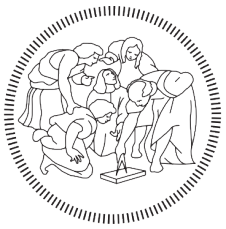

POLITECNICO
MILANO 1863

MSc. Music and Acoustic Engineering

Computer Music - Languages and Systems

Homework 2
Additive Synthesis

Authors:
10504151
10813836
10426992
10754748
[Group 16]

Link to the repository:
https://github.com/IronZack95/CMLS_Second_Project

1 Our Project

1.1 Additive Synthesis

At its most basic, additive synthesis is a sound generation method that involves the combination of simple waveforms. This is in order to produce a more complex sound.

Like in FM synthesis, sine waves are the most commonly utilized waveforms (as they're the simplest and most basic to work with). In practice however, most any type of waveform can be used, and some systems even utilize short segments of recorded sound.
The individual waveforms used in additive synthesis typically have different frequencies and amplitudes. When combined, these differences affect the length and tone of the final sound to varying degrees.
The additive synthesis consider sine waves. They essentially have only a specific frequency and amplitude. But when combined with other sine waves of different frequencies and amplitudes, they can produce radically different sounds, with the combined sound taking on a wildly different character.

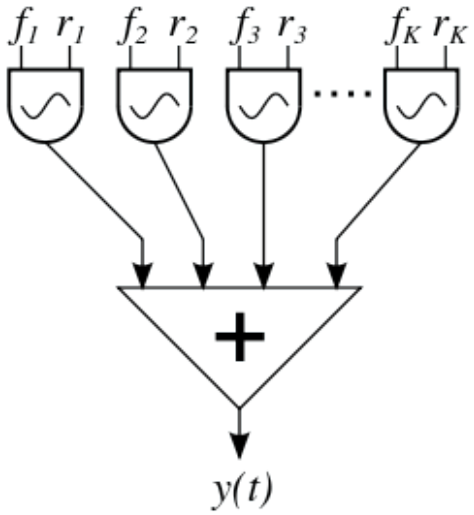

Figure 1: Additive Synthesis Schematic

In our plugin we implemented 4 oscillators and an ADSR section (**A**ttack - **D**ecay - **S**ustain - **R**elease) to control the sound envelope. Frequency and amplitude of the first three oscillators are manually set by the user, while the frequency of fourth one is controlled by a MIDI input (i.e. notes played on an external MIDI keyboard).
We also implemented polyphony to allow the user to play more than one note at the time (for example in a chord).

1.2 Our Additive Synthesis

The first problem we had to deal with is how to obtain an additive synthesis in JUCE. We started designing 4 oscillators. Our goal was to allow the user to play a MIDI keyboard (or a general MIDI input) and to associate a frequency reproduced from the oscillators for

each note played.
The first oscillator is directly controlled by the note played on the MIDI input and it directly reproduces the related frequency associated to that note, while the other 3 oscillators reproduces the same frequency of the played note but they also allow the user to modify that frequency in a certain range with a specific knob.
In order to sum all the 4 signals of the oscillators as in the Figure 1 schematic, we used the **JUCE audio buffer** and we summed every signal there.

1.3 Front-end and Back-end

Any software application (be it a web app, a mobile app or a computer program) is composed of two distinct parts: the front-end and the back-end. In the first one the developer programs everything that the user will interact with, while in the latter, which can not be seen by the user, the "engine" of the application is developed.
Usually, while programming, it could be useful to distinguish these two concepts also in the code, creating different files for the front-end and the back-end.
Providing the Processor files and the Editor files, JUCE's intent is exactly that of separating the front-end code and the back-end code. This sharp separation of the code could be made with any programming language, but the Object Oriented nature of the C++ eases it.
The main reason is that the main concept of C++ is the class, each class has specific functions with specific duties; thus, it is not hard to think of having classes specifically created for the functioning of the front-end and other classes for the back-end.
While drafting the code, this concept of separation has always been kept in mind.
In fact it has been taken beyond the front-end – ¬back-end duality, being applied also to classes and objects with generic different duties. This work organization has led to the creation of files other than those provided by JUCE, each of which will be discussed in the next sections, depending on their belonging to the front-end part or the back-end one.

