Synth 101

Prompt 1: Multi effect: Flanger - Wah Wah - Phaser

The Synth 101 group has been asked to develop a Multi effect software capable of managing multiple effects at the same time in different chain orders in supercollider. The requested effects were the Flanger, the Wah Wah and the Phaser, the code has been tested both through the input of a guitar and of a microphone passing through a sound card.
The following report is subdivided in different chapters:

- Realization of the individual effects
- Creation of Static GUI elements
- I/O management of the chain of effects
- Implementation of dynamic elements (GUI and audio bus switch)

This report will follow such sections closely and shed light on the code behind our multi effect.

1 - Realization of the individual effects

This section will cover the realization of the individual effects. Note that all of the pedals implement the *bypass* argument, which if equal to 1 makes the output signal be identical to the input signal, properly bypassing the effect.

Flanger

Let us introduce the flanger,whose Ugen graph can be seen below. In order to achieve the characteristic sound of this effect, we had to implement a feedback loop combined with the delay effect coming from the class *DelayC.*
In detail, in our implementation, *DelayC* takes the following parameters:

- The signal *in*, which is the input signal coming from the bus *input_bus*
- The maximum delay time of the uGen, which is 0.01 seconds
- The effective delay time that the Class needs to apply to the input signal, which is *myLFO+0.005* seconds*,* therefore it is not constant

Where *myLFO* is a sinusoidal oscillator with frequency *freqLFO* and amplitude *ampLFO*, whose values are to be seen in chapter 2.

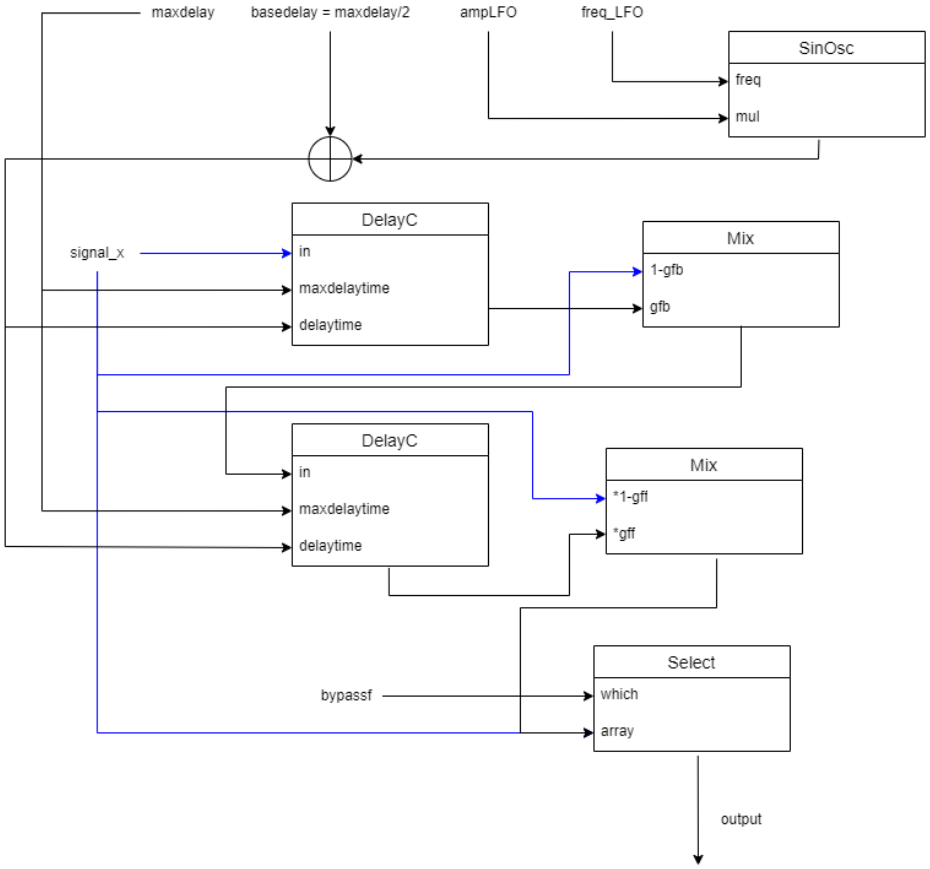

Image 1) The Ugen graph of the flanger

Subsequently, we implement the feedback loop by summing the delayed signal *signal_x_delayed* with the dry input signal. In this operation the parameter *gfb* has the active role of controlling the amount of feedback present. Once we have our *signal_xCycle2* we can send it through the DelayC Ugen a second time and finally proceed to mix the output signal with the input dry signal. In this last passage the parameter *gff* comes into play acting as a dry/wet controller. Lastly, we send the final signal to the output bus.

