OranJam

Lelio Casale, Marco Furio Colombo, Marco Muraro, Matteo Pettenò

10582124, 10537094, 10866647, 10868930

Abstract

Sound Synthesis refers to the possibility of generating sound from scratch, using electronic hardwares or softwares. The sound is produced according to some chosen parameters in function of the result that the musician wants to get.
A synthesizer is an electronic musical instrument which is able to generate different signals and then manipulate them to achieve a specific sound as a final result.
The Hammond Organ, released in 1935, was the first electronic instrument to become popular within the music industry and today it's considered a precursor to modern synths. The very first synthesizer was RCA's MARK II Sound Synthesizer, released in 1957. This instrument was able to read punch cards that controlled a synthesizer made of 750 vacuum tubes. Nevertheless, the dawn of modern synthesizers is considered 1964 to around the mid-70s. As a matter of facts, in those years engineer Robert Moog brought a real revolution among electronic instruments. This revolution started with the Moog synthesizer which was released in 1964. Today, different synthesis techniques can be exploited to generate different kinds of sounds. The most common ones are **subtractive synthesis**, **additive synthesis** and **frequency modulation synthesis** (FM synthesis).
The project presented in the following paper is the implementation of a subtractive synthesizer and its interface meant to allow for the user to control a different kind of parameters shaping the generated sound.

1 Introduction

1.1 Basic Principles of Subtractive Synthesis

Subtractive synthesis technique comes from the idea of generating a sound starting from an existing signal whose partials are attenuated by a filter to modify its timbre. The starting sound is often the result of different signals that are mixed together. Generally, the musician starts from sounds that are very rich in harmonics and exploits different kinds of filtering and tools that shape the spectrum of sound.
Subtractive Synthesizers are sound signals generators which can be controlled by a musician through a control panel, a keyboard or other devices. These synthesizers are equipped with one or more oscillators, also called VCO (Voltage Controlled Oscillator, with regard to analog synthesizers), that generate standard waveforms such as sine wave, saw tooth wave, square wave and triangle wave. Furthermore, a noise generator could be available to produce white noise or pink noise to obtain more interesting sounds. These sound waves are mixed together to get the starting signal to be filtered later. Sometimes frequencies are slightly different so that it's possible to obtain beating effect and more interesting and complex sounds.
Several kinds of filters could be available to be used for modifying the timbre of the sound according to the final result that the user wants to obtain: low-pass filter, high-pass filter, band-pass filter, band-reject filter, shelving filters (high, low, peak/notch shelving filters).
Normally a hardware synthesizer is provided with a keyboard or it can be connected to an external MIDI keyboard so that the oscillators are triggered every time a Note On MIDI message is sent by the keyboard to the synth itself. The generated sound is then modified in its amplitude by a so-called VCA (Voltage Controlled Amplitude) which defines attack, decay, release and sustain envelope phases. Subsequently, one or more filters are applied on the signal. In some cases, the cut-off frequency depends on the fundamental frequency of the note that has been played. This parameter is called key follow

because the cut-off frequency follows the frequency related to the key that has been pressed on the MIDI keyboard.