1.4 Our Goals

In order to obtain a professional synthesizer we have set ourselves different goals to achieve.

For the Back-end part, we set 5 goals:

1. **Optimization of resources and readability**. Starting from the default 4 JUCE C and C++ plugin files (Plugin Editor and Plugin Processor), we split the project adding new code files to better control the oscillator and ADSR parameters and to manage the GUI in an easier way.
   This allow us to modify and check every part of the code in an easier way. The main 4 JUCE plugin files call back all the other components in the subfolders.

2. **Polyphony**. We wanted a polyphonic synthesizer that allows the user to play more than a note at the time (for example in a chord). We dealt with it in the *synthvoice* section.

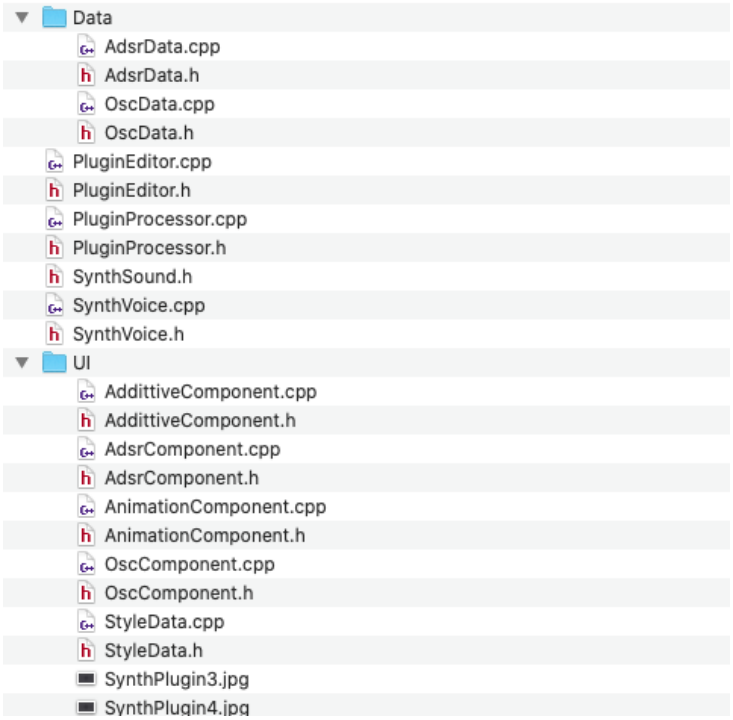

Figure 2: Organization of JUCE files in subfolders

3. **Waveform**. We wanted different waveforms (such as sine, saw, square) in our synth and we wanted to allow the user to set a specific waveform. We inserted a drop-down menu with different waveforms and we defined them in the *OscData* section.

4. **Parameters control**. All the oscillators and ADSR parameters need a knob control. The plugin has in total 11 parameters (7 for the oscillators and 4 for the ADSR section) and we managed these controls in the **PluginEditor** section.

5. **Real Time modifications**. Allow the user to modify the oscillators Frequency and Amplitude parameters in real time.

Moreover, for the ***Front-end part***, we set 3 goals:

1. **Layer graphical structure of the GUI**. In order to show knobs positions variations and to better design the plugin, we used a layers superposition graphical structure for the GUI. Each graphic layer includes some elements such as oscillator sections and knobs.

2. **Cool animation**. We also included a very cool animation around the central knob (that controls the amplitude of the first oscillator). The animation is a waveform that varies with the synth frequency. To obtain this effect, the animation is directly connected to the audio bus and it takes the value of the real output signal of the synthesizer.

