BitWinds

A subtractive synthesis powered wind instrument emulator

Francesco Colotti *(francesco.colotti@mail.polimi.it)*
Gioele Fortugno *(gioele.fortugno@mail.polimi.it)*
Matteo Gionfriddo *(matteo.gionfriddo@mail.polimi.it)*
Emanuele Greco *(emanuele2.greco@mail.polimi.it)*

May 23, 2023

In this report, we will briefly present our implementation of a subtractive-synthesis-powered wind instrument emulator in SuperCollider.

Our synth, which we nicknamed *Bitwinds*, uses one oscillator and noise, each shaped by two separate series of two filters, to attempt to mimick a variety of wind instruments sounds. A selection of effects is then available to make the sound more complex.

1 Sound Design

The sound design process focused on the creation of two synth definition, each for a different family of instruments. In particular, here we will focus on a generic flute and a generic brass sound. Our decisions were taken accounting for the physical characteristics of the instruments and a special emphasis was put in the creation of a model that could be extensively modified by the user and not be strictly fixed.

1.1 Pan Flute

To design the sound of a pan flute, we started from its physical model. A pan pipe can be considered as a pipe closed at one side and open at the other. Therefore, it can be proven that standing waves appear inside whenever someone blows into the pipe, with a fundamental wavelength of four times the length of the pipe itself. From acoustics theory, we know that in open-closed pipes there can only be standing waves with frequencies that respect the following law:

$$f_n = (2n+1)f_0$$

One can immediately notice that, for $n = 0, 1, ..., +\infty$, only odd harmonics are present. For this reason, we chose a square wave as the main wave of our synthesized sound, since the coefficients X_k of its Fourier series are:

$$X_k = \frac{A}{2}\,sinc\,(\frac{k}{2})$$

which go to zero for even values of k. To do this, we use the `Pulse.ar` UGen. Obviously, a simple generator of square waves is not enough to reproduce the behaviour of a real wind instrument. One of the main factor that characterizes the sound of a pan pipe is turbulence, that adds a strong noise component to the final sound. A correct interaction between noise and sound is thus needed. Subtractive synthesis can be used to 'carve' the noise (e.g. white noise) in such a way that its most prominent component are highlighted. As explained in [1], most of the noise is audible at higher frequencies; so, we pass our white noise signal through a gentle high-pass filer, with cutoff frequency of $hpf\_freq = freq + (20000 - freq) * 0.4$, where $freq$ is the fundamental frequency of the square wave. Furthermore, to give the noise a "breathy" quality, we send our high-pass filtered signal through a bank of six band-pass filters, centered at three octaves (including the fundamental) and their relatives fifths; this emulates the sound of air passing through. An ADSR envelope is then applied to the final output.

Another important feature of BitWinds is the noisy "chiff", independent both from the tonal and noise component, that is heard at first when a pan pipe is played. In order to reproduce this, the high-passed noise is filtered through a low-pass filter which frequency is controlled by a AD(Attack-Decay) envelope, which is also applied to the the output signal of the filter. Finally, we mix the chiff with the output from the band-pass filter bank.

As for the tonal component, the square wave is passed through a low-pass filer with cutoff frequency `freq` and we apply some vibrato using a LFO. An ASR (Attack-Sustain-Release) envelope is used, with a longer attack than the noise. Finally, the tonal and noise component are mixed proportionally to an optimum ratio.

1.2 Trumpet

From the physical and spectral analysis of the sound of a trumpet, we gather that it has a complete harmonic series, with components even at higher frequencies[1]. This characteristic restricts our choice to a (filtered) Sawtooth wave.

Analyzing how the "brightness" of the sound, i.e. the amount high frequencies in the spectrum, evolves over time we can generalize the behaviour saying that the harmonics beneath the instrument's natural cutoff, given in the code by the experimentally found value of `freq+(10000-freq)*0.18`, reach their sustain level together and more quickly. Moreover, the sound emitted by a trumpet has a small high frequency portion, given by the initial high pressure pulse of air emitted by the player. This behaviour is simulated applying an ADSR(Attack-Decay-Sustain-Release) envelope to the frequency of the low-pass filter. The steady state in the sustain phase of the envelope, after a peak in the attack phase. Beside that, another modulation occurs in the filter. When a brass instrument is played there's a small but noticeable instability period where the wave produced reaches its steady state. This is simulated by another modulation of the filter frequency. In this case we use the low frequency Triangle wave generator `LFTri.kr`, modulated by a ramp envelope of fixed duration, in order to affect only the initial part of the sound and immediately shut off.

