CMLS Homework 2
Assignment 1 - Additive Synthesizer

Group 18
10751302 - 10531235 - 10314186 - 10703215

2nd Sem. A.A. 2019/2020

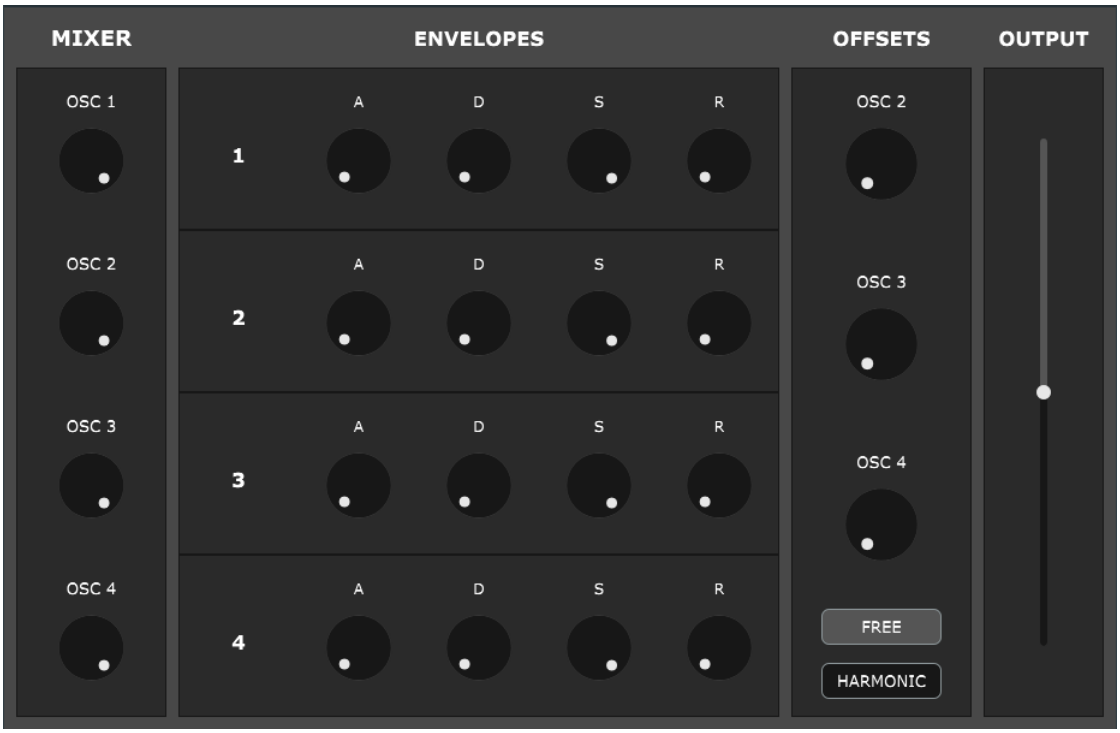
GitHub Repository

https://github.com/minettiandrea/hw2-cmls-synth

Introduction

In this report we will show and explain how we have structured and implemented our polyphonic synthesizer based on **Additive Synthesis**.

Plugin Overview



Interface of the plugin

As can be seen in the graphic interface the synthesizer can be subdivided into 4 main sections: the *Mixer* section, the *Envelopes* section, the *Offsets* section and the *Output* section.

- **MIXER**
  The Mixer section allows the user to control the gain of each oscillator independently from the others.
- **ENVELOPES**
  The Envelopes section allows the user to control the envelope (Attack, Decay, Sustain and Release) of each oscillator independently from the others.
- **OFFSETS**
  The Offsets section allows the user to apply a frequency offset to the oscillators 2, 3 and 4. Two types of offsets are possible (and can be activated by toggling the respective button):
  - **Free Offset**: The frequency offset can be set to any value between 0 Hz and 1000 Hz. This configuration allows the user to experiment with particular timbres (along with different envelopes and gains) to generate complex sounds.
  - **Harmonic Offset**: The frequency offset can be set to a multiple of the frequency corresponding to the note played by the Oscillator 1. In particular it can go from the fundamental (frequency of the first oscillator) up to the tenth harmonic. This configuration allows for a more classic and "organ-like" sound.
- **OUTPUT**
  The output section allows the user to control the main output gain.

Implementation

For the implementation, instead of using the Juce Synthesisers classes to manage the synthesizer functionalities we decided to create our owns:

- **OSCILLATOR CLASS**
  The Oscillator class is the basic building block of the synthesizer. In this class all the parameters of each oscillator and the wave processing are managed. Each oscillator is characterized by the *fundamental* (frequency corresponding to the note played) and an *offset* (that can be Free or Harmonic as explained in the previous section). In the processing phase the actual frequency of the oscillator is calculated as $fundamental + offset$. Since only the secondary oscillators (Oscillator 2, 3 and 4) can have their offset modified the main oscillator (Oscillator 1) will always play the fundamental.

  We also decided to split the volume control into 2 variables: the *amplitude*, that gets modified by the MIDI note played and thus depends on the velocity (more information can be found in the MIDI Management section below) and the *gain*, that is intended as the main gain of the oscillator and can be modified only by the Graphical User Interface dedicated control.

  Each oscillator contains also a *Juce ADSR* member, that manages the Attack-Delay-Sustain-Release of the respective oscillator. The parameters can be modified by the Graphical User Interface dedicated controls and their values are stored in the *Juce ADSR::Parameters* member.