Phaser

The dreamy sound of the phaser is obtained by sending a signal several times through an allpass filter. We accomplished this by feeding our input signal *num* times to the *AllPassC* class thanks to the use of the control structure while.

The delay of this all-pass filter is modulated by a SinOsc that has frequency *mod_freq*, while the amplitude is fixed in value. It is important for both these parameters to be very very low in value since this is meant to be a modulating signal. The signal that comes out of the last

allpass filter is then mixed with the original input signal. The dry/wet parameter regulating the sum of the two signals is called *depth*. Finally, the signal is then sent to the output bus.

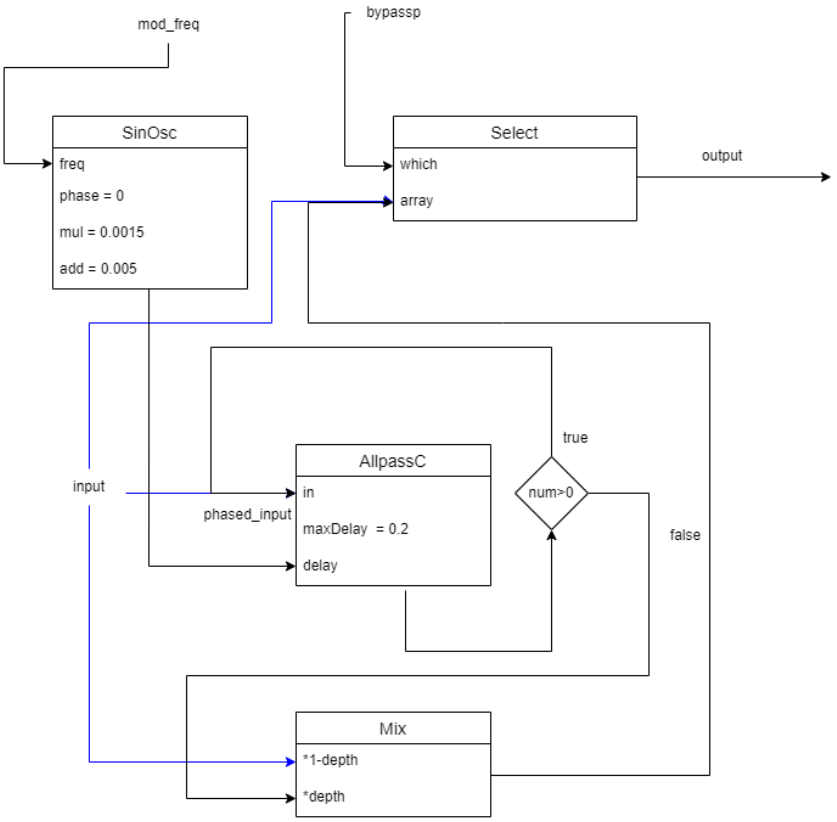

image 2) the Ugen graph of the phaser

Wah wah

This particular effect is obtained thanks to the action of a band pass filter and a low frequency oscillator. The first of the two is called *bpf*, this takes as input the signal coming from the input_bus and has as central frequency the parameter *fw* multiplied by the LFO. The LFO is realized by using an EnvGen that makes the central frequency shift from 350 Hz to 2200 Hz, this is the key point of this filter. The EnvGen is basically a triangular envelope whose gate is controlled by a SinOsc that has a frequency equal to *1/dur* that rules the speed of the frequency shift.

The output of the band pass filter is mixed with the dry input signal and finally sent to the output channel. When mixing the two signals the parameter *amount* comes into play, acting as a dry/wet regulator.