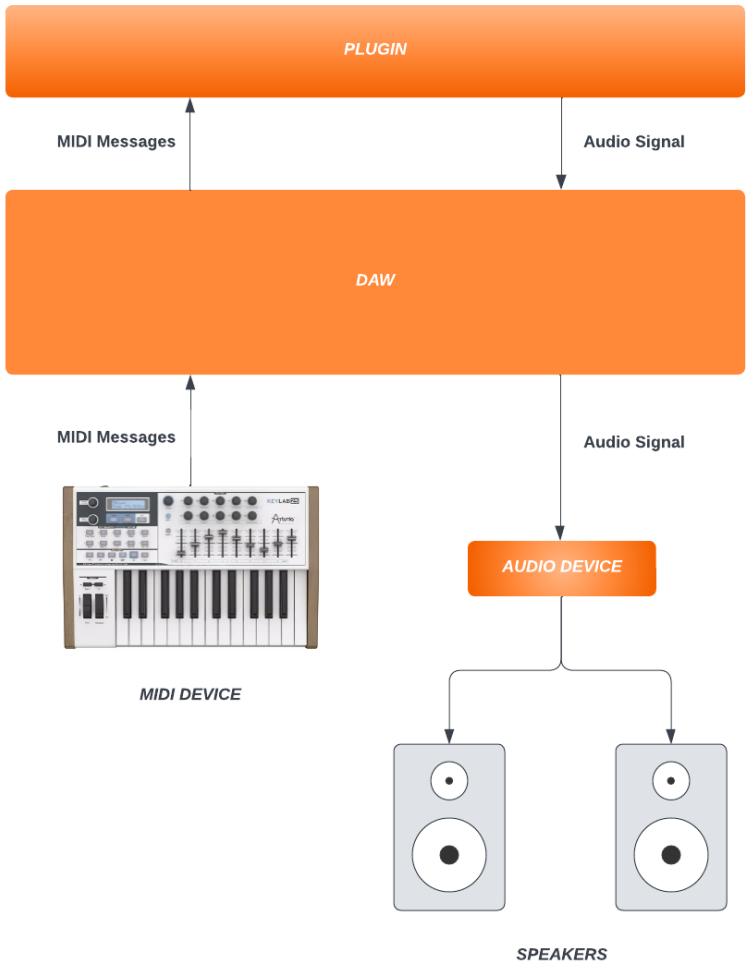

Figure 1: Scheme of interaction between MIDI inputs, DAW, Plugin and Audio Device.

The filter could also have its own free-standing envelope to be applied on cutoff frequency (VCF: Voltage Controlled Filter). Of course, Q factor could be controlled in the same way with a different envelope. As said before, filter's parameters can be controlled by the frequency or the amplitude of the note that has been played. Another control tool that could be present is the LFO (Low Frequency Oscillator) that provides the user with the possibility of producing a sort of vibrato effect on the final sound.
Subtractive synthesis allows the musician to play very interesting and bright sounds at a low computational cost. The user can easily choose the starting waveform, the type of filter to be applied and its parameters such as cutoff frequency and Q factor. Few and simple controls to generate complex and wonderful sounds.

1.2 Purposes

The project aims to develop an instrument plugin which implements a subtractive synthesizer. The software was implemented in C++ language with the aid of JUCE framework, a very powerful multi-platform and cross-standard library which supports different operating systems (MacOS, Windows, Linux) and plugin standards available on the market today (VST, VST3, Audio Unit, Audio Unit V3,

RTAS, AAX). The synth should provide the musician with the following primary features:

- different sound **generators**: a white noise generator and two or more **oscillators**. Oscillators can produce soundwaves of different nature, such as sine wave, saw tooth wave and square wave.
- **envelope** control. The user should be able to change attack, decay, sustain and release parameters of the starting sound that has been obtained by mixing together signals coming from noise generator and oscillators bank.
- different kinds of **filters** to be used for modifying the timbre of the sound according to the final result that the user wants to obtain. Three types of filters should be available: low-pass, high-pass and peak filter. For each of them the musician can control cut-off frequency and resonance. Moreover, it should be possible to control a gain parameter for the peak filter.
- an **LFO** (Low Frequency Oscillator) to be applied as a gain control on the starting signal for creating vibrato effect.
- a **Graphical User Interface (GUI)** to enable the user to control all the parameters specified above.

2 Implementation

2.1 Software Architecture

The file *SubtractiveSynth.jucer* allows the user to run the software and export it in an external IDE in order to actually build the code. The alternative building method is trought the usage of CMake and the needed files are included in the project as well.