3. **Graphical parameters control**. In order to allow the real time rotation of the parameter knobs, we created specific code files (located in the UI subfolder) that are linked to the knobs graphical layer.

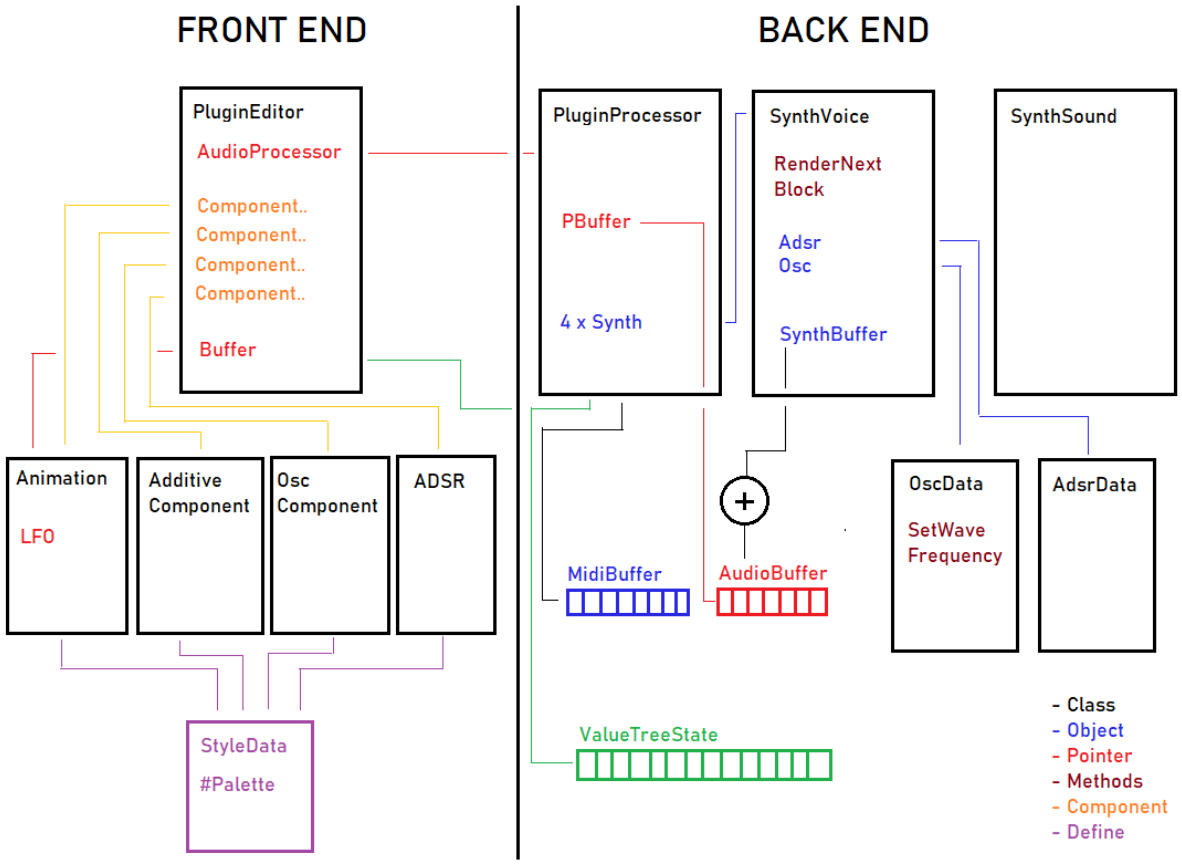

Figure 3: Structure of the synthesizer

2 Back-end

As previously stated, the back-end part of a project, or an application, consists of everything which happens behind the front-end part; in other words: everything hidden from the user interaction which allows the program to work.
As far as it concerns a Plug-in, the back-end part embodies the mathematical calculations that the software has to do on the incoming digital signal in order to manipulate it.
Since the way the sound is created has already been discussed in the second subsection of this paper, only the 5 goals for the back-end part of the project will be considered here.

