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 code has been subdivided in different sections:

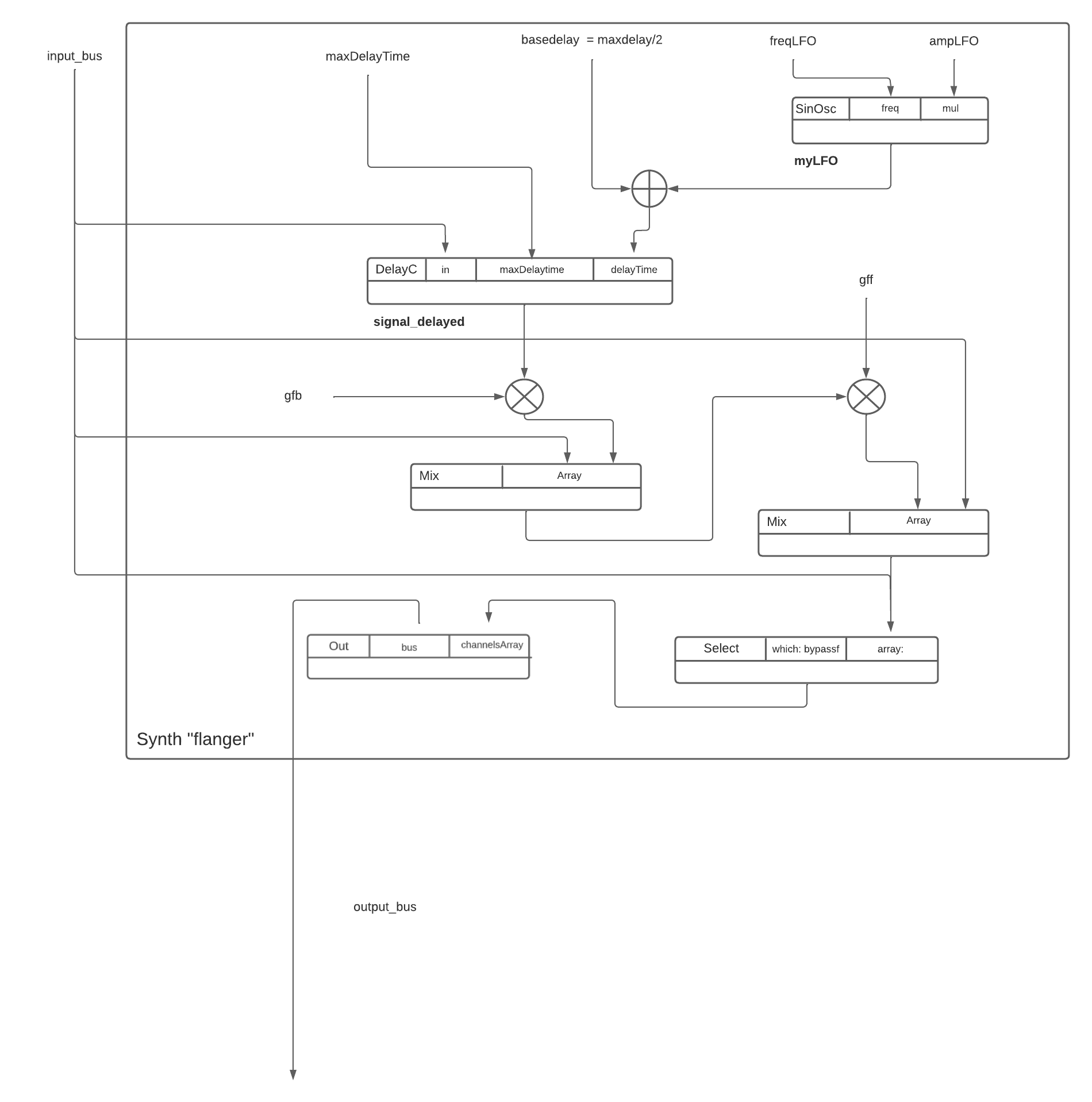
* Realization of the single effects
* Creation of Static GUI elements
* I/O management of the chain of effects
* Implementation of dynamic elements (GUI and audio bus switch)