The amplitude response of the sound is controlled by a classic four stage ADSR envelope, with short attack and decay, and modulated by a low frequency sinusoidal oscillator `SinOsc.kr`, delayed in order to affect only the sustain phase of the sound, to give the sound a slight tremolo effect. Finally, We mix the sound produced with a turbulent noise component, obtained through the same method shown before, that is a White Noise passed through six bands band-pass filters.

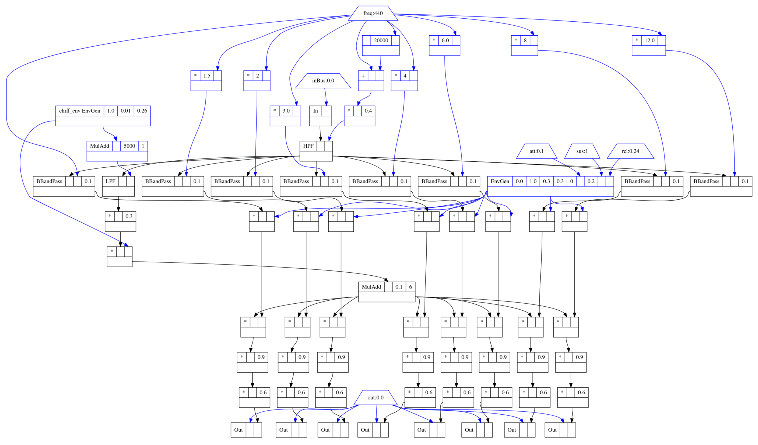


Figure 1: UGen graph of the noise component of the Pan Flute

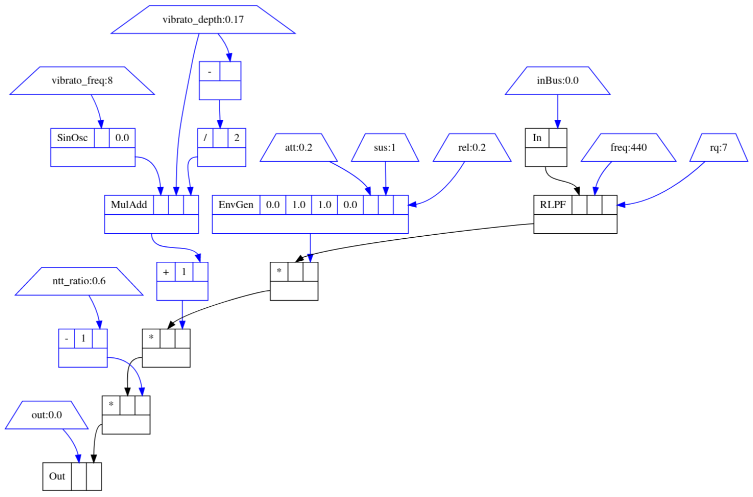


Figure 2: UGen graph of the tonal component of the Pan Flute

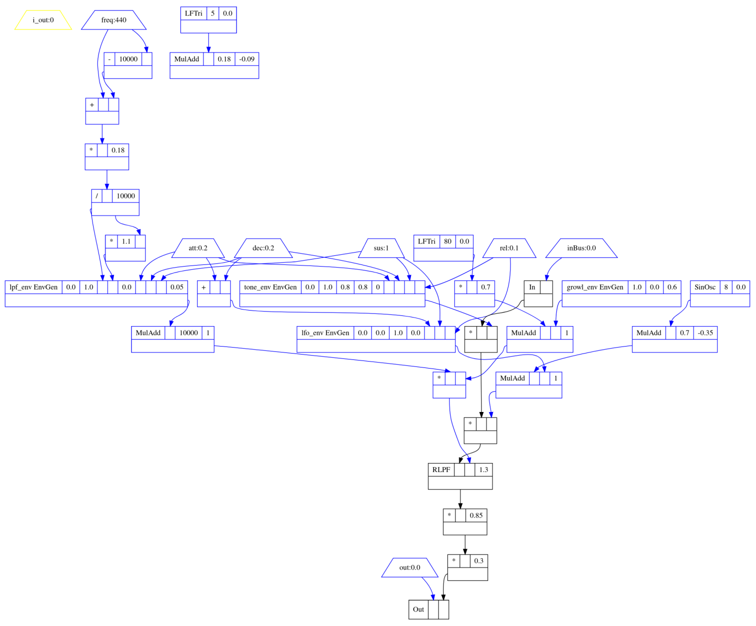


Figure 3: UGen graph of the tonal component of the Brass

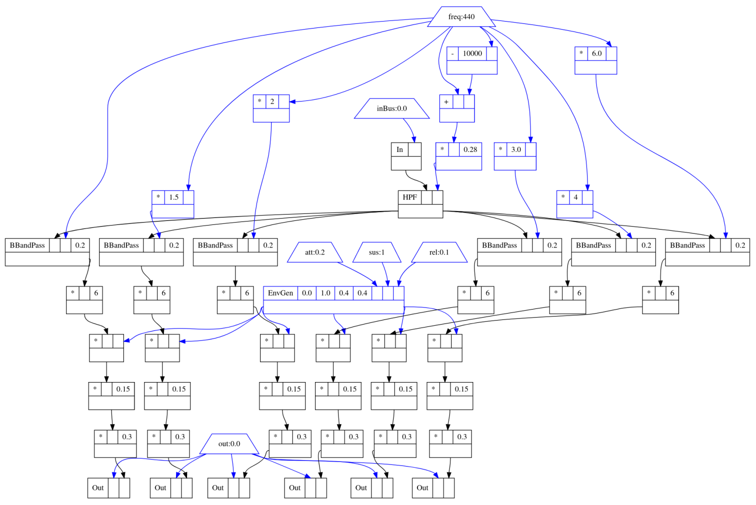


Figure 4: UGen graph of the noise component of the Brass

2 Filters

In this section, we will talk about the signal processing made by the filters. In particular we will explain the routing of the signal between the various stages and what kind of filters we used.

Our sound source consists of two oscillators: waveform and noise. To give more possibilities to the user, we have decided to create a filtering system consisting of two filters in series to both oscillators (two *per* oscillator). This way we are able to work on different frequencies and thus alter the two sounds independently.

All four filters are constructed the same way. Firstly, we have a drop-down menu that allows the user to choose the type of filter (high-pass or low-pass only). Then, for each filter, we have a bypass button that can be used to activate or deactivate it. This way we can decide if our source goes directly to the effects chain (see Section 3) or gets processed first by the filter(s). The user can control each filter cutoff and resonance. The cutoff is set from a minimum frequency of 20 Hz to a maximum of 20000 Hz, while the resonance value is between 0 and 1.