  In addition to *getters* and *setters* for the main parameters the Oscillator class contains also the *play* and *stop* methods (which respectively trigger the attack and release phase of the envelope) and the *getBlockSineWave* method that is the sine wave processing block. In particular it returns the wave value at each sample (taking care of calculating the actual frequency based on the offset applied).
- **SYNTH CLASS**
  As the name suggest the Synth class is the main synthesizer class. It contains the four *Oscillator* and a reference to the *State*, that stores all the parameters values (more information below).

  The methods are mostly utility methods to set the different oscillators parameters except for the *process* method, that is the main processing block. It basically retrieves the value of the four sine waves at

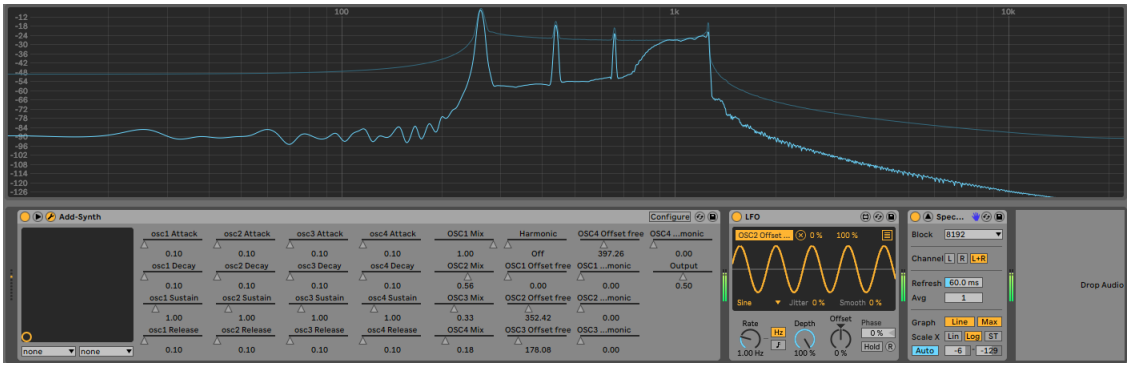
each sample (through the *getBlockSineWave* method) and plays the resulting output on the Left and Right channels (accordingly to the main output gain value).

In order to make the Synth polyphonic each time a note is pressed a new *Synth* is created and after all the oscillators have stopped playing (their envelope release phase has ended) the *Synth* gets destroyed to free up resources.

- **STATE CLASS**
  The role of the state classes *State*, *EnvelopeState*, *MixerState*, *OffsetState* is to keep stored the general status of the *Synth*.

  Having a centralized state enables every *Synth* to get the current state of all the parameters.

  Moreover we implemented all the status parameters using subclasses of the Juce *AudioProcessorParameter* class and we added them all in the *AudioProcessor* using the *AudioProcessor::addParameter* method. By doing so Juce exposes all the parameter to a DAW environment so that they can be used for automations, mapping on MIDI controllers and so on.



Plugin controls from a DAW

- **PLUGIN PROCESSOR CLASS**
  Like in every Juce audio plug-in application *PluginProcessor* is the main processing class. In particular in the *prepareToPlay* method the sample rate is set and the state of the notes played is reset.

  Then in the *processBlock* the MIDI Management is taken care of and the sound processing is performed by calling the methods already explained above.

  Moreover, the *processBlock* method is also in charge to check if every note is still active, and if it is not the case it removes the *Synth* corresponding to the unused notes.

- **PLUGIN EDITOR CLASS**
  The *PluginEditor* class manages all the Graphical User Interface components. In particular we decided to split the GUI into four sections (see Plugin Overview), each of which contains the management of the respective parameters. In particular, when a parameter is changed in a section of the GUI its new value is set in the corresponding *State* class, so that all the *Synth* gets affected by the change.

MIDI Management

For the scope of the project we handled only three types of MIDI messages:

- **Note On**
  When a *note on* message is received we first check if a *Synth* dedicated to that note has already been initialized. If this is the case then the value of the *velocity* gets updated and for each oscillator the envelope is restarted taking the latest values from the *Store* class. Otherwise, if the *Synth* dedicated to

that note hasn't already been initialized we instantiate it and set the fundamental frequency and the velocity corresponding to the played note.
We store all the instantiated *Synth* in a map in the *AudioProcessor* class with the MIDI note number as key.

- **Note Off**
  When a *note off* message is received the corresponding *Synth* is selected from the map using the MIDI note number, and the envelope release phase gets triggered for all the ocillators.

- **Pitch wheel**
  When a *pitch wheel* message is received it starts by getting normalized to a value from -0.5 to 0.5 (the standard position is in the middle so we want the zero value in that position). After normalization the value gets scaled to a range from -25% to +25% of each active note and then gets summed to the note fundamental frequency to shift it.

Possible improvements

- **GUI response to an external change of parameters**
  Right now when the parameters of the *State* are changed from an external source (like a DAW for example) the GUI does not reflect those changes. For if a parameter gets mapped to a MIDI controller by changing the parameter from the controller the GUI does not show the change.

- **CPU usage**
  Even though he plugin is simple the CPU usage is pretty high so some optimization can be done.

- **Frequency offset algorithms**
  For the frequency offset we just implemented two simple algorithm: free and harmonic (integer multiples of the fundamental). In order to get more interesting sounds some others algorithm may be implemented (for example multiples of the harmonic with some detune).