

MIDI plugins with JUCE

EECS 4462 - Digital Audio

Click to edit Master text styles

Second level

Third level

F

Fifth level

September 25, 2018

What is a plugin?

- A plugin is a software component that adds functionality to an existing application
- Many applications support a plugin architecture
 - Digital Audio Workstations
 - Email clients
 - Browsers
- Main benefits
 - Third party developers can add functionality
 - The size of the main application is reduced
 - Adding new features becomes much easier

What is JUCE?

- A framework that allows for the development of cross-platform audio applications and audio plugins
- Allows the developer to focus on the processing they want to implement
- Supports both MIDI and audio plugins
- Used by many professional audio plugin developers
- Free for personal or educational use
- <https://juce.com/>

Assignment 1

- The goal of Assignment 1 is to develop a plugin that arpeggiates incoming MIDI notes
- We will use the Arpeggiator Tutorial plugin provided by JUCE as a starting point
- Download the tutorial and open ArpeggiatorTutorial.jucer
- Click on Modules and ensure that the paths point to where you installed JUCE
- Under Release, you can see the VST Binary location. This is the path that will contain the .dll file you will submit

Assignment 1

- Click Save Project and Open in IDE...
- Build
- Test with the Audio Plugin Host that the MIDI events get arpeggiated
- Let's look at the code
- The online tutorial explains the code line by line
 - Some additional information in the next few slides

Important class: AudioProcessor

- Base class for audio plugins
- Your plugin class must inherit from AudioProcessor

```
class Arpeggiator : public AudioProcessor  
{ ... }
```

- Must declare a global function called createPluginFilter() that returns an instance of your plugin

```
AudioProcessor* createPluginFilter()  
{  
    return new Arpeggiator();  
}
```

JUCE API

- JUCE provides an API for all its classes
 - See link under Assignment 1
- Check out **AudioProcessorParameter**
- Base class for all parameter types you might want to add to your plugin's GUI
 - **AudioParameterBool**
 - **AudioParameterChoice**
 - **AudioParameterFloat**
 - **AudioParameterInt**

C++ info

- Class constructors work similar to Java
 - Same name as the class
 - Can have overloaded versions
- C++ also has destructors
 - Run when an instance is destroyed
 - Same name as class with a ~ in front

```
~Arpeggiator() {}
```


C++ info

- Declaring an object in C++ is enough to create an object at run time

```
Arpeggiator arp;
```

- The above creates an Arpeggiator object
- Such an object gets destroyed automatically when out of scope
- To dynamically create objects, use pointers

```
Arpeggiator *arp;  
arp = new Arpeggiator();
```

C++ info

- C++ has no garbage collection
- You must delete dynamically created objects manually

```
Arpeggiator *arp;  
arp = new Arpeggiator();  
delete arp;
```

- This will call the destructor before releasing the memory
- **malloc**, **realloc**, **free** etc. can also be used for dynamic memory allocation

Important function: addParameter

- This function adds another parameter to your GUI that the user can affect in real time
- In the Arpeggiator Tutorial all you need is the call to addParameter
 - Since no **AudioProcessorEditor** is defined, JUCE uses a **GenericAudioProcessorEditor**
- More on GUIs next time

Buffer processing

- In JUCE, processing takes place in buffers
- For audio plugins, this buffer contains a number of audio samples (more in this in a week or two)
- For MIDI plugins, the buffer contains the MIDI events that took place since the last buffer
- Time information is based on the sample rate, even in the case of MIDI
- The duration of a buffer is
 $\text{Sample Rate} \times \text{Number of Samples in Buffer}$

Important function: prepareToPlay

- Called once before processing starts
- Can be used to initialize any variables in your plugin
- Also sets the Sample Rate

Important function: processBlock

- Called repeatedly
- All the processing (converting input to output) happens in its body
- Received an AudioBuffer and a MidiBuffer
 - Only one of them will contain data based on the type of the plugin
- Timing information is obtained from the AudioBuffer even in the case of a MIDI plugin

C++ info

- Static functions in C++ are similar to static methods in Java, but the syntax is a bit different

```
MidiMessage::noteOff (1, lastNoteValue)
```