Assignment 4: Voice Harmonizer

10497475|10788381|10833877

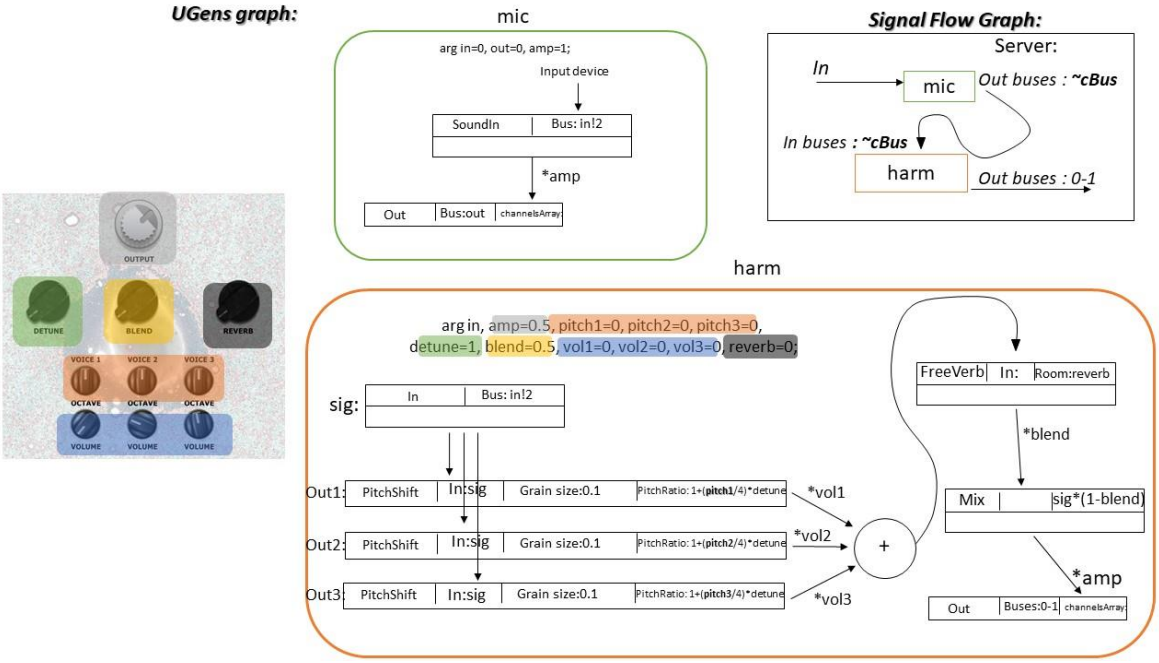
GITHUB

Introduction

Vocal harmonizers are largely spread among singers since they are able to add depth and emotional charge to voice. Most used and reliable vocal harmonizers on the market simply *add harmony* to the original sound. This effect is often obtained in the following way: the input sound pitch is shifted above or below an amount corresponding to a third (5/4) or a fifth (3/4) of the scale (according to Pythagorean tuning).
Our harmonizer is equally based on this behavior. The user can control three different voice lines: for each one, he can decide which note must be added (third or fifth above or below) and the volume. In addition to that, it is also possible to control the blend parameter, a detune factor and a reverb effect. Finally, the user can control the overall output volume.

CODE SECTION: SUPERCOLLIDER

Our Supercollider script shows a simple yet efficient implementation of the harmonizer. It is mainly based upon two different synths: the one controlling the input from the microphone and the one that processes it creating the actual harmonization.
In the first one we simply read the signal coming through the microphone exploiting the UGen *SoundIn.ar(in!2).* The argument between parenthesis specifies Supercollider to use both the channels available so that it doesn't result in a mono signal.
For the second synth, *\harm*, we first retrieve the input and then we proceed to compute the output of the three harmonizing voices and save it on three different variables: *out1*, *out2*, *out3*. We achieved that via the use of *PitchShift.ar* which shifts the frequency of the input signal by a fixed multiplier that we mapped and control via the value of the voice knobs. It is important to notice also that since we decided to add the possibility to slightly detune the signal, this value is also multiplied by the detune factor.
Finally, the three *out1, out2, out3* variables are summed together to obtain a single final *out* which is reverbed according to the user's choice and mixed together with the clean signal before sending it to the output. Of course the percentage of the harmonized signal over the clean one can be weighted through the blend knob.
The last part of the code is devoted simply to send the final result in the two pre-allocated busses and to establish the communication between Supercollider and the other platforms controlling the messages from the UI.
The graph below shows the overall structure of the Supercollider code enlightening the relationship between the different signals and how they are retrieved and processed.