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

# 1 - Realization of the single effects

## Flanger

Let us introduce the flanger. As we can see in the Synth graph below (image 1), the most important UGen of this effect is the *DelayC* one. Let’s quickly analyze our implementation of this class. Firstly *DelayC* takes as parameter *in*, which in our case is the input signal coming from the bus *input\_bus*, containing the desired input signal, secondly we decided that in this Ugen it was convenient to set a maximum delay time of 0.01 seconds, lastly we set the effective delay time that the Class needs to apply to the input signal, which in our case is the sum of a sinusoidal oscillator called *myLFO* and a constant value of 0.005 seconds stored in the variable *maxdelay*. *myLFO* has a frequency and an amplitude set respectively by the parameters *freqLFO* and *ampLFO,* with initial values 1.5s and 0.0025. Later, when we will define the range of values for the *ampLFO* that the knob on the flanger pedal needs to control, we will set the maximum amplitude as 0.005, therefore the maximum value of *delayTime* is exactly the value of *MaxDelayTime*, exactly as it should be.



*Image 1) The Synth scheme of the flanger*

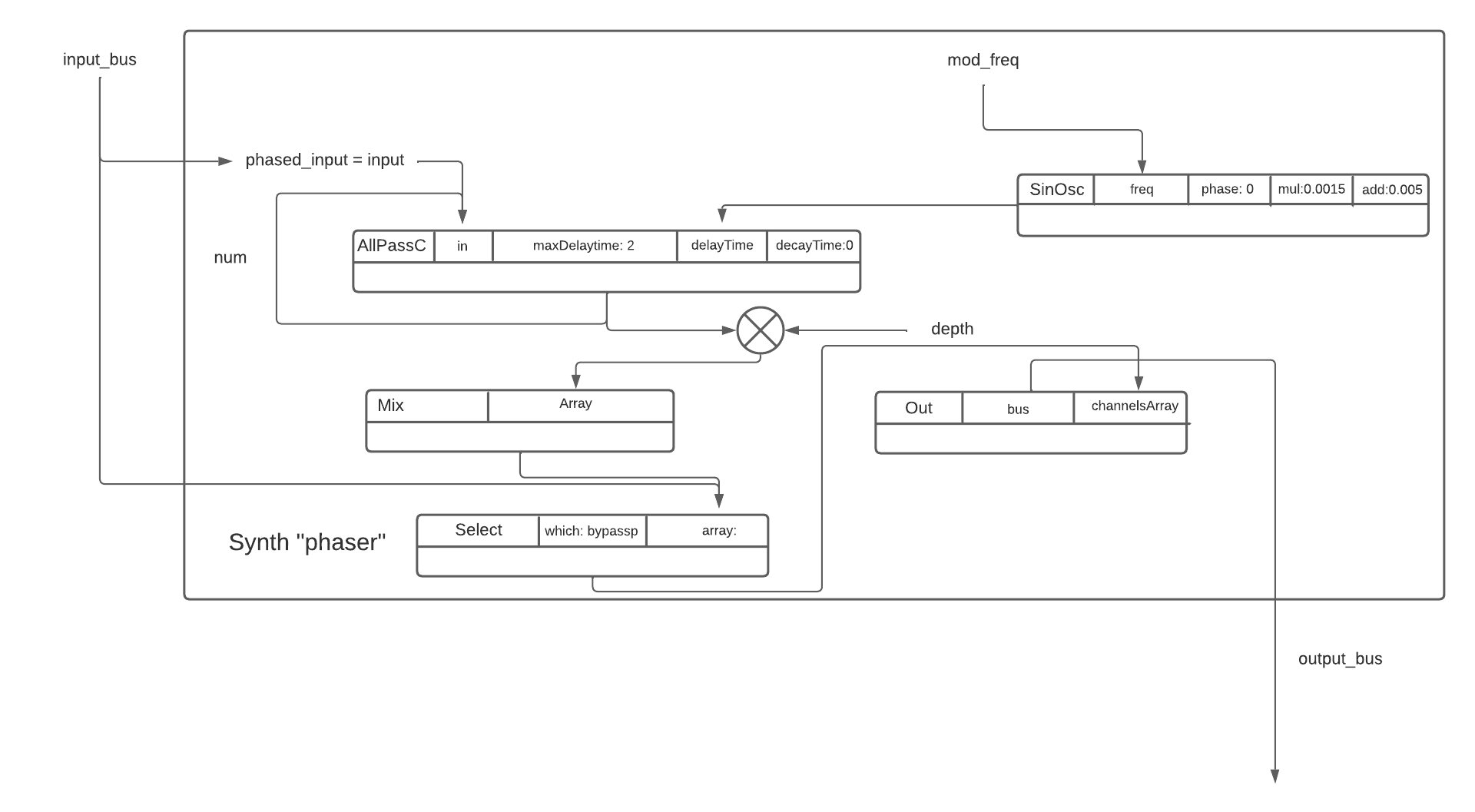
The output signal of *DelayC* is then multiplied by the parameter *gfb*, which stands for gain feedback, and mixed with the original input signal to give us the *signal\_x\_delayed* signal. *signal\_x\_delayed* passes again through the *DelayC* Ugen and is then almost ready to be sent to the output. Before we do so, though, we have to weigh this last signal by the value *gff*, which stands for gain feedforward and mix the final result with the original input signal *signal\_x*. Finally, we send everything to the output bus.