Before diving into each one of the features (embedded in the goals), a few words on the files needed for implementing them are needed.
As mentioned earlier, for the back-end part, seven files have been added to the source code in addition to those that JUCE provides (i.e.: Processor.cpp and .h):

- **SynthVoice.cpp and .h**
  This couple of files only contain the class SynthVoice and the set of the instructions for a voice to behave when it is created, played, altered, or destroyed.

- **SynthSound.h**
  Just like the one previous ones, this file keeps a class which is in charge of retrieving the sound waveform from oscData and apply it to the note played. Due to its simplicity, only the header file was sufficient for the document to stand alone.

- **AdsrData.cpp and .h**
  These files contain the class AdsrData which, through the method updateADSR(), updates the value of the ADSR parameters every time they are modified by the user.

- **OscData.cpp and .h**
  These files contains the class OscData. This is a class of major importance since its duty is to set all of the parameters for the various oscillators. For instance, it sets the waveform, the frequency.

Let us now begin with the discussion about how these classes have been used in order to accomplish each of the objectives stated above.

2.1 Polyphony

Polyphony is a feature that allows instruments to play different keys at the same time, thus the possibility of playing chords.
It was the simplest feature to include in the project, as it only entailed the creation of a "for cycle" in the *PluginProcessor.cpp file*.
During iterations, which lasts a number of cycle equal to the number of voices one would like to have (in this case they were four), a new voice and a new sound are created for every synth, each of which has been previously created in the private section of the *Processor.h* file.

2.2 Waveform

The synth created during the course offers the possibility to select over three different wave' shape generated by the four oscillators.
In order to implement this feature, two new files have been instantiated: *OscData.cpp* and *OscData.h*.
In the *.cpp* a new class has been created under the name of "OscData", and has been made to inherit two JUCE's classes: *juce::dsp::Oscillator* and *juce::dsp::Gain*.
Then, the class has been filled with the methods needed to control the parameters of the oscillators such as the frequency, the gain, and the waveform itself.
Focusing the attention on the latter, the possibility of changing the waveform has been implemented via a user-defined function called *setWaveType*. It consists of a "switch" control flow which allow to choose between 3 cases which represent the three wave-forms: sine, saw and square.

Since we have introduced the *OscData* class, it could be helpful to stick to it to introduce the next feature in the next subsection.

2.3 Non linear frequency scale and real time modification

The synth discussed in this paper has four oscillators, one keyboard-controlled, and three more GUI-controlled.
It means that, while the frequency of the first one is controlled via keyboard (MIDI), the other ones have to play a frequency that is the one of the main oscillator plus a delta of frequency controllable via the GUI.
At first, the idea was very simple: the user selects the frequency delta of each secondary

oscillator through the GUI, and the program calculate the final frequency according to the following formula.

$$f_s = f + \delta \tag{1}$$

where f is the frequency of the main oscillator, fs is the one of the secondary oscillator, delta is the parameter changeable via GUI. This method, though, led to a particularly unpleasing sound due to the fact that the frequency scale is not linear. For this reason, it was decided to implement a different formula in the code, this time based on the equal temperament:

$$f_s = f * 2^{\delta} \tag{2}$$

This time, the emitted sound was way more pleasant since the frequency delta respect the non-linear nature of frequency scale.

This feature, together with the possibility to change the delta of the oscillators in real time while playing, has been implemented in the *OscData* class by means of the *setWaveFrequencyRuntime* method, a user-defined method.

2.4 Parameters control

To conclude this section, let us now talk about the *back-end* part of the GUI controls: the oscillators parameters and the ADSR parameters.