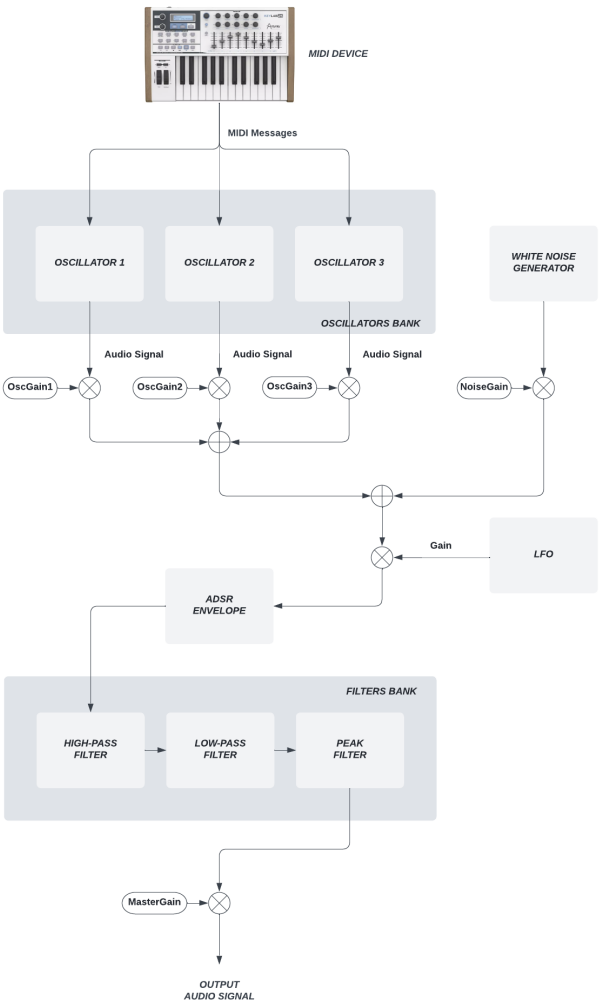

Figure 2: Signal Flow Diagram.

The software architecture is designed to be as modular as possible. Exploiting the Object-Oriented nature of the C++ programming language, the software is divided in many classes: each one implements a specific functionality of the plugin. The namespaces ***∼Generators***, ***∼Oscillators*** and ***∼Filters*** are external modules, included in the Modules package.

The class ***∼PluginProcessor*** creates the plugin and instantiates the oscillator voices needed for the Generators. It also prepares the parameters and sets up the graphical features of the plugin.

The class ***∼SubtractiveSynth*** contains the main core of the plugin. The method ***prepareToPlay()*** prepares all the parts of the plugin (oscillators, filters, LFO, gain knobs) to be ready to process the signal. The goal of the ***update()*** method is to interact with the GUI: thanks to this implementation, every time a knob or slider is changed by the user in the graphical part, the corresponding value is sent to the processor through the ***∼AudioProcessorValueTreeState*** and the parameters are updated in real time to provide the desired signal processing. The method ***renderNextBlock()*** implements the effective signal flow chain: first step are

the three oscillators. On the resulting signal a white noise and then the LFO and the envelope are applied, then the high-pass, low-pass and peak filters in sequence, and finally the master gain (*Figure 2*).
The class ***~ParametersManager*** initializes all the parameters of the subtractive synthesizer and sets all their value ranges thanks to the ***~juce::NormalisableRange*** object.
The namespace ***~Oscillators***, thanks to its subclass ***~MultiWavesOscillator***, generates the sinusoidal, square and saw original waves.
The class ***~Envelope***, placed inside ***~Generators***, implements the ADSR in the signal flow, and ***~WhiteNoise*** provides all the functionalities that are needed to generate the white noise, added afterwards to the waveforms.
The namespace ***~Filters*** has everything it needs to implement the filters, importing the ***~HPFilter***, ***~LPFilter*** and ***~PeakFilter*** classes.
The package *foleys_gui_magic* is imported in order to provide all the functionalities of the graphical user interface.

