Dai Fujikura
Prism Spectra
for viola & live electronics

Commissioned by Ircam/Centre Georges Pompidou
World premiere performed in June 2009 at Auditorium du Musée d'Orsay (Paris)

Viola : Odile Auboin
Computer music designer : Manuel Poletti
Sound ingeneer : Gérard Delhia
Duration : 18 mn

This documentation describes the procedures needed to install, setup and perform the live-electronics used in the piece.

About the live-electronics

In section 1, 5 and 6, the musician is constantly "followed" by the live electronics, which perform a kind of sound extension of the live instrument. They perform some spatialized live effects such as granulation, harmonization, delays, frequency shifting, filtering..., which continuously improvise (given certain contraints) "around" the live sound, in order to produce an homogenic overall material, together with the live sound.

In addition, pitch, attack and amplitude detection is performed, in order to modulate the density and the choice of the improvisation of the treatments, and their spatialization. There are also some independant algorithms that will reverberate or not a given channel of a given module at a given time, as well as the live instrument. In section 3, the musician triggers a semi-random collection of viola pre-recorded samples each time the computer detects that a string is attacked. Sections 2 and 4 are made of soundfiles that will accompany the player like if there was an string ensemble playing.

To summarize the approach of the live electronics design, we could say that a maximum of flexibility (with musical constraints) was put into the control of the modules, in order to get an "organic" sound material, which remains close to the instrument sound. They will sound slightly different at each performance.

About the live electronics performer

The live-electronics performer should take care of the following (described in details below) :

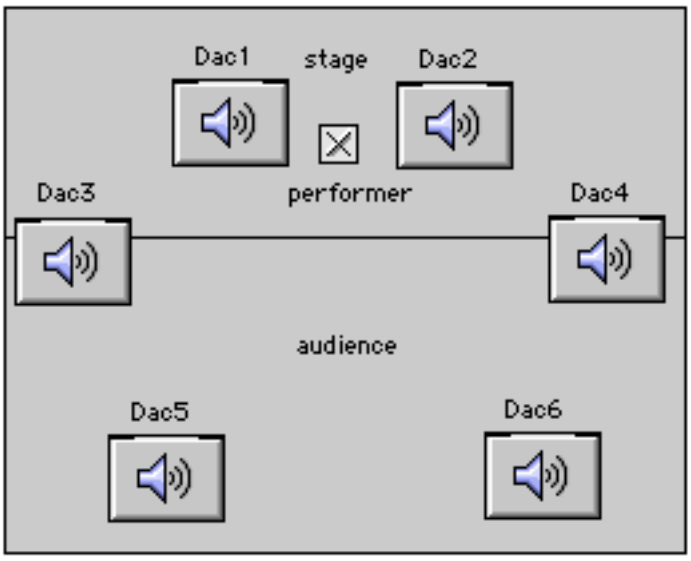
- install and setup the MaxMSP application, as well as the specific MaxMSP resources contained in the archive.
- launch the performance patch and setup some settings (gains, MIDI pedal...)
- trigger the cues (or events) during the performance, according to the score
- cues might be triggered by the musician on stage, using a MIDI trigger pedal : in this case, the live electronics performer follows the events and corrects the eventual mistakes from the musician
- cues are mentionned as numbers in the original score, which are reported within the patch

System requirements

- 1 Macintosh Intel MacPro computer 4X 2.5 GHz (minimum), with 1 Gb RAM (minimum), running OSX 10.5 or later - **the patch will NOT perform well (lack of CPU resources) with an Intel MacBookPro 2.4 GHz, for instance, but may work on latest MacBookPros.**
- 1 MaxMSP 4.6.3 or Max 5 or later application
- 1 soundboard able to output 6 separate audio channels (ex : RME Fireface 400), with 1 MIDI input
- an appropriate wireless and classA microphone - a wireless DPA was used at the premiere
- 1 mixing console with up to 8 separate output audio busses
- 6 loudspeakers (ex : 600 W) & appropriate P.A (including subwoofers)
- 1 MIDI trigger pedal

About the sound

All sound material, except for the live sound, is spatialized over 6 speakers. One should amplify the live sound using the two frontmost speakers (1 & 2 below). The shape of the overall plan formed by the speakers should be similar to a classical 5.1 system, except that the center point is synthesized by the two frontmost speakers, rather by only one speaker.



- Viola microphone goes to adc 1

- dac1 goes to loudspeaker 1 (preferably matrix of loudspeakers)
- dac2 goes to loudspeaker 2 (idem)
- dac3 goes to loudspeaker 3 (idem)
- dac4 goes to loudspeaker 4 (idem)
- dac5 goes to loudspeaker 5 (idem)
- dac6 goes to loudspeaker 6 (idem)

It is highly recommended to diffuse the electronics to a matrix of triples of loudspeakers rather than assigning each computer dac to one loudspeaker. This means that, for example, the Dac 1 from the computer soundboard should be sent to speaker 1, and to speaker 2 and 3, with around 3/6 dBs energy less in speakers 2 and 3 than in speaker 1. Then, Dac2 should be sent to speaker 2, and to speaker 1 and 4, with around 3/6 dBs less energy in speakers 1 and 4 than in speaker 2. And so on (triples : 3+1/5, 4+2/6, 5+3/6, 6+4/5) , until the electronics sound "spatial" and "smooth".