## Phaser

The core characteristic of this Ugen is the *AllPassC* Class, which helps us archive the classic sound of the phaser, it’s important to notice that this particular block is executed multiple times thanks to the use of the control structure while.

We chose to put the number of iterations of this cycle equal to 4 as specified by the parameter *num*. In other words, we are taking the input signal and applying the allpass filter 4 consecutive times. The delay of this all-pass filter is modulated by a SinOsc that has determined frequency (*mod\_freq*), amplitude and add parameters, which are all meant to be low in value. For reference, we made the *mod\_freq* parameter span between 0.2 and 1.

The signal that comes out from the fourth filter is then mixed with the original input signal after being weighted by the multiplication with the parameter *depth*. The final signal is then sent to the output bus.

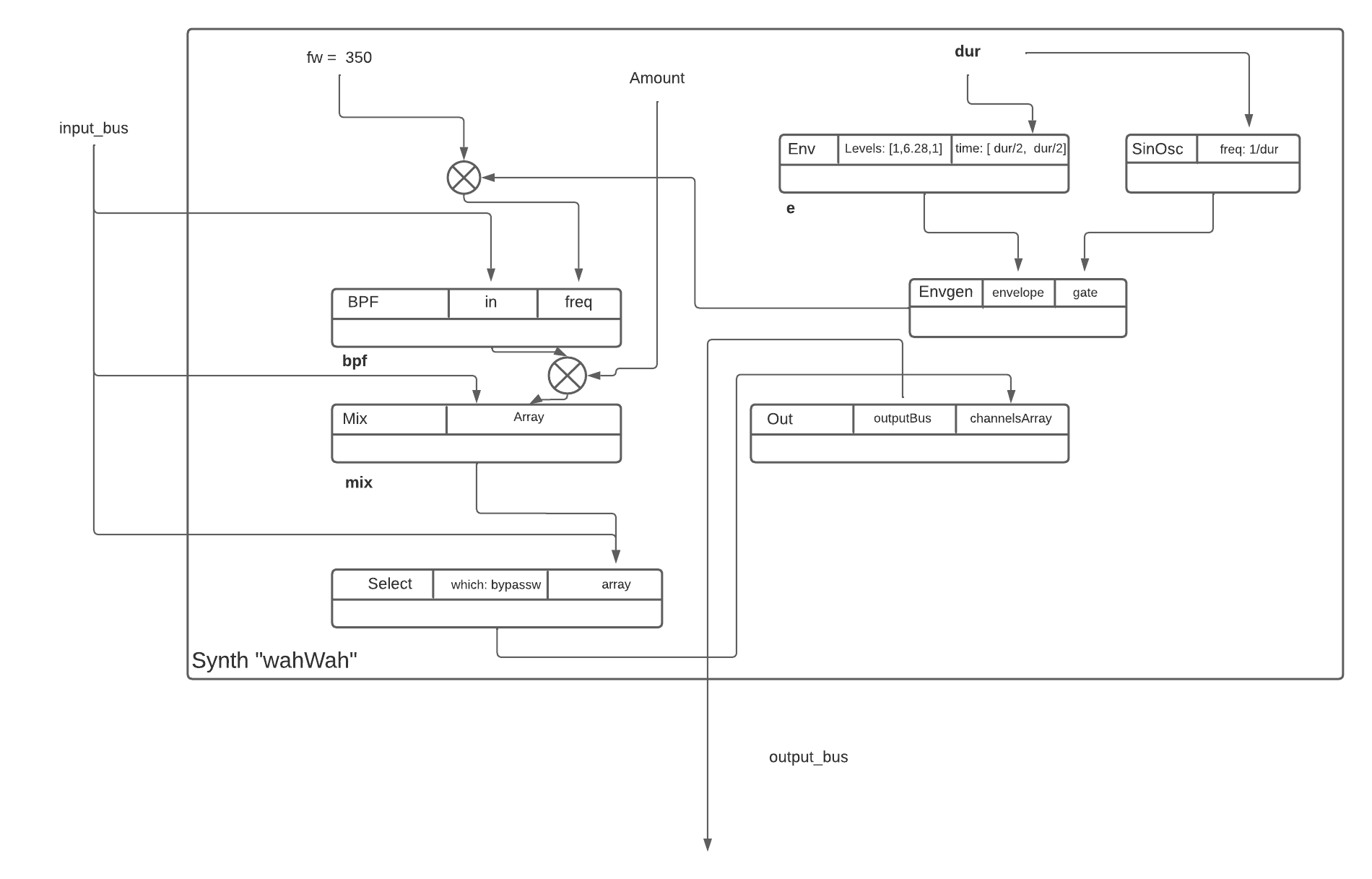


*image 2) the synth scheme of the phaser*

## Wah-wah

The most important thing to notice in this effect is the presence of a band-pass filter, called *bpf* in the scheme, that brings in the signal coming from the input\_bus and has as central frequency the parameter *fw* multiplied by an envelope. This envelope permits the window center shifting that is the key point of this filter. Our envelope 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 BPF is then multiplied with the parameter amount, mixed with the dry input signal and finally sent to the output channel.

**

*image 3) the synth scheme of wah-wah*

Lastly we implemented a final SynthDef called *outputAudioStereo* that merely sends to the speakers the output signal at the end of the chain of effects after making it a stereo signal.

It is relevant to say that in each of the previous effects we included the *bypass* argument, which is equal to 1 makes the effect send the output signal receive the input signal and not the processed signal, properly bypassing the effect.

# 2 - Creation of static GUI elements

The Graphic User Interface is based on an instance of the class *Window* which substantially constitutes the main container for everything that appears on screen. In detail, the window *w* has been built to contain the three pedal views, two frequency representation spectrums and two switch buttons(which are going to be addressed in chapter 4) inside, the rest of the GUI is instead developed in other children views.