2.2 Audio Processing

2.2.1 Oscillators

The implemented synthesizer features an oscillators bank which consists of three different oscillators. The software provides the musician with the possibility of setting different waveforms (sine wave, saw tooth wave and square wave) according to the desired sound. For this purpose, an external module was implemented. As a matter of facts, the single oscillator is modeled through the ***~Oscillators::MultiWavesOscillator*** class.
MultiWavesOscillator::prepare() calls prepare method of ***~juce::dsp::Oscillator*** class which takes a ***~juce::dsp::ProcessSpec*** object as an input. This method allows to set sampleRate, samplesPerBlock and outputChannels parameters so that the oscillator is ready to play. ***MultiWavesOscillator::prepare()*** is called in ***SynthVoice::prepareToPlay()*** method where a processSpec object is instantiated. Then, a for cycle allows to call prepare method on each oscillator instance.
MultiWavesOscillator::process() is called within ***renderNextBlock()*** of ***~SynthVoice*** class. This method, on its turn, calls ***Oscillator::process()*** method of juce dsp module which processes the input and output buffers that are supplied in the processing context that is passed as an input.
Another very important method within this class is ***setWaveType()*** which allows to set the chosen waveform for the single oscillator. The musician can change wave type directly through the graphical user interface, which was implemented using the *foleys_gui_magic* module. Then, ***updateParameters()*** method enables updating of waveType, frequency and gain of the single oscillator. This method is called within ***SynthVoice::update()*** that updates all the processing parameters.

2.2.2 Generators

Moreover, the synth features a **white noise generator**, an **ADSR envelope** and a **LFO**. For this reason, an external module, called ***~Generators***, was implemented including ***~WhiteNoise***, ***~Envelope*** and ***~LFO*** classes. ***~Generators::WhiteNoise*** class models a white noise generator.

- ***process()*** method is called within ***SynthVoice::renderNextBlock()*** method and exploits ***~juce::Random*** class to generate random-valued samples to be inserted within the audio buffer that is passed as an input.
- ***updateParameters()*** method simply updates the gain that is set by the musician through the GUI.

~Generators::Envelope class implements the envelope which is provided by the synthesizer. It inherits from ***~juce::ADSR*** class that needs a ***~juce::ADSR::Parameters*** object as a member.

- ***prepare()*** method is called within ***prepareToPlay()*** of ***~SynthVoice*** class. This method resets the envelope to an idle state and then sets the sampleRate.

- ***updateParameters()*** simply updates attack, decay, sustain and release parameters. Then, ***setParameters()*** of ***~juce::ADSR*** class is called so that the ***~juce::ADSR::Parameters*** object is updated as well. Before updating ADSR parameters, ***Generators::Envelope::prepare()*** method must have been called. As a matter of facts, ***setParameters()*** needs to be called after the sampleRate is updated, otherwise the processed values could be incorrect.

- ***process()*** is called within ***~SynthVoice::renderNextBlock*** and, on its turn, calls ***applyEnvelopeToBuffer()*** of ***~juce::ADSR*** class so that the envelope is applied to the audioBuffer that is passed as an input.

~Generators::LFO class models a Low Frequency Oscillator (LFO). The LFO is implemented through a ***~juce::dsp::Oscillator*** object. Similarly to other classes described before, ***prepare()*** sets the sampleRate, samplesPerBlock and outputChannels of the oscillator. Then, ***process()*** method computes the LFO sample by sample. Subsequently, the LFO is applied to the whole input audio buffer as a gain. ***updateParameters()*** enables to initialize the oscillator and set its frequency and amount.

2.2.3 Filters

Another important part of the synthesizer is the **filters** block, which provides the user with the possibility of cutting some frequencies and amplifying others. This is made thanks to the three filters implemented: an **high-pass**, a **low-pass** and a **peak** filter. All these filters are infinite impulse response filters and are implemented in the same way through the ***makeLowPass()***, ***makeHighPass()*** or ***makePeakFilter()*** methods provided by juce in the ***~IIR::Coefficients*** class included in the dsp module. The choice of infinite impulse response filters was made in order to maintain the signal as much analog as possible: IIR filters derive from analog, while FIR filters have no analog history.
The high-pass and low-pass filters have two parameters: the cutoff frequency and a resonance value, also called Q factor. On the other hand, the peak filter has three parameters: the peak frequency, the resonance or Q factor value and the gain which the filter provides the peak frequency with.
Each filter type has its own juce class with three main methods:

- the ***prepare()*** method is called in the ***preparetoPlay()*** method of the ***~SubtractiveSynth*** class and initializes the filter to be ready for its functionalities.