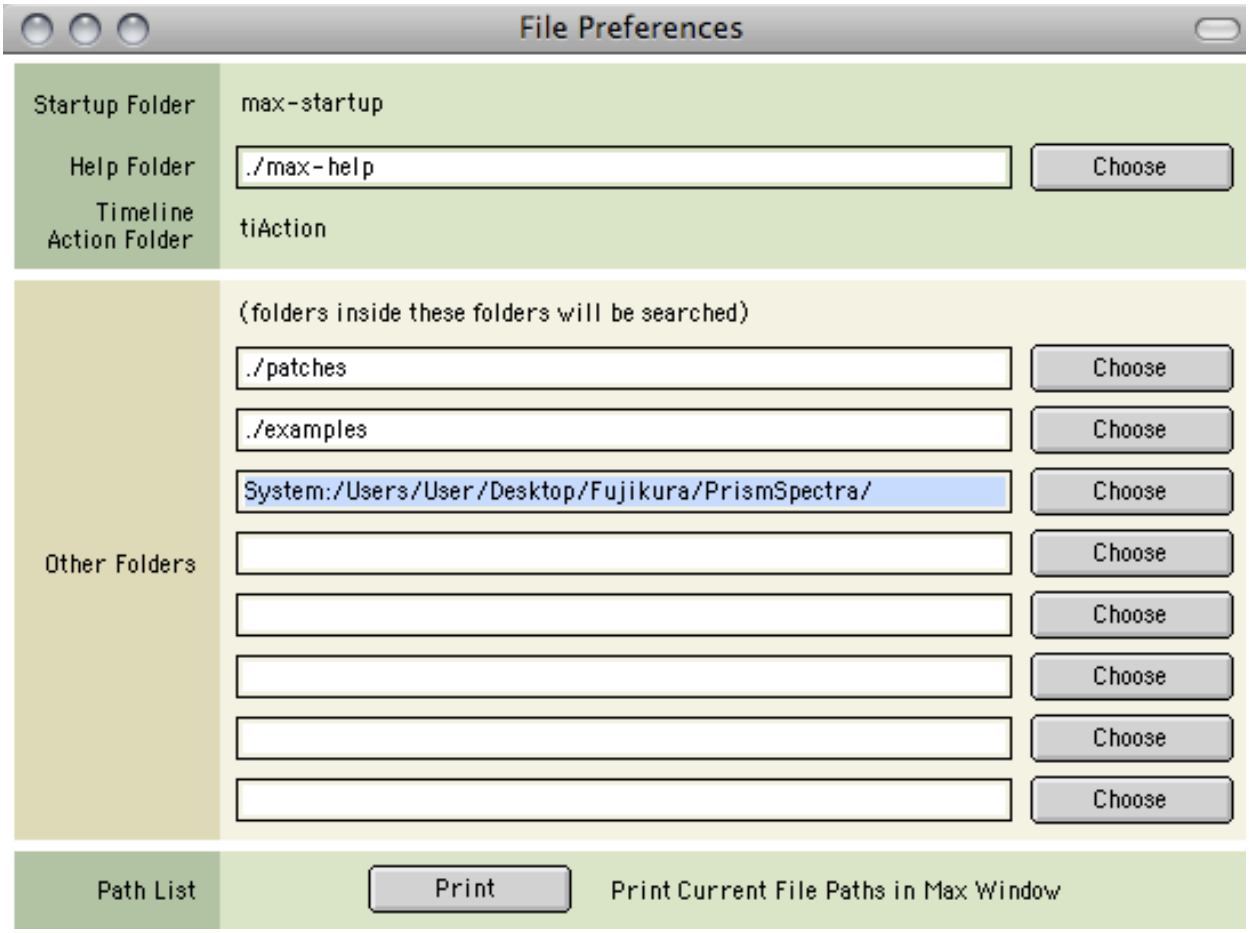
The player should have a pair of stage monitors, in order to hear a mix of the live electronics while performing.