For the low-pass filter we used a UGen named RLPF [2], which is a low-pass resonant filter.

For the high-pass filter we used a UGens named HLPF [3], which is a high-pass resonant filter.

2.1 Organization of the filters signal flow

The filters signal flow is organized into multiple SynthDefs and buses, one for each filter. Each one reads the signal from its input bus and writes it to the input bus of the next stage until it reaches the final output bus (input bus of the first effect).

As we mentioned, the user is able to bypass a filter in the chain; in that case the block in question is replaced by a new "dummy" SynthDef which sends the signal on its input bus directly to the output one (input of the next block). By doing this we can still "forward" the signal when a filter (or both) is deactivated.

We included a graphical representation of the signal flow in Figure 5.

The SynthDefs are allocated in groups that are the client-side representation of a group node on the server, which is a collection of other nodes organized as a linked list. The Nodes within a Group may be controlled together, and may be both Synths and other Groups. In total we have created 4 groups that include all the SynthDefs created. Group "c" includes the SynthDefs that generate the initial sound of the synthesizer: osc and noise. The "u" group includes the SynthDefs dedicated to make the timbre and create the envelope of the sound. the "g" group includes all the SynthDefs dedicated to the creation of low-pass and high-pass filters. Group "o" includes all the SynthDefs dedicated to effects. Groups allowed us to achieve the correct order of execution of each processing block.