- the ***process()*** method is called in the ***renderNextBlock()*** method of the ***~SubtractiveSynth*** class and implements the signal filtering in the process chain.

- the ***updateParameters()*** method is called in the ***update()*** method of the ***~SubtractiveSynth*** class and sends the commands from the graphical user interface to the plugin processor.

3 Graphical User Interface (GUI)

The *juce* framework offers the chance to create a custom Graphical User Interface (GUI) that implements and connects all parameters of interest in a designed component called the *PluginEditor* basically from scratch. However, during the preliminary phase of design of the synthesizer, we found an external library called *foleys_gui_magic* (GitHub repository), that would allow us to have a better workflow and ultimately a more pleasing GUI. For this reason, we decided to use it.

3.1 foleys_gui_magic

We can essentially break what this library does down to two aspects: creates a CSS cascading stylesheet in an *xml* file to define rules for the appearance of the GUI and fetches the parameters from the ***~AudioProcessorValueTreeState*** matching them with the GUI elements of choice. This allows the ***~PluginEditor*** to be substituted with the ***~MagicPluginEditor***, generated by the library itself.
Besides allowing us to avoid writing a lot of boilerplate code and to not reinvent the wheel when making the interface, *foleys_gui_magic* helped us with our workflow. It made possible preparing all the components and linking them to the interface before actually creating one and therefore we were able to create an effective graphical design when we already had a clear view of all the components

we prepared without issues. The *xml* file that defines the rules that model the layout has also proven really useful, giving an alternative method to modify views in a CSS-like fashion, granting additional flexibility to the framework.

The inclusion of *foleys_gui_magic* is just as simple as the one of any other module and can be done both using the **projucer** app or **CMake**. Of course in the deployed product none of that is necessary, since the inclusion has already happened in build. Once the *foleys_gui_magic* is added to the project, the usage is rather simple: it offers both a graphical way of modifying the plugin layout and the said *xml*.
Understanding how this module works has proven to be quite a challenge, due to the minimal documentation, but was quite interesting for us in the end. Its work pattern is to create a **"Magic"** extension of the *juce* components of interest, expanding their functionalities or entirely substituting them. What allows parameter refreshing is the ***∼MagicProcessorState***, which is linked to the ***∼AudioProcessorValueTreeState*** and listens to its changes, feeding them to the ***∼MagicProcessor***. Said processor is created by a dedicated builder using the *xml*-specified layout and the designed connections, such as the ones for spectral visualization.