Contents

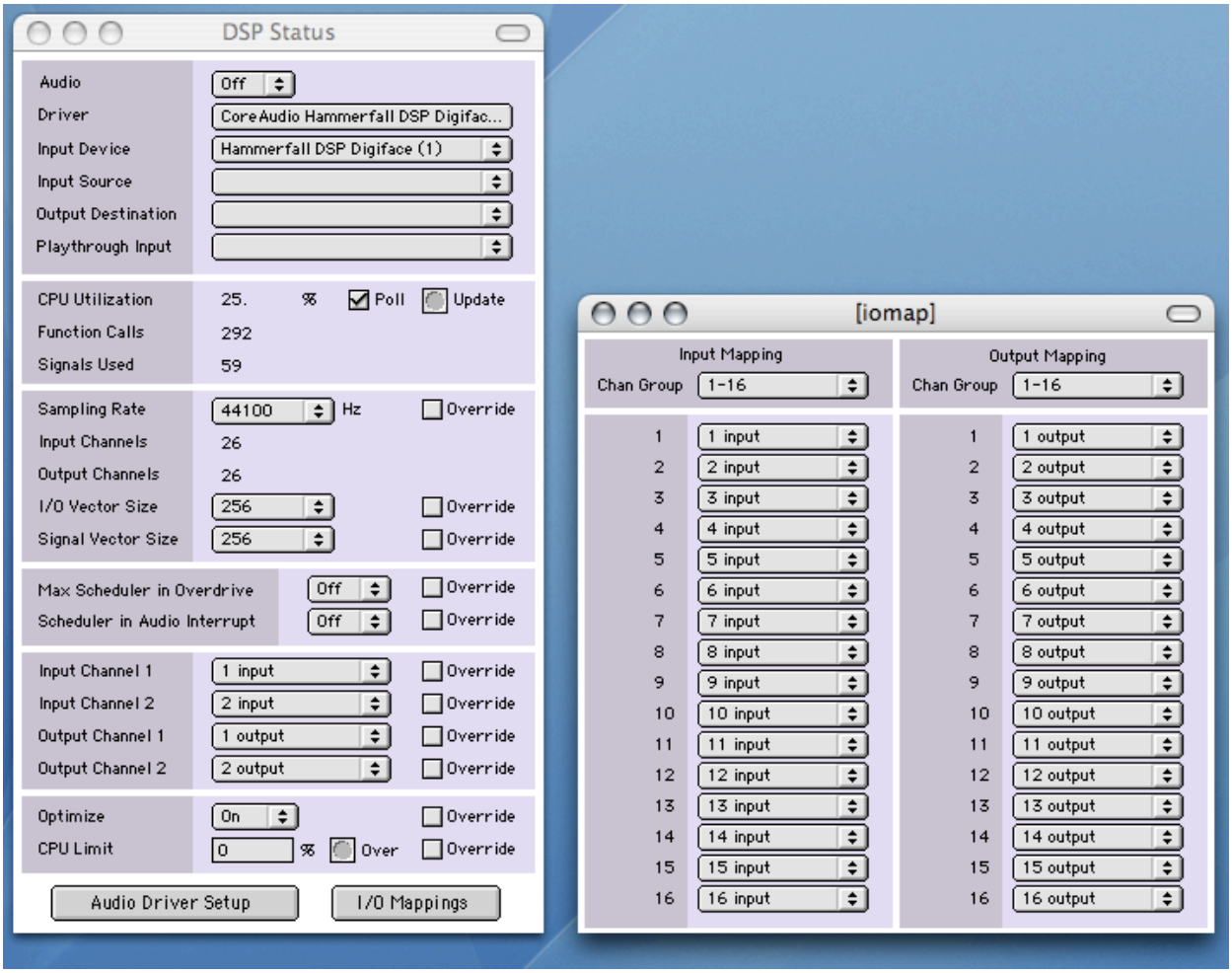
The archive contains all resources needed to perform the live-electronics of the piece within MaxMSP.
The **MaxMSPConcert** folder contains everything needed for running the concert patch. Inside that folder, there are two main Max documents :
These patches are the performance patches :
- PrismSpectraMax4UB runs under Max 4.6.3
- PrismSpectraMax5 runs under Max 5 and later
Select the Max4 or Max5 version according to the version of MaxMSP you have.

Installation

- download and install MaxMSP 4.6 or 5
- copy the content of the archive somewhere in a folder on your hard disk
- launch MaxMSP
- declare the folder containing the concert patches and their related files in the Max file preferences, as follows (for instance) :



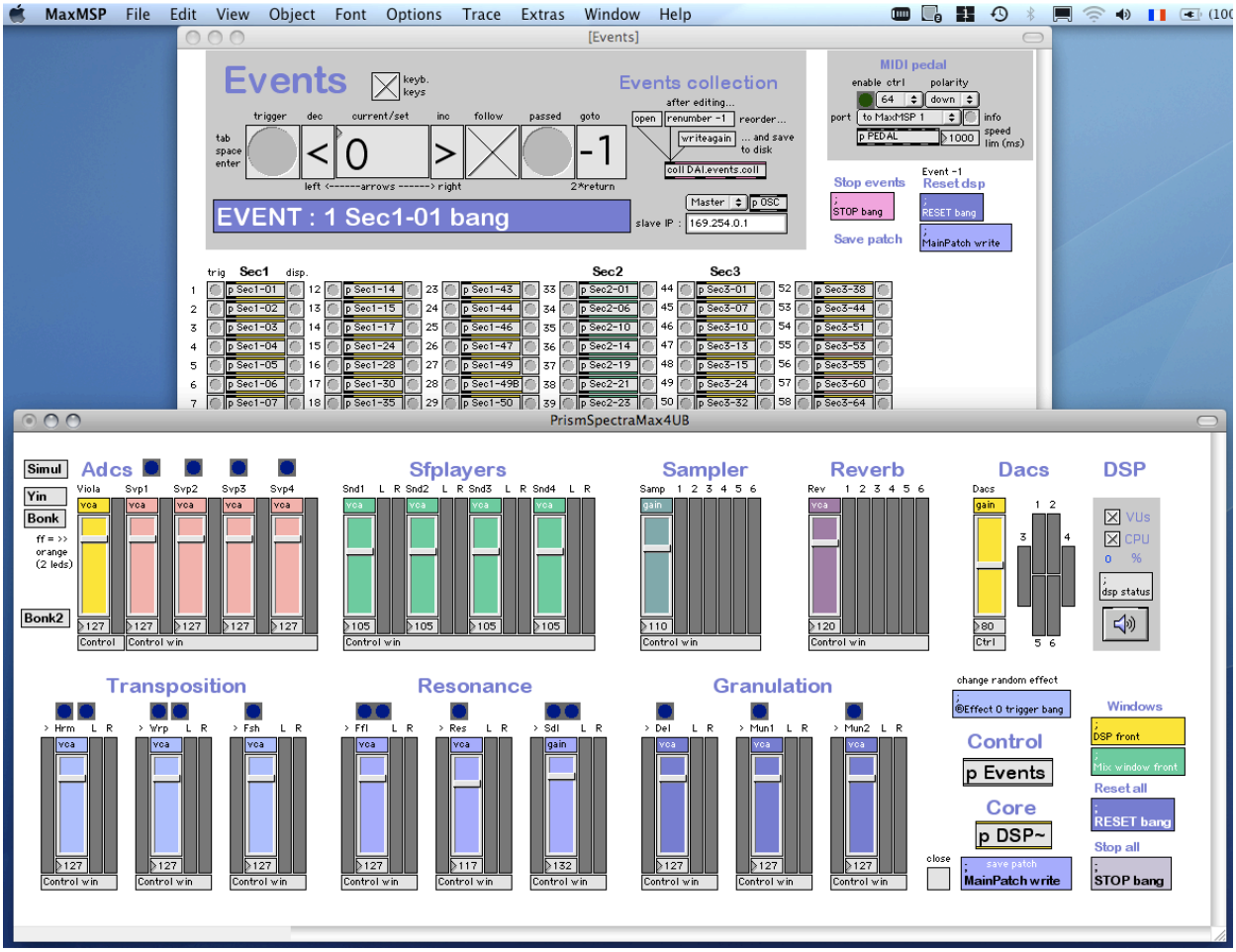
- set the DSP status (using the driver of your soundboard) as follows :



