CMLS - HW2 Report

10494722
10800940
10502570
10719249

May 12, 2021

Contents

Chapter 1

HW2 - Oscillator plugin

1.1 Homework

The aim of Assingnment 1 is to create JUCE plugin composed by 4 oscillators. The first of all has a MIDI input that rules the frequency of the oscillator while the gain is modified using a rotary slider. Other three oscillators have 2 rotary sliders that allows to change values of frequency and gain. The output must be the sum of the four oscillators. The GitHub repository of the code is available at this link:

`https://github.com/clarmasm/Additive_synthesis-CMLS_HW2.git`

The repository is already public. Click on link above to reach.

1.2 Idea

We decided to modify the first version of oscillator seen in classroom in order to accept a midi input coming from a digital MIDI keyboard. We had several ideas of implementation:

- Multiple Plugin composition: the first idea was to create 2 plugin
  - one able to manage the input of the keyboard with a rotary knob for the gain
  - one instanced three times with also a rotary slider for the frequency change

  The idea was to instance one of MIDI and three of the other type and sum all outputs "graphically" in the AudioPluginHost.
- Single Plugin multiple function: code all the four blocks in a single plugin.

We decided to follow the second idea because it seemed to be closer to the specification given.

1.3 Implementation

1.3.1 Plugin Processor

In order to achieve the homework's goal we needed to pass the frequency parameter from the MIDI input, which can be both physical and virtual, to the four oscillators. The first oscillator takes the frequency parameter as it is, while the other three change it by a given offset controlled by the user thanks to the specific rotary knobs, which are located on the GUI preferences screen of our plugin in the AudioPluginHost window. In the `PluginProcessor.h` file we declared all the variables necessary for computing the additive synthesis. These include:

- A global variable for the frequency controlled by the MIDI input (`car_freq`)
- Variables related to the frequency offset of the three 'secondary' oscillators (`freq1, freq2, freq3`)
- `gain` variables related to the slider of each of the four oscillators and `gainNow`) variables for the computation of the output signal
- Phase variables for each oscillator

We also declare the functions that were included in the editor, which set the value given by the slider into the variables `gainMIDI, gain1, gain2, gain3, freq1, freq2, freq3`. By connecting the MIDI input to our plugin, it will receive the MIDI information thanks to the `getNoteNumber()` and `getMidiNoteInHertz` functions applied to the MIDI message `m`: the former digitally returns the number of the key pressed on the MIDI keyboard, while the latter - considering this number - returns its frequency value. It is stored into the `car_freq` variable. This last process must be done for each message in the MIDI buffer. Inside the `processBlock` section of `PluginProcessor.cpp` we start by implementing a `for` loop for the MIDI data reading. If the MIDI message gives a `NoteOn`, then we update the `gainNow` variables of each oscillator to the value given by their sliders and we store the frequency value. If we get a `NoteOff`, all their values are set to zero.

Additive Synthesis

The total output of our plug-in is the sum of the sinusoids given by the four oscillators; we implemented a second `for` loop for each sample of the audio buffer, in which we sum the value of four sine functions. Inside the `prepareToPlay` section we initialize all the private variables to zero, and then we declare `sinMIDI, sin1, sin2, sin3`. The output sine wave of the MIDI oscillator is made in this way: `sinMIDI = gainMIDINow * (float) sin ((double) phaseMIDI);` where the `gainMIDINow` is the gain chosen by the user and the `phaseMIDI` is the phase, which is updated for each iteration thanks to these code lines: `phaseMIDI += (float) ( M_PI * 2. * ( ((double) (car_freq) / (double) SAMPLE_RATE)));`
`if( phaseMIDI > M_PI * 2.  )  phaseMIDI -= M_PI * 2.;` We construct the sine waves of the other oscillators in a similar way, except for the fact that the frequency is affected by the value chosen with the slider. Since we chose to use multiplying factors for the frequency and the possible values for the slider go from -1 to 1, we chose to modulate the frequency by multiplying it with and exponential: car_freq*(pow (2.0, freq1)). The result is a modulation in frequency that goes from on octave below to one octave higher. The resulting synthesis block is the following:

```
for (int i = 0; i < numSamples; ++i){
        sinMIDI = gainMIDINow * (float) sin
                ((double) phaseMIDI);
    sin1 = gainNow1 * (float) sin ((double) phase1);
    sin2 = gainNow2 * (float) sin ((double) phase2);
    sin3 = gainNow3 * (float) sin ((double) phase3);
    channelDataL[i] = sinMIDI + sin1 + sin2 + sin3;
    channelDataR[i] = channelDataL[i];
    phaseMIDI += (float)( M\_PI * 2. *(((double)
        (car\_freq) / (double) SAMPLE\_RATE)));
    if( phaseMIDI > M\_PI * 2. ) phaseMIDI \-= M\_PI * 2.;
    phase1 += (float) (M\_PI * 2. *(((double)
        (car\_freq*(pow (2.0, freq1))) /
                (double) SAMPLE\_RATE)));
    if( phase1 > M\_PI * 2. ){
    phase1 -= M\_PI * 2.;
    phase2 += (float) ( M\_PI * 2. *(((double)
        (car\_freq*(pow (2.0, freq2))) /
                (double) SAMPLE\_RATE)));
    }
    if( phase2 > M\_PI * 2. ){
    phase2 -= M\_PI * 2.;
    phase3 += (float) ( M\_PI * 2. *
        ( ((double)(car\_freq*(pow(2.0, freq3)))
        /(double)SAMPLE\_RATE)));
    }
    if( phase3 > M\_PI * 2.) phase3 -= M\_PI * 2.;
}
```