## Window interface

The definition of the window *w* has been done by passing an instance of *Rect* to the Window class, specifying the desired dimensions, we purposely decided to not make the Window resizable as most of the plugins out there make this decision. In addition, we have furthermore customized the graphics with the use of the *Pen.fill* command, implementing the edges around *w* and around all the three pedals.

## Realization of the pedals

In order to represent the three pedals, we have decided to develop three different views called *view1*, *view2*, *view3*, respectively correlated to the phaser, the flanger and the wah wah effect. By exploiting the command *CompositeView*, we managed to put the pedals into our view w, in particular each pedal includes different elements inside:

* The text labels of the Synth
* The knobs that modify the range of the parameters
* The button that turns ON or OFF the related Synth

Firstly, to give some explanations, the text present in the pedals has been written by using the class *StaticText*, which allowed us to decently customize our interface (i.e. background color, font, alignment).

Secondly, the section regarding the knobs has been dealt through the *EZKnob.new* command, we have also attached some labels to those knobs, overall the specifics are the one 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 “depth”, 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 “amplitude”, the knob takes as value *ampLFO*, as range *specAmpf* and as initial value **‘0.0025’**

For *view3* (WAH-WAH effect) *:*

* Label “frequency”, the knob takes as value *dur*, as range *specDurw* and as initial value **‘0.5’**
* Label “amount”, 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.

The graphical and functional part of the buttons are enclosed in the *Button* command, making the color of the filling and of the text inside change on every transition “ON”/”OFF”.

## Frequency representation spectrum

In our multi effect we also thought of adding two frequency views in order to give to the user a clear representation of the signal changes made by the pedal. The two frequency analysis views have been developed through the *FreqScopeView* class, which receives the window *w* and an instance of *Rect* as parameters to create the desired structure.

The input spectrum is associated with the first input bus of the device on which the program is running, calculated through the variable *firstInputChannel*, while the output spectrum is associated with the output bus *~bout*, the role of each bus is going to be clearer in the next chapter. Both the spectras have been set to 120 db as far the amplitude range is concerned.

# 3 - I/O management of the chain of effects

In order to make the multi effect work in a series layout 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:

* Standard first input channel, 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

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.

The buses are defined as global variables so that we can access them both during the compilation of the code regarding the effects and the one responsible for the rest of the program.

Since the Flanger effect involves a delay effect in it, we had to add to the signal flow some sort of tool to make the effects coming after the flanger wait for the output coming out of it, this tool is the Group class. Thanks to the groups we can clearly define the order of execution of the various effects in the chain, fixing the aforementioned delay issue. The name of the groups clearly explain what they are meant to control, and they are: *~phaserGroup*, *~flangerGroup*, *~wahGroup*, *~outputAudioStereoGroup*

## 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 these in the two sections below. For reference, this section regards everything from line 389 onward.

## Graphical pedal swap

In order to make the pedals swap with one another we inserted two buttonscalled *leftSwitchButton* and *rightSwitchButton* in the gaps between the three pedals. Let’s take in consideration only *leftSwitchButton* w.l.o.g., in total there are 6 possible combinationsin 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. Therefore in the initial configuration we have the phaser followed by the flanger and the wah wah.

When the user clicks the *leftSwitchButton* we first take into consideration which are the first two numbers in the array, therefore 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 code 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 come the GUI and Audio changes.

We can shift the position of the pedals for each specific case by using the method .*moveTo* on the views, we called this method on the two leftmost views, properly making them swap. The same logic stands for the *rightSwitchButton*.

## Audio switch of the effects order

In order for the audio to follow the GUI we decided to enhance the just mentioned 6 cases of the switch buttons 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 accomplished 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 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 as we want them to be after the swap.

So basically, every time we click a swap button we are actually just recreating all the Synths anew in the correct order. Each case has its custom order of execution for the effects, therefore we have to do these steps for each one of them. As before, the same reasoning also applies to the *rightSwitchButton* cases.