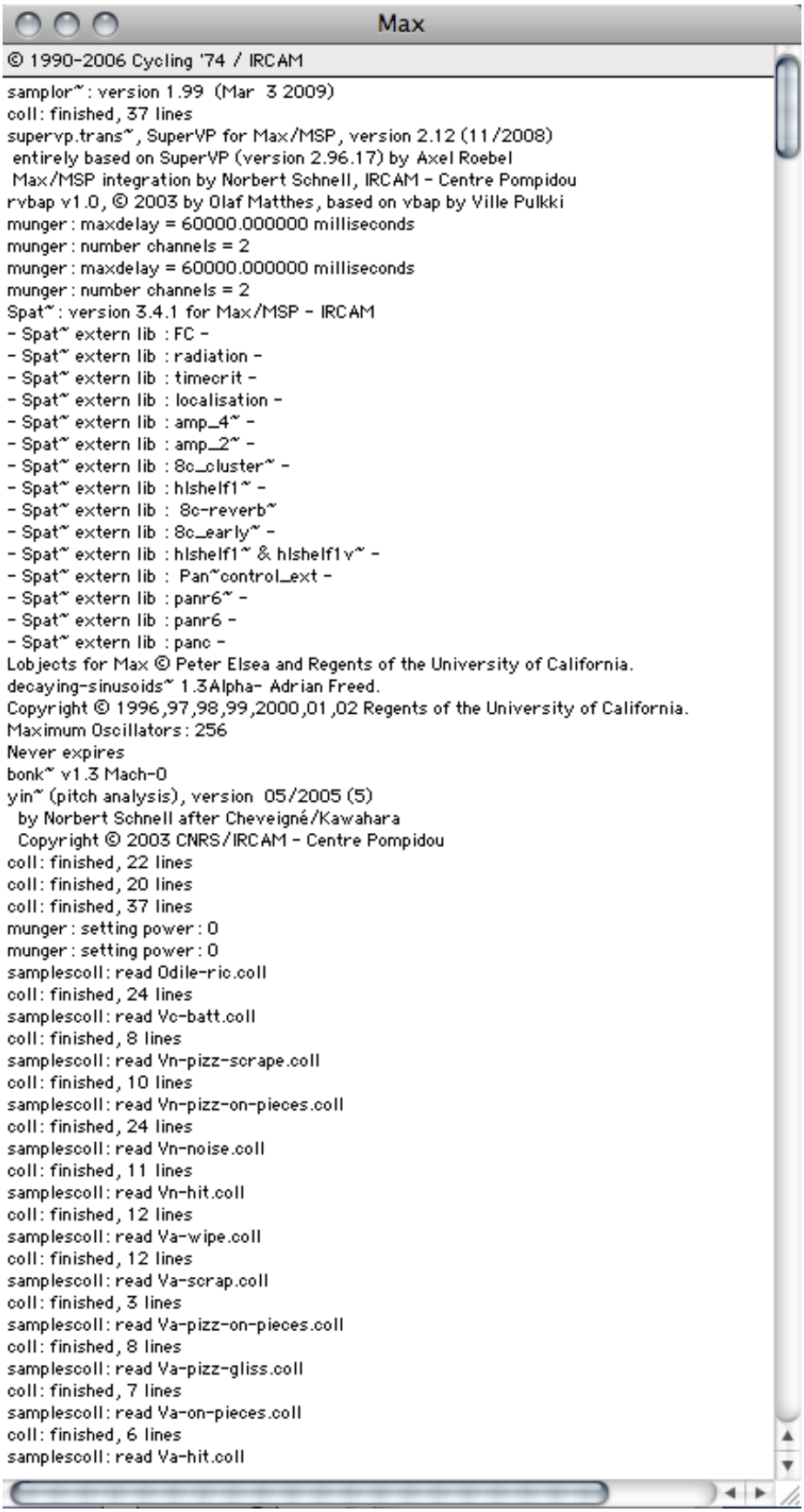
Quit and relaunch MaxMSP in order to validate the preferences.