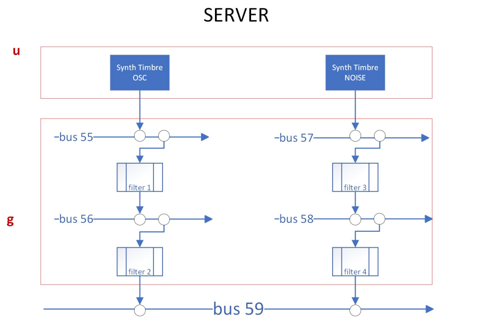


Figure 5: Representation of the signal flow through buses

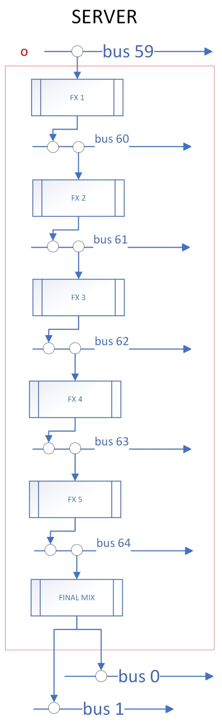
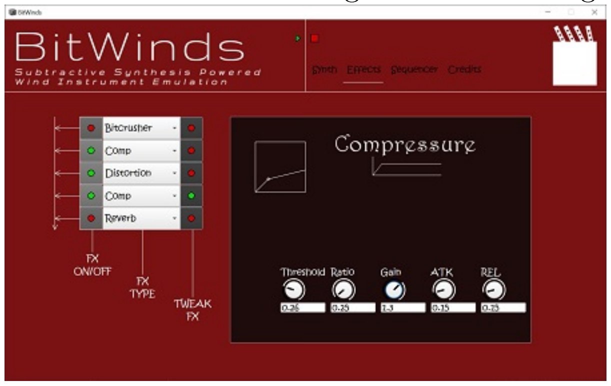


Figure 6: Block diagram of effects



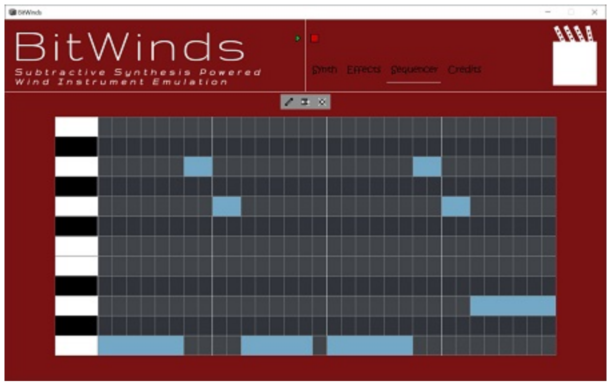


Figure 7: The "Effects" and "Sequencer" Tabs

3 Effects

For sound processing we have prepared a chain (series) of effects. Each effect is associated to a slot. The user can modify some parameters of each effect, change the effect associated to the slot and of course turn each slot on or off. Effects are implemented, from a practical viewpoint, using an array (*fx_params*), where for each slot we save a dictionary containing the effect type and its parameters.

More specifically four effects are made available to the user: a bitcrusher, a compressor, distortion and a reverb.

The bitcrusher uses the Decimator UGen [4]. It can control the effective sample rate and reduce the bit depth of the input, and can thus be used to transition between a clean signal and a lofi one.

The compressor uses the Compander UGen [5] which represents a general purpose (hard-knee) dynamics processor.

The distortion uses the CrossoverDistortion UGen [6] which is a simulation of the distortion that happens in class B and AB power amps when the signal crosses 0.

The reverb uses the FreeVerb UGen [7]. We tried using more complex reverbs, such as the GVerb Ugen, but weren't able to make them work as flawlessly as FreeVerb.

The Decimator and CrossoverDistortion UGens are taken from the *SC3plugins* extension [8], which is a requirement for this synth.

A full explanation of the arguments of these UGens is out of the scope of this report, but their documentation can be found in the references of this report.

3.1 Organization of effects flow

The effect signal flow is organized similarly to the filter chain described in Section 2. Specifically we have 5+1 buses, one per slot plus the global out bus for the final stage, and 4 SynthDefs, one per type of effect. The bus number is associated with the input of the slot using global variables. The output signal of the filtering stage is written to the bus of the first slot and then the signal is passed on sequentially in the chain in the same manner. In order to be able to use only some effects or even none, the signal can be routed to a "dummy" SynthDef which takes the signal input bus of the slot directly to its output. For the last stage we created a SynthDef in which the signal is made stereo and processed by a limiter.

A scheme of the effects signal flow can be observed in Figure 6.