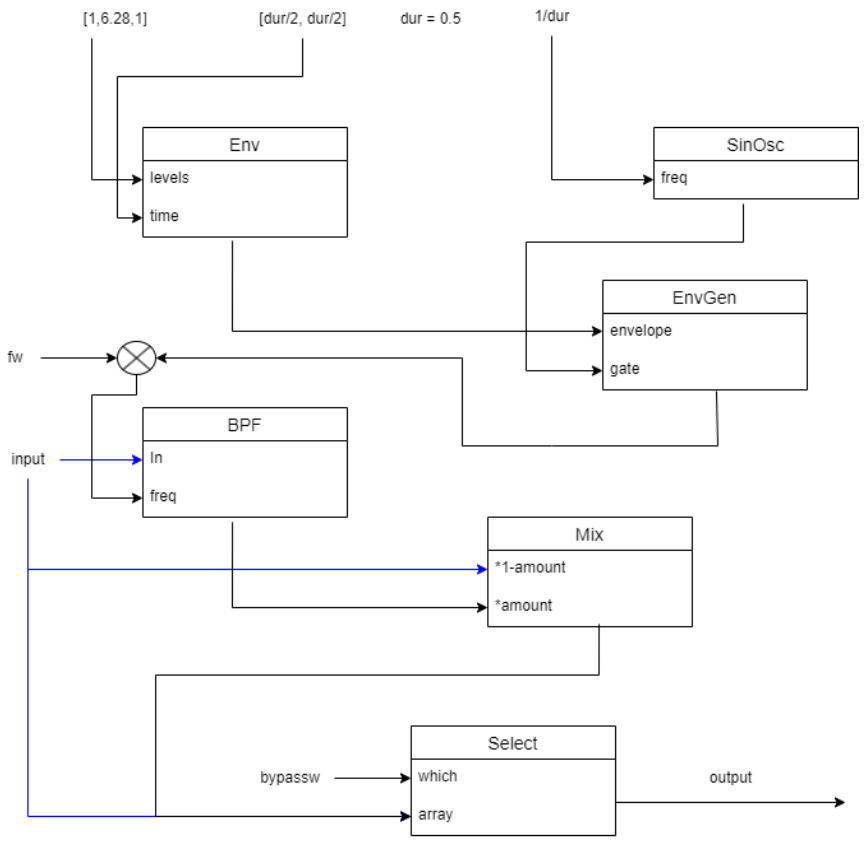

image 3) the Synth scheme of wah wah

OutputAudioStereo

Lastly we implemented a final SynthDef called *outputAudioStereo* which has the job of taking the output signal at the end of the chain of pedals and make it a stereo signal. *outputAudioStereo* is also responsible for the pan effect, the mute button and the master volume.

2 - Creation of static GUI elements

The Graphic User Interface is based on an instance of the class *Window* which constitutes the main container for everything that appears on screen. Moreover, the window *w* contains 5 child views, three of them realize the pedals interface, while the other two are instead dedicated to the signal representation graphs. In *w* we also placed several graphical embellishments, like the double-sided arrows, thanks to the use of the class Pen.

Realization of the pedals

The views correlated to the pedals, realized with the command *CompositeView*, include different elements inside:

- The text labels of the Synth
- The knobs that control the the non-static parameters of the Synth
- The button that turns ON or OFF the related Synth

In particular, the specific knob settings for each pedal are the ones here below:
For *view1* (PHASER effect):

- Label “frequency”, the knob takes as value *mod_freq*, as range *specFreqp,* and as initial value **‘0.6’.**
- Label “dry/wet”, the knob takes as value *depth*, as range *specDepthp* and as initial value **‘0.5’**

For *view2* (FLANGER effect):

- Label “frequency”, the knob takes as value *freqLFO*, as range *specFreqf* and as initial value **‘1.5’**
- Label “feedback”, the knob takes as value *gfb*, as range *specGfb* and as initial value **‘0.5’**
- Label “width”, the knob takes as value *ampLFO*, as range *specAmpf* and as initial value **‘0.0025’**
- Label “dry/wet”, the knob takes as value *gff*, as range *specGff* and as initial value **‘0.5’**

For *view3* (WAH WAH effect):

- Label “frequency”, the knob takes as value *dur*, as range *specDurw* and as initial value **‘1.25’**
- Label “dry/wet”, the knob takes as value *amount*, as range *specAmountw* and as initial value **‘0.5’**

The aforementioned ranges have been defined using the *ControlSpec* class in a linear progression fashion around the two extreme values of each controlled parameter.

Signal representation

In our multi effect we also thought of adding a button that allows the user to switch the representation of the signal between time domain and frequency domain.
The input and output frequency analysis have been developed through the *FreqScopeView* class, which receives the window *w* and an instance of *Rect* as parameters to create the desired structure. In detail, the input spectrum is associated with the first input bus of the device on which the program is running, while the output spectrum is associated with the output bus *~boutForFrequencyOutGraph***.** Both the spectra have been set to 120 dB as far the amplitude range is concerned.
For what regards the input and output time domain view, we had to allocate two stereo buffers. Their contents are constantly updated through the use of a *ScopeOut2* Ugen which reads respectively from *~firstInputChannel* and the output channels [0, 1]. Then, the visualization of both signals is performed by exploiting the class *Scopeview*, whose function is to periodically scan the buffers and refresh the plots.