2.4.1 Oscillators parameters

The implementation of the controls for the oscillators has been accomplished in the *OscData* class.
In here, there two methods that mainly take care of it: *setWaveFrequency()* and *setGainLinear()*.
The former's duty is to update the frequency played by the oscillator accordingly to the position of the rotary slider on the GUI and by means of the formula discussed above.
The latter, intstead, takes care of updating the volume of the oscillator according to the position of the slider on the GUI.

2.4.2 ADSR parameters

ADSR indicates the evolution of the sound of a single note through time.
To implement the possibility of modifying it has been pretty easy since JUCE offers an API suited for the ADSR; it is called *juce::ADSR* and it has been made to inherit the user-defined class *AdsrData*.
Inside of it, the method *updateADSR()* has been implemented.
It takes as inputs the values to update (attack, decay, sustain and release) and takes care of updating them according to the value chosen by the user on the GUI.

3 Front-end

3.1 Design of the *Graphical User Interface*

Let us now talk about the *front-end* side of our application, that is the part related to everything that interacts directly with the user.

The *Graphical User Interface* (GUI) provides point-and-click control of the software application, so that there is no need to learn a language or type any command in order to run the application.
The design of the GUI critically affects the usability, usefulness, learnability, and enjoyability of the system, together with shaping the ultimate success and acceptance of the entire project.
After the development of all the features of our synthesizer, we focused our attention on designing an interface with controls for the additive synthesis which was both graphically elegant and easy to interpret in order to obtain a friendly and immediate visual impact.

The parameters needed to be interactive were only the gain for the first oscillator (since the frequency is controlled by a MIDI input) and both the gain and the frequency of the other three oscillators.
We organized the controls of our synthesizer in different modules and displayed them in a rectangular interface.
The GUI was built as a layer structure and the different components were manually designed, trying to find a layout that was pleasant but also respectful of the synthesizer's architecture logic.

3.2 GUI organisation

Let us now give a description of what makes up the user interface.
As stated, it is organised in five overlaid layers each of one showing different section of the GUI. The first one is the plugin editor which is not visible because it is overwritten by the second layer showing a background image.
Then, the other layers that compose the GUI are, in order:

- **Animation component**
  Controls the background and the animation of the different components.

- **Additive component**
  Contains a central rotary slider which controls the gain of the first oscillator and the couples of rotary sliders of gain and frequency which control the other oscillators. They are arranged in a mirror image at the corners of the rectangular interface.

- **Oscillator box component**
  Drop-down menu that allows the user to select the type of waveform (sine wave, square wave, etc.) the synth has to play.

- **ADSR component**
  Contains the four rotary sliders that control the **Attack** (the time taken for initial run-up of level from nil to peak, beginning when the key is pressed), the **Decay** (the time taken for the subsequent run down from the attack level to the designated sustain level), the **Sustain** (the level during the main sequence of the sound's

duration, until the key is released) and the **Release** (the time taken for the level to decay from the sustain level to zero after the key is released).

3.3 The valueTreeState array

In order to allow the communication between these components and the *back-end* part, a object directly called from the JUCE's APIs has been used; that is the *valueTreeState*.
It is an array, instantiated inside the *Processor*, whose task is to store every data belonging to the GUI components.
Each time a new element is to be added to the GUI, it is instantiated as a valueTreeState object, and pushed inside this array.
Then a class for each type of object (the animation, the knobs, etc.) has been created together with a pointer linking to the corresponding section of the *valueTreeState* array.

4 Conclusions

During the development of this Plug-in we have been seeing how to create an application using the JUCE *framework*, and how it manages in bringing the development of sound manipulation applications almost on the same level for both programmers and artists, fostering more robust collaboration and faster creative development. This will allow access to the world of Plug-In development to a wider audience of users and its consequences will have observable in various fields, from the artistic one to the engineering one.
In addition, the spread of this type of programming will also positively affect the market for this software, where an increase in the availability of these products will lead to greater competition, and a consequent reduction in prices.
Let us conclude by showing the final result of the Plug-In look.

Figure 4: Final concept of the plugin