4 The Graphical User Interface

The Graphical User Interface for the project is presented to the user in a single window divided in 4 tabs ("Synth", "Effects", "Sequencer", "Credits"). Each part can be accessed by pressing a button on the top of the View that refreshes the GUI. The graphical elements have been created by stacking UserViews on top of each other, all on the same window. We chose to use these in order to be able to use the Pen object to draw lines, squares and basic geometrical elements in a manner similar to Canvas in HTML.

Whenever the user changes tab, three functions are called: *changeWind*, *showPart* and *hidePart*.

The function *changeWind* simply hides unneeded UserViews and, only the first time a tab is loaded, initialises knobs, static texts and other elements for that tab.

After initialisation, *showPart* and *hidePart* are used to show the elements of a tab and hide the elements of the previously shown one. These two call more specific functions, respectively *showEffects* and *hideEffects*, to hide the GUI components of the current visualised effect or show them. Additionally, *showEffects* also load the parameters value for a certain slot from memory to correctly set the knobs (this is also called when switching between slots).

4.1 Synth GUI

The synth GUI is rather direct and leaves very little to the imagination. The user is able to set the ASR envelope (no decay!) of an oscillator and a noise source, then can morph the sound using 2 filters per side.

The filters are applied only to the sound source on their side.

4.2 Effects GUI

In the "Effects" part the user is able to build a chain using the 4 effects we talked about in Section 3. A part of the window is used to provide the user with the tools (e.g. knobs) needed to customise the effect; this is implemented by stacking a second UserView on top of a section of the main one covering the whole window. While the compressor, distortion and reverb are pretty straightforward and provide the user with a set of knobs only (see the provided code), the bitcrusher uses a less technical GUI, providing the user with a grid and a dot they can move to set the sample rate and number of bits.

4.3 Sequencer GUI

The sequencer covers one octave (actually only from C to B) and allows the user to place, resize and delete notes with three available tools. The octave is represented by a Dictionary object containing 12 entries (1 per note), to each of whom we associate an Order object (which is actually an automatically ordered Dictionary). The Order contains all the events (sounds to be played) for the Note it is associated to, using [location, duration] couples.

The sequencer has 32 steps and represents a full measure in a 4/4 metre. For the sake of simplicity we chose a harsh quantisation for events (events can only happen at integer locations, durations can also only be integers).

We implemented a function, *notesToSeq*, to port all the events into a single array for playback. This function exploits the Order properties to find the next event, then packs it into an array in the [note, location, duration] format (in the case of simultaneous events the array will contain multiple arrays). A variable keeps track of the current octave, which can be changed with two buttons, and it is simply added to an offset to obtain the MIDI note number.

We recommend looking at the code to get a better idea of how the full algorithms works, both for the UserViews (correct placement of elements, decorations, etc.) and the sequencer (managing note insertions, deletions and resizes, and converting them into a sequence). For convenience, we included two screenshots of the GUI (effects and sequencer tabs) in Figure 7.

5 Conclusion

Our emulation of a wind instrument not only aims to provide the user with a timbre similar to that of some real instruments but is also versatile for creating other sounds through the addition of modulable filters and effects. In particular, we can shape the length of the notes by creating long sound textures with the variable ASR and then shape the sound through customizable chains of effects.

Our approach uses subtractive synthesis, but other types of synthesis are of course also viable to recreate the timbre of wind instruments using different methods.

References

[1] *Synthesizing Wind Instruments.* URL: https://www.soundonsound.com/techniques/synthesizing-wind-instruments (visited on 2023-04-20).

[2] *Low pass filter - SuperCollider Documentation.* URL: https://doc.sccode.org/Classes/RLPF.html (visited on 2023-04-20).

[3] *High pass filter - SuperCollider Documentation.* URL: https://doc.sccode.org/Classes/RHPF.html (visited on 2023-04-20).

[4] *Decimator - SuperCollider Documentation.* URL: https://doc.sccode.org/Classes/Decimator.html (visited on 2023-04-20).

[5] *Compander - SuperCollider Documentation.* URL: https://doc.sccode.org/Classes/Compander.html (visited on 2023-04-20).

[6] *CrossoverDistortion - SuperCollider Documentation.* URL: https://doc.sccode.org/Classes/CrossoverDistortion.html (visited on 2023-04-20).

[7] *FreeVerb - SuperCollider Documentation.* URL: https://doc.sccode.org/Classes/FreeVerb.html (visited on 2023-04-20).

[8] *sc3-plugins GitHub Repository.* URL: https://github.com/supercollider/sc3-plugins (visited on 2023-04-20).