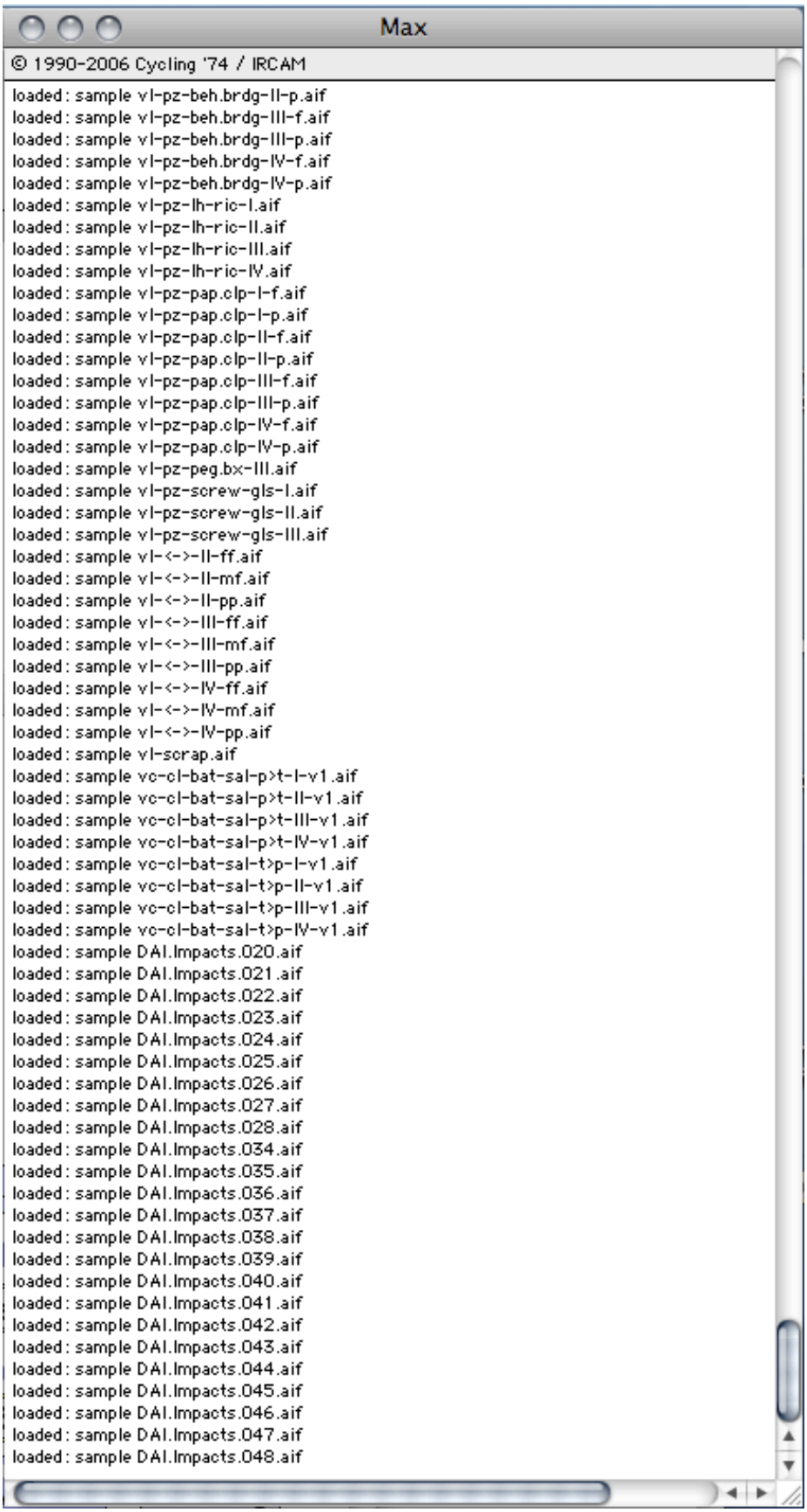
DSP Patch

Launch the PrismSpectraMax4UB or PrismSpectraMax5 patcher file, according to the version of Max (4 or 5) you're using; The Max4 patcher should look like this (the Max5 patch is equivalent) :



In order to verify if preferences files are correct, open the Max window, which should look like this :