CODE SECTION: JUCE

Juce part is totally dedicated to the user interface implementation and OSC communication. For the communication towards SuperCollider, we exploited the functionalities of the juce module *juce_osc*. To obtain more flexibility, we also used an *AudioProcessorValueTreeState* that is able to create and link sliders to the Processor class in a rapid manner.

PluginEditor.h /PluginEditor.cpp

We declared the knobs, their attachment and their labels as scoped pointers in the Editor.h. Being Juce expected to sense any knob modification performed by the user, we made Editor.h inherit from *AudioProcessorValueTreeState::Listener*. Thanks to this class, we have overridden the function *parameterChange* that changes the value of the parameter specified by an univoque ID: this function will be automatically called anytime a knob is modified.
In the Editor.cpp constructor's we specified all knob's graphical characteristics, we linked each slider to the relative slider attachment and we added a Listener for each parameter. In the resize() function we specified the characteristics of the main window and the knob's position within it. Furthermore we implemented the *parameterChanged* function, that sends an OSC message to SuperCollider anytime a knob is changed through the function sendOscInput, implemented in the processor.

PluginProcessor.h/PluginProcessor.cpp

In PluginProcessor.h we declared the following features: an AudioProcessorValueTreeState scoped pointer pointing to the state of the processor, sendOscInput function and some other functions necessary to retrieve useful parameters. In addition to that, we declared an OSCsender, required to allow OSC communication.
In PluginProcessor.cpp constructor we created the different parameters the user can control and added them to the ValueTree representing the processor. Moreover, we linked them to the slider declared in the Editor class exploiting the unique parameter ID previously stated. Here we set the ranges of the different parameters of the respective sliders.

OSC Communication:

In the constructor we also set up the connection between Juce and Supercollider, in the line shown below:

```
//2) SETTING UP CONNECTION
sender.connect("127.0.0.1", 57120);
```

where 127.0.0.1 represents the network on which the communication takes place (local host) and 57120 is the fixed port on which SuperCollider receives the messages.
Finally, in the sendOscInput function the actual OSC messages submission takes place though the *send* function of juce_osc module. Here we must specify the OSC method that will receive the parameter and the actual value to send.

GUI

The graphical user interface that we implemented through the JUCE framework is composed of two main sections.

VOICE SECTION

The section responsible for the control of the three voices is structured by 6 knobs.
For each voice the user can regulate the ***pitch*** at which it will be shifted and its output ***volume***.
The pitch Knobs can assume integer values between –2 and +2 that correspond to the inferior and superior harmonics added to the input signal.

OUTPUT SECTION

In the upper part of the GUI the user can find other four knobs:

- The ***Output Level Knob***, that controls the volume of the output signal;
- The ***Blend Knob***, that regulates the percentage of input signal and pitched voices that compose the total output;
- The ***Reverb Knob***, that sets the size of the room of the reverb applied to the voices;

- The ***Detune Knob***, that allows the user to detune the voices' frequency.

All the knob values are stored in a state of a AudioProcessorValueTreeState structure in order to facilitate the parameters retrieving and the communication with Supercollider.
Furthermore, we used a simple Look&Feel class in order to customize the knobs.

TouchOSC

In order to realize an interface utilizable also for different types of devices we exploited TouchOSCEditor for the design of a layout executable with TouchOSC App.

The TouchOSC GUI follows the main structure of the JUCE one, previously described. Indeed, it contains:

- 6 Knobs for the voice section (three for the ***pitches*** regulation and three for the ***volume***);
- The ***Output Level Knob***;
- The ***Blend Knob***;
- The ***Detune Knob***;
- The ***Reverb Knob***.

Communication from TouchOsc towards SuperCollider

The interaction between the graphical user interface realized in TouchOsc and the script written in Supercollider is very easy to configure: the user has to push the Sync button on the Touch Osc Editor and connect the device to the same WIFI network of the personal computer that is running the Supercollider script.
After that, from the TouchOSC App is possible to load the corresponding layout that we realized on the TouchOsc Editor and select the personal computer to match.
Choosing from the App the standard receiving port of Supercollider (57120) the interface will start to send the knobs and sliders values to the script.

Thanks to this kind of implementation the user can choose to control the parameters of the Harmonizer with the JUCE or the TouchOSC graphical user interface. In order to switch from one to the other is simply necessary to set up in the Supercollider Script the sending port of the desired interface: in the case of TouchOSC the number of this port it can be found and changed in the settings menu, while as regard the JUCE interface it changes at each execution (in this case we exploited both WireShark and OSCTrace for the selection of the right port).

Results

The resulting output is well harmonized. The three voice lines together with the reverb effect produce a deep and enveloping sound. Despite this, the sound produced by the artificial harmonics is clearly far from a human voice and is perceived as “strange” to our hearing. To improve this aspect a filtering operation of low and high frequency could be put in place and we could deepen the research about an ASRD model for human voice.