1.3.2 Plugin Editor

In `PluginEditorḣ` , after including the default header files for a JUCE plugin Editor, we defined the `Add_synthAudioProcessorEditor` class.

Besides the usual default properties and methods of JUCE classes we added the inheritance of `Slider::Listenener class to the Editor`, i.e. we added it as base class, so that we can register our class to receive slider changes. Then we had to define all the necessary sliders to control our oscillators from the GUI and we did the declaration, creating Slider Objects as private members of our class. In particular, we needed seven sliders.

- Four of them are used for controlling the gain of each oscillator.
- Three refer to the frequency offset with respect to the frequency of the MIDI-controlled oscillator, so we needed one slider for each one of the other three oscillators.

We had also to define a proper label for each Slider.

For instance:

- `juce::Slider gainMIDISlider; juce::Label gainMIDILabel`
- `juce::Slider gain1Slider; juce::Label gain1Label`
- etc.

In the first of the previous cases we clearly refer to the directly MIDI-controlled oscillator while the other ones are idientified by 1, 2, and 3 (`gain1Slider, gain2Slider` etc.). Besides the declaration of sliders, we finally declarared the default callback function `sliderValueChanged`. The callback function has to notify the listener about the changes and pass the value to update the previously stored one.

In `PluginEditor.cpp` we started setting the size of the Editor properly with the command `setSize`. Then, we did the following operations for each of the four oscillators. We linked the sliders and their label using `setText`. After that, we set some properties of the gain slider with various functions: `setRange(0.0, 1.0, 0.1)`, `setSliderStyle(juce::Slider::Rotary)` and `setTextBoxStyle`.

The gain range is thus between 0, with which we don't have sound at the output, and 1, corresponding to not limiting at all the sound (this is the maximum possible, so we always reduce and never increase loudness with respect to the default one), while the gain step is 0.1, so we can get the 10%, 20%, 30% etc. of the gain.

The slider stile is rotary like in common oscillators. We then add the slider listener to our slider with `addListener(this)`. Finally, with the command `addAndMakeVisible` we let both the slider and the corresponding label appear in the editor. As said before, we repeated these operations for all the oscillators.

We used the same set of functions for the frequency offsets. What changes from the gain case is the range: This time we have -1 and 1 as bounds. The step will be 0.01, that limits the possible offset choices. Moreover, each offset is set to an initial value of zero. These values refer to a factor in a logarithmic scale (a negative value implies multiplying the frequency for a number between 0 and 1).

After all this, we worked on the graphic aspect. Being our component opaque, we had to completely fill the background with a solid colour:

`g.setColour(juce::Colours::white)`.

We also set the font of the labels (`g.setFont(15.0f);`) After that, with the function `setBounds(x, y, width, height)`, possible to use thanks to the resized functions in the header file, we layed out the positions of the sliders in the editor (this function changes the component's position and size).

The values of the coordinates are relative to the top-left of the component's parent. We had to be careful all the seven sliders were perfectly recognizable and separated and chose the coordinates consequently. Finally, implementing the `sliderValueChanged` function, we were able to make it possible for JUCE to understand our manual use of the sliders by the GUI and set properly the wanted gain and frequency offset. If the compharison matches the values are updated.

1.4 Conclusions

1.4.1 Features

We decided to left free values of gain in our plugin. Setting all gain sliders in mid-high position we can have the sound clipping. We decided to let it possible to allow users to get every kind of sound he want, included clipping ones.