Max
© 1990-2006 Cycling '74 / IRCAM
samplor~: version 1.99 (Mar 3 2009)
coll: finished, 37 lines
supervp.trans~, SuperVP for Max/MSP, version 2.12 (11/2008)
entirely based on SuperVP (version 2.96.17) by Axel Roebel
Max/MSP integration by Norbert Schnell, IRCAM - Centre Pompidou
rvbap v1.0, © 2003 by Olaf Matthes, based on vbap by Ville Pulkki
munger: maxdelay = 60000.000000 milliseconds
munger: number channels = 2
munger: maxdelay = 60000.000000 milliseconds
munger: number channels = 2
Spat~: version 3.4.1 for Max/MSP - IRCAM
- Spat~ extern lib : FC -
- Spat~ extern lib : radiation -
- Spat~ extern lib : timecrit -
- Spat~ extern lib : localisation -
- Spat~ extern lib : amp_4~ -
- Spat~ extern lib : amp_2~ -
- Spat~ extern lib : 8c_cluster~ -
- Spat~ extern lib : hlshelf1~ -
- Spat~ extern lib : 8c-reverb~
- Spat~ extern lib : 8c_early~ -
- Spat~ extern lib : hlshelf1~ & hlshelf1v~ -
- Spat~ extern lib : Pan~control_ext -
- Spat~ extern lib : panr6~ -
- Spat~ extern lib : panr6 -
- Spat~ extern lib : panc -
Lobjects for Max © Peter Elsea and Regents of the University of California.
decaying-sinusoids~ 1.3Alpha- Adrian Freed.
Copyright © 1996,97,98,99,2000,01,02 Regents of the University of California.
Maximum Oscillators: 256
Never expires
bonk~ v1.3 Mach-0
yin~ (pitch analysis), version 05/2005 (5)
by Norbert Schnell after Cheveigné/Kawahara
Copyright © 2003 CNRS/IRCAM - Centre Pompidou
coll: finished, 22 lines
coll: finished, 20 lines
coll: finished, 37 lines
munger: setting power: 0
munger: setting power: 0
samplescoll: read Odile-ric.coll
coll: finished, 24 lines
samplescoll: read Vc-batt.coll
coll: finished, 8 lines
samplescoll: read Vn-pizz-scrape.coll
coll: finished, 10 lines
samplescoll: read Vn-pizz-on-pieces.coll
coll: finished, 24 lines
samplescoll: read Vn-noise.coll
coll: finished, 11 lines
samplescoll: read Vn-hit.coll
coll: finished, 12 lines
samplescoll: read Va-wipe.coll
coll: finished, 12 lines
samplescoll: read Va-scrap.coll
coll: finished, 3 lines
samplescoll: read Va-pizz-on-pieces.coll
coll: finished, 8 lines
samplescoll: read Va-pizz-gliss.coll
coll: finished, 7 lines
samplescoll: read Va-on-pieces.coll
coll: finished, 6 lines
samplescoll: read Va-hit.coll