3 - I/O management of the chain of effects

In order to make the multi effect work in serie we have to make a smart use of the input and output of each single pedal, this is why we have decided to implement a linking system based on 4 private audio buses. In particular the buses are the following:

- ~firstInputChannel bus, whose number is calculated by using the method s.options.numOutputBusChannels;
- the *b1* bus, that links the output of the first pedal to the input of the second
- the *b2* bus, that links the output of the second pedal to the input of the third
- the *bout* bus, that links the output of the third pedal to the input of the *outputAudioStereo* Synth

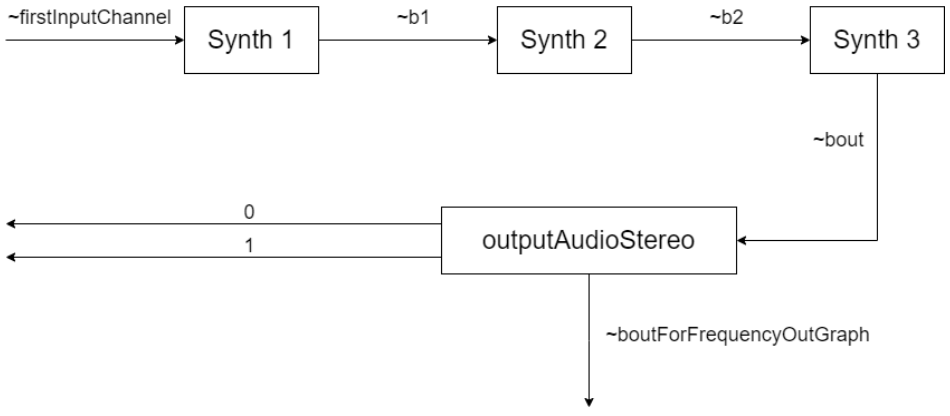

image 4) the bus management system

The *outputAudioStereo* Synth then simply transforms the final signal of the pedals chain into a stereo signal sent directly to our main audio output device. All these buses are passed as arguments each time we implement one of the Synths mentioned in chapter 1, so that we can control the audio flow properly. Note how the buses are defined as global variables, so that we can access them both during the compilation of the code regarding the effects and during the execution of the lines responsible for the rest of the program.
Since some of the pedals involve a delay line, we had to add some sort of tool to make the effects wait for the delayed output to be processed before starting any computation. This tool is the Group class. We implemented several groups whose names are *~phaserGroup*, *~flangerGroup*, *~wahGroup* and *~outputAudioStereoGroup*. Naturally each one of these is bound to the corresponding Synth.

4 - Implementation of dynamic elements

Now comes the real challenging part of the code, and that is realizing the switch between the effects. This switch concerns both the GUI and the Audio aspects of the code, we are going to address them both in the two sections below. For reference, this section regards everything from line 541 onward.

Graphical pedal swap

In order to make the pedals swap with one another we inserted two arrows in the gaps between the three pedals. The arrows outlines are drawn thanks to the class Pen, on top of these are created two separate transparent views that react when the user presses the mouse on them. Let's take in consideration only the left arrow w.l.o.g., in total there are 6 possible combinations in which the pedals can appear on the first two slots. To keep track of the order in which the pedals are currently disposed we made use of the global array *~pedalOrder,* which contains 3 elements corresponding to the three pedals. The array has been initialized like this:

~pedalOrder=[0,1,2];

We decided to represent the 3 pedals as numbers by making the convention of correlating the number zero to the phaser pedal, the number one to the flanger pedal and the number 2

to the wah wah pedal. This way, in the initial configuration [0,1,2] we have the phaser followed by the flanger and the wah wah.
When the user clicks the left arrow we first take into consideration which are the first two numbers in the array, this way we are actually evaluating all the possible combinations of pedals on the first two slots, these combinations are simply represented by the variable *case* in the code, which spans from 0 to 5. After the variable *case* has been defined we update the two leftmost elements of the array by switching them thanks to the method *.swap* called on the *~pedalOrder* Array, then in the following lines of code the GUI and Audio changes follow.
We can shift the position of the pedals for each specific case by using the method *.moveTo* on the views. We call this method on the two leftmost views, properly making the pedals interface swap. The same logic stands for the right arrow swap.

Audio switch of the effects order

In order for the audio to follow the GUI we decided to enhance the just mentioned 6 cases by adding some extra lines of code. What we need to accomplish in this stage is to reorganize the groups order and to change the order of execution of the effects. We achieved this through this kind of workflow:

1. Freeing the previous groups
2. Generating the new linking between the groups based on how they are meant to work after the switch.
3. Redefining the four new instances of the Synths discussed in Chapter 1, while being careful of maintaining the same values of each Synth argument before and after the switch and also rearranging the buses order to be coherent with how we want them to be after the swap.

So, basically every time we click one of the two arrows we are actually just recreating all the Synths anew in the correct order. Each of the six cases has its custom order of execution of the effects, therefore its own lines of code.