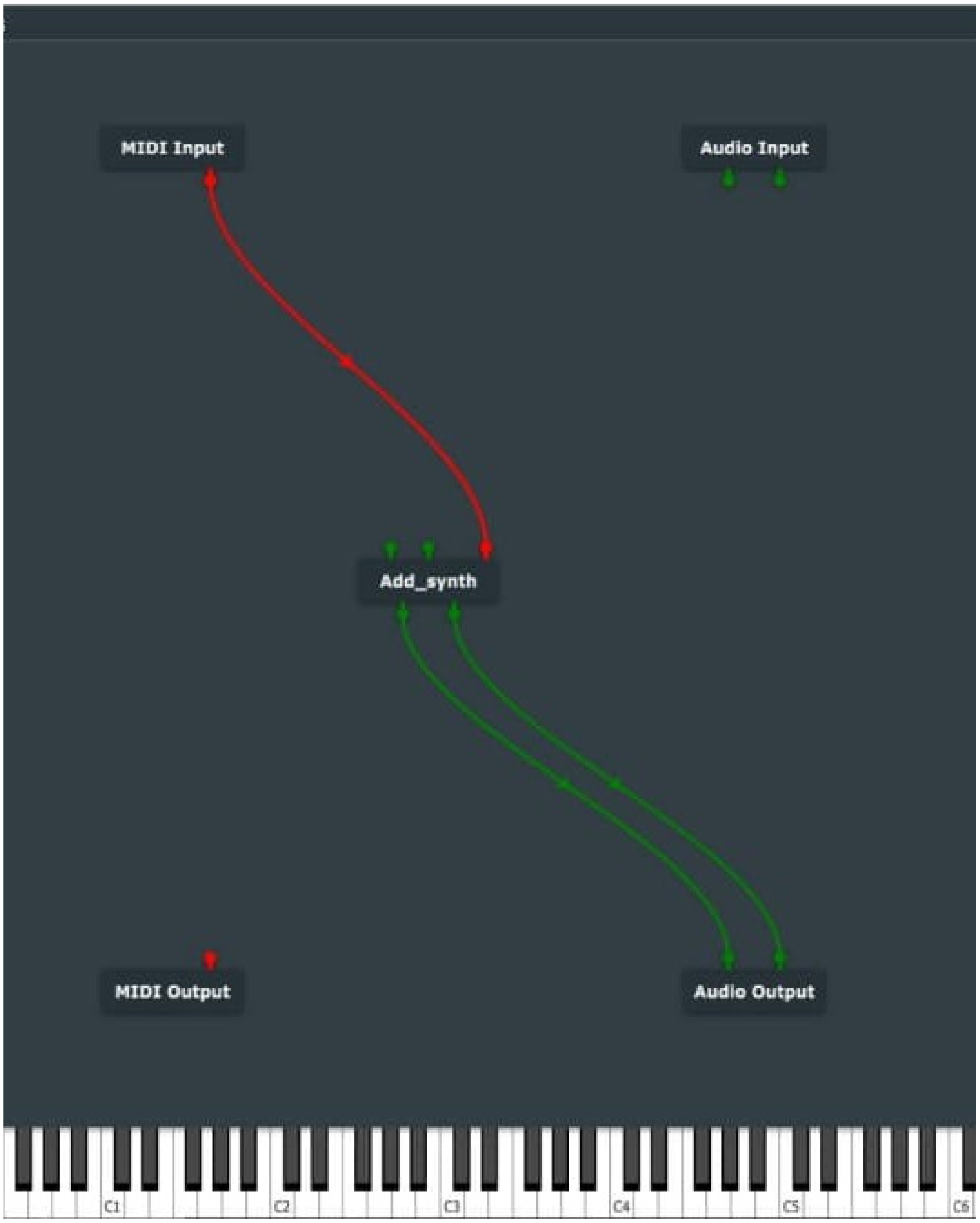
In a first moment we tryied to use an external MIDI keyboard, Virtual MIDI Piano Keyboard, with a proper virtual cable to connect it. We decided to abandon it and use the pre defined keyboard of AudioPluginHost.