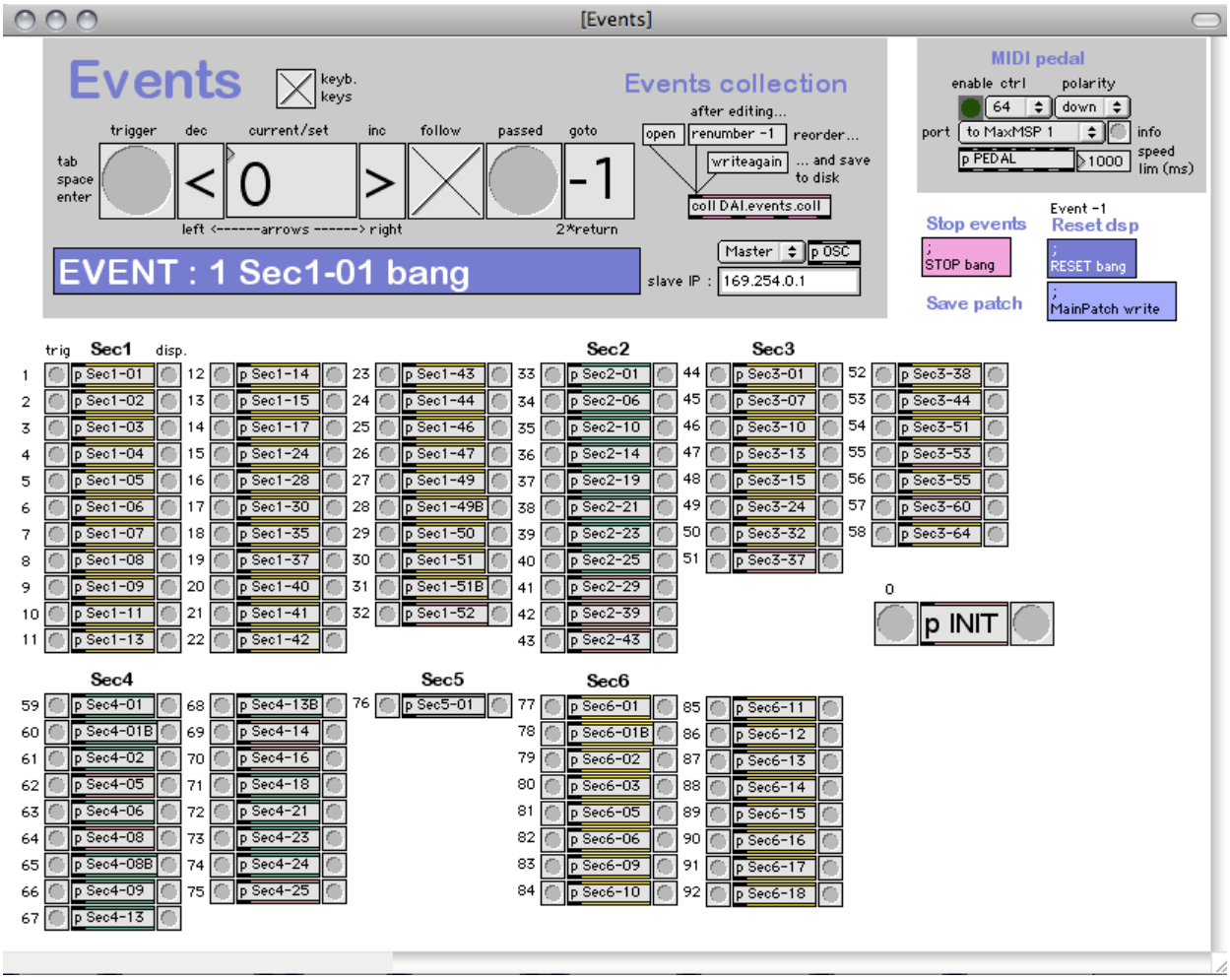

Max
© 1990-2006 Cycling '74 / IRCAM
loaded: sample vl-pz-beh.brdg-II-p.aif
loaded: sample vl-pz-beh.brdg-III-f.aif
loaded: sample vl-pz-beh.brdg-III-p.aif
loaded: sample vl-pz-beh.brdg-IV-f.aif
loaded: sample vl-pz-beh.brdg-IV-p.aif
loaded: sample vl-pz-lh-ric-I.aif
loaded: sample vl-pz-lh-ric-II.aif
loaded: sample vl-pz-lh-ric-III.aif
loaded: sample vl-pz-lh-ric-IV.aif
loaded: sample vl-pz-pap.clp-I-f.aif
loaded: sample vl-pz-pap.clp-I-p.aif
loaded: sample vl-pz-pap.clp-II-f.aif
loaded: sample vl-pz-pap.clp-II-p.aif
loaded: sample vl-pz-pap.clp-III-f.aif
loaded: sample vl-pz-pap.clp-III-p.aif
loaded: sample vl-pz-pap.clp-IV-f.aif
loaded: sample vl-pz-pap.clp-IV-p.aif
loaded: sample vl-pz-peg.bx-III.aif
loaded: sample vl-pz-screw-gls-I.aif
loaded: sample vl-pz-screw-gls-II.aif
loaded: sample vl-pz-screw-gls-III.aif
loaded: sample vl-<->-II-ff.aif
loaded: sample vl-<->-II-mf.aif
loaded: sample vl-<->-II-pp.aif
loaded: sample vl-<->-III-ff.aif
loaded: sample vl-<->-III-mf.aif
loaded: sample vl-<->-III-pp.aif
loaded: sample vl-<->-IV-ff.aif
loaded: sample vl-<->-IV-mf.aif
loaded: sample vl-<->-IV-pp.aif
loaded: sample vl-scrap.aif
loaded: sample vc-cl-bat-sal-p>t-I-v1.aif
loaded: sample vc-cl-bat-sal-p>t-II-v1.aif
loaded: sample vc-cl-bat-sal-p>t-III-v1.aif
loaded: sample vc-cl-bat-sal-p>t-IV-v1.aif
loaded: sample vc-cl-bat-sal-t>p-I-v1.aif
loaded: sample vc-cl-bat-sal-t>p-II-v1.aif
loaded: sample vc-cl-bat-sal-t>p-III-v1.aif
loaded: sample vc-cl-bat-sal-t>p-IV-v1.aif
loaded: sample DAI.Impacts.020.aif
loaded: sample DAI.Impacts.021.aif
loaded: sample DAI.Impacts.022.aif
loaded: sample DAI.Impacts.023.aif
loaded: sample DAI.Impacts.024.aif
loaded: sample DAI.Impacts.025.aif
loaded: sample DAI.Impacts.026.aif
loaded: sample DAI.Impacts.027.aif
loaded: sample DAI.Impacts.028.aif
loaded: sample DAI.Impacts.034.aif
loaded: sample DAI.Impacts.035.aif
loaded: sample DAI.Impacts.036.aif
loaded: sample DAI.Impacts.037.aif
loaded: sample DAI.Impacts.038.aif
loaded: sample DAI.Impacts.039.aif
loaded: sample DAI.Impacts.040.aif
loaded: sample DAI.Impacts.041.aif
loaded: sample DAI.Impacts.042.aif
loaded: sample DAI.Impacts.043.aif
loaded: sample DAI.Impacts.044.aif
loaded: sample DAI.Impacts.045.aif
loaded: sample DAI.Impacts.046.aif
loaded: sample DAI.Impacts.047.aif
loaded: sample DAI.Impacts.048.aif

If the Max window prints some errors, then you probably have a conflict in your Max Search Path and should check the Preference Files again.

Events Subpatcher

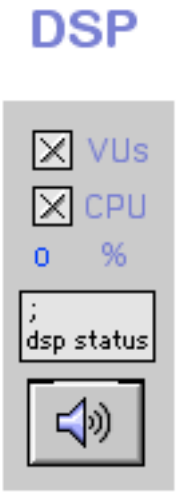
If you **double-click on the Events subpatcher box** located in the main patcher patcher file...

...the Events window is brought to the front. This is the place where the cues are triggered from, in the case where the performer won't trigger the cues using the pedal.

