio sample rate being used
// by the host as well as an estimate of the buffer size that will
// be used in the processing. This function should be used to do any
// preparation the audio processing needs (allocating memory,
// calculating filter coefficients etc.).
void prepareToPlay (double sampleRate, int estimatedSamplesPerBlock);

// This is the function where the processing takes place. The host
// will pass in audio and MIDI buffers by reference for the
// plug-in to process.
void processBlock (AudioSampleBuffer &buffer, MidiBuffer &midiMessages);

// This function is used to free any resources that were allocated
// in prepareToPlay.
void releaseResources();
```

Managing Parameters

- Plug-Ins need to provide certain functionality with regards to parameters:
  - Functions to provide information about the parameters (total number, names, units).
  - Functions to query and set the values of parameters.
- Hosts use this functionality to provide interfaces by which a plug-in can be controlled without using its GUI (automation tracks for example).
- Parameter values are passed between the host and the plug-in as floats in the range 0.0 to 1.0. The plug-in must scale these for it's own use.

The AudioProcessorParameter Class

- JUCE provides a number of different ways to manage plug-in parameters.
- The one we will look at today is the *AudioProcessorParameter* class.
- JUCE provides several classes which inherit this and manage parameter functionality for different data types (*AudioParameterFloat* for example).
- These derived classes deal with the necessary conversion between the 0.0-1.0 values used by the host and the actual range of values used by the plug-in.

The AudioProcessorParameter Class Cont.

- An *AudioParameterFloat* can be used by declaring a pointer to one as a member of your plug-in.

```
AudioParameterFloat *gainParam;
```

- It is then added to the plug-in in its constructor using *addParameter()*.

```
addParameter (gainParam = new AudioParameterFloat ("gain", "Gain",
                                                    -20.0f, 20.0f, 0.0f);
```

- The parameter value can be accessed by dereferencing the pointer to it.

```
float parameterValue = *gainParam;
```

The AudioProcessorParameter Class Cont.

- The classes derived from *AudioProcessedParameter* provided in JUCE provide very basic parameter management.
- For more complex parameter management you will have to write your own parameter class.
- See this weeks example code for some examples of custom parameter classes.
- You could also look at the other ways of managing parameters in JUCE plug-ins by reading the documentation.

Thanks For Listening!

Any Questions?

Let's Make a Dank Plug-In!