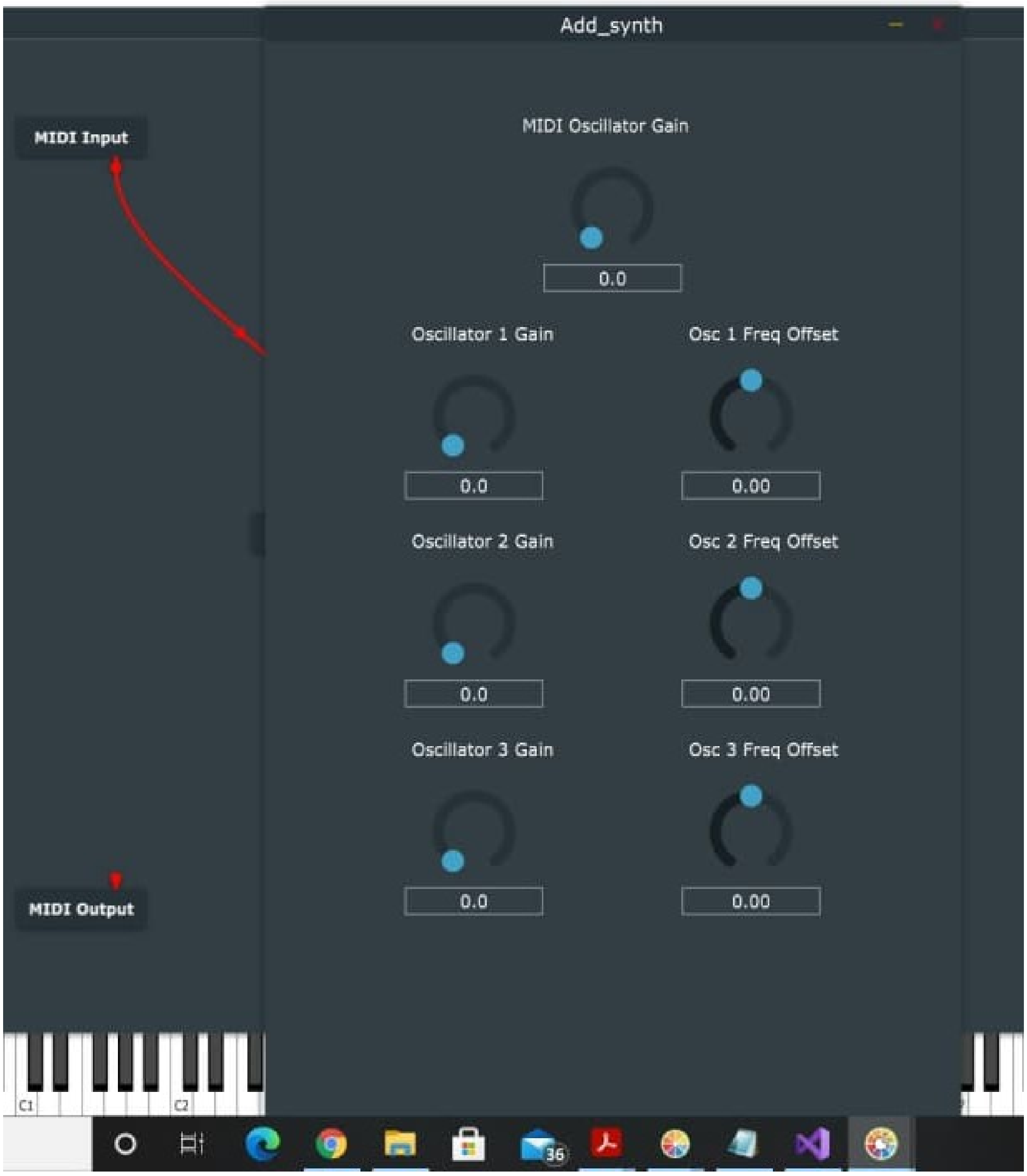
1.4.2 Issues

We had several problems with the compilation of the plugin. Every time we have modified the code, we had nonsense error of not declarated variables, even though they were correctly initialized. That made our code not compiling at all or compiling with sounds very short and out of tune. We have finally been able to solve this issue deleting the VisualStudio Folder of the project that contains all builts files.

1.4.3 Graphic Screenshots

Here we have some screenshot about the final result.


MIDI Input
Audio Input
Add_synth
MIDI Output
Audio Output
C1
C2
C3
C4
C5
C6


Add_synth
MIDI Input
MIDI Output
MIDI Oscillator Gain
0.0
Oscillator 1 Gain
0.0
Osc 1 Freq Offset
0.00
Oscillator 2 Gain
0.0
Osc 2 Freq Offset
0.00
Oscillator 3 Gain
0.0
Osc 3 Freq Offset
0.00
C1
C2