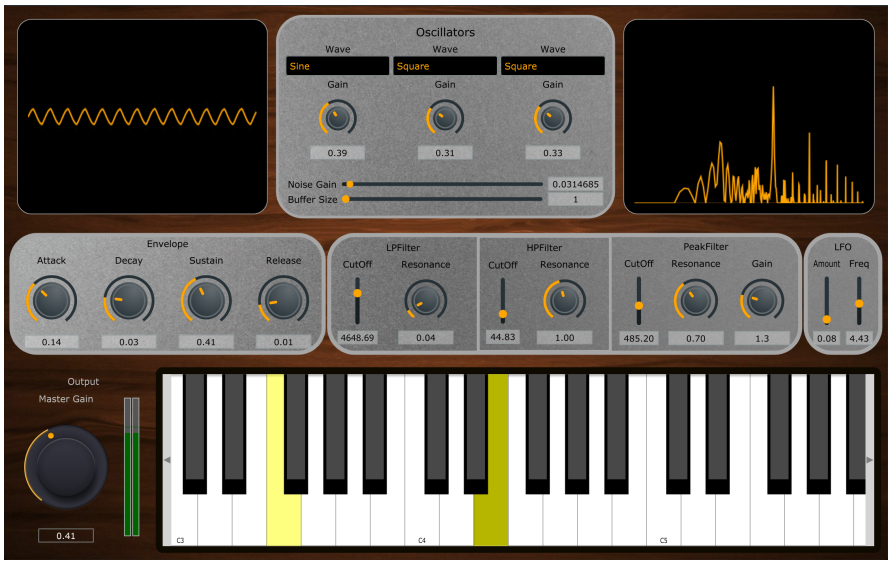

Figure 3: SubSynth Graphic User Interface

3.2 Controls

The available controls are graphically divided in six groups:

- The **Keyboard** generates midi signals, allowing the user to have a ready to play virtual instrument. This is good for testing it and experimenting with it, but of course, the plugin also works with an external midi generator or an external midi keyboard, which is recommended.
- The **Output** group, composed by
  - The **Master Gain**, a knob at the end of the processing chain, used to control the overall output volume of the synthesizer.
  - The **Meter**, which shows the overall output level.
- The **Oscillators** group, composed by
  - Three **Wave** selectors, one for each oscillator, that allow switching between Sinusoidal, Saw and Square waveform and the relative **Gain** knob, controlling the corresponding Volume.

  - The **Noise Gain**, controlling the level of a random process that is added in cascade after the signal created by the oscillators and conceptually acting as an additional oscillator as well.

- The **Envelope** group, composed by **Attack, Decay, Sustain** and **Release** knobs, allows the user to shape the amplitude of the played note in four moments linear regions with a linear behavior.

- The **Filters** group, composed by

  - A **Low-pass Filter** that applies low-pass filtering to the signal, offering the user control on the **CutOff** frequency and **Resonance**, which boosts the signal around the frequency of **CutOff** by the selected amount.
  - An **High-pass Filter** that applies high-pass filtering to the signal, offering the user control on the **CutOff** frequency and Resonance, which boosts the signal around the frequency of Cutoff by the selected amount.
  - A **Peak Filter** that applies a *peak filter* to the signal, offering the user control on the **CutOff** frequency, **Gain**, which boosts the signal around the frequency of Cutoff by the selected amount following a bell-shaped function and **Resonance** which determines the said bell's width.

- The **LFO** group, that controls a Low Frequency Sinusoidal Oscillator, which modulates the volume of the three oscillators and the Noise, through the available knob control

  - The **Amount** knob, which is the amplitude of the generated sinusoid.
  - The **Frequency** knob, dictating the frequency of the oscillator.

4 Conclusions

4.1 Results

In conclusion, the result of our implementation is a versatile subtractive synthesizer able to generate a wide range of sounds and ready to be imported in a Digital Audio Workstation (DAW).
Software maintenance and future improvements are made possible thanks to the **modular** architecture adopted. Adding new features only requires the files of the new classes in the modules and, after calling the relative methods of initialization, updating and processing of these classes, new filters or other features can be easily added in the GUI and will be fully functional. The graphical user interface is also completely customizable both by modifying the *magic.xml* file or with the dedicated graphic tool offered by *foleys_gui_magic*, that can be activated straight from Projucer.

4.2 Future Improvements

Even if we think that the synthesizer we implemented could generate a very big variety of sounds, and for that reason we are quite satisfied with that, the audio plugin could still be expanded both on the graphics and its functionalities.
On the graphical user interface a more convenient filter box could be implemented, with the possibility of moving the cutoff and peak frequencies of the filters directly on the frequency domain.
Talking about the functionalities, a future goal is to implement an ADSR and a few more LFOs to control automatically the parameters of the filters, especially the frequency of the peak filter, which, controlled automatically, could generate many types of sounds similar to wind and instruments like oboe or flute. Also the variety of filters can be enlarged, adding for example notch filters or second order low-pass or high-pass filters, and maybe a distortion effect could be a good add.