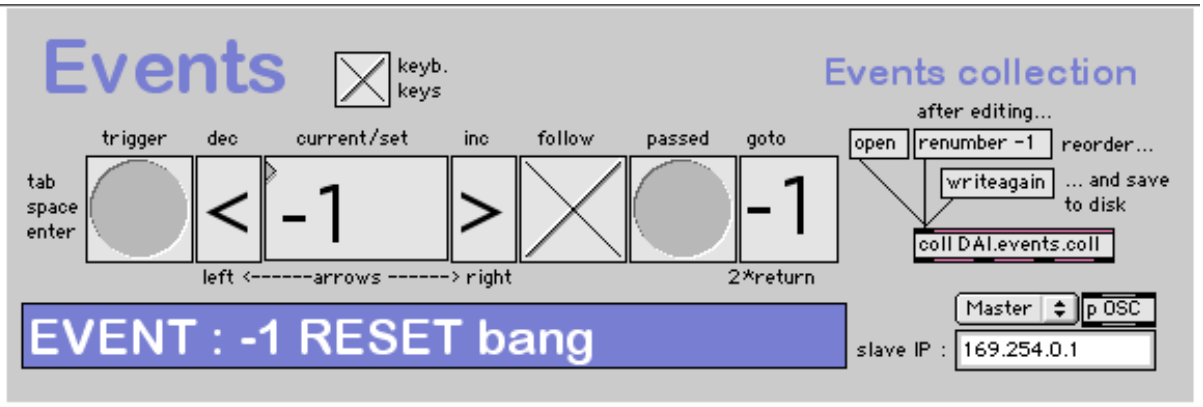


To start the piece :

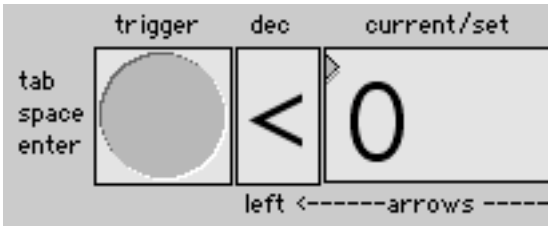
Turn DSP on in the main patcher :



Press the event "-1"button in the Events window, which performs a complete initialization of the whole patch :



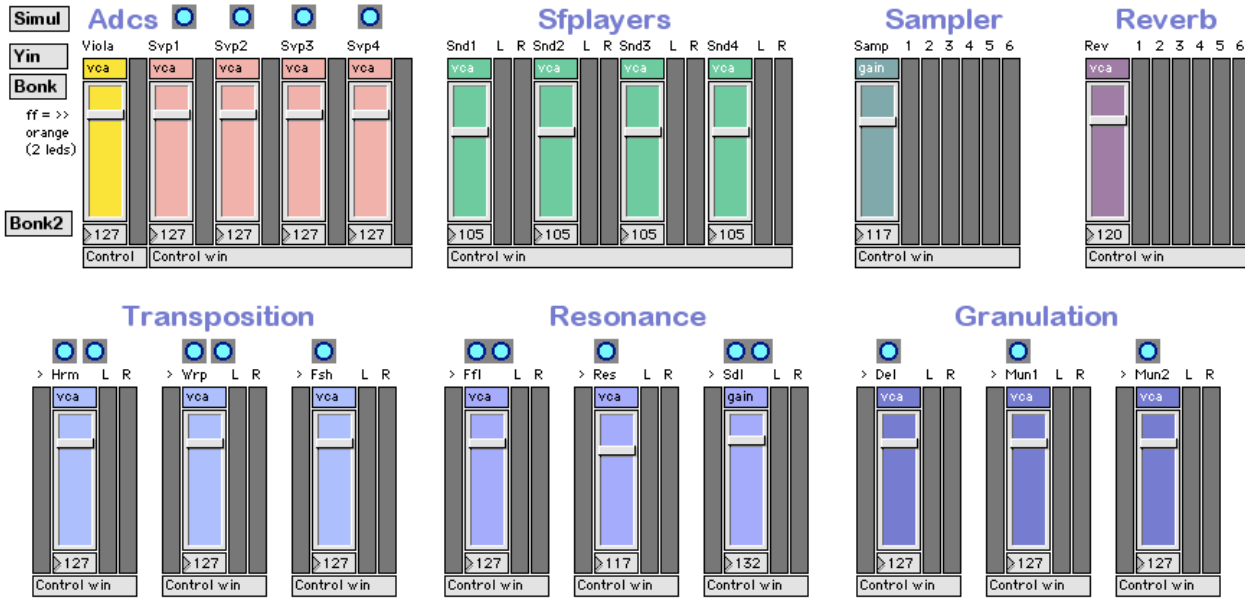
Trigger event "0" using the "trigger button", which performs a local initialization of the event patch :



You're ready to play the piece (event 1 and others).

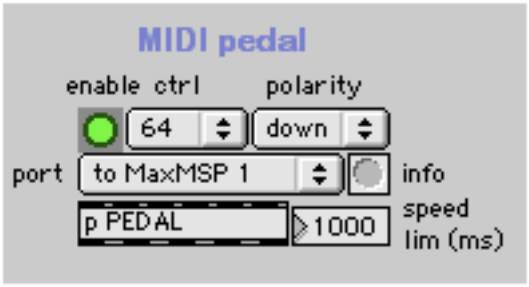
Mixing

The events patcher was designed in order to set the gains of the different modules from the DSP patch automatically.
One shouldn't have to control them during the performance but, according to the live conditions, you may want to adjust the output levels of the modules.
To do so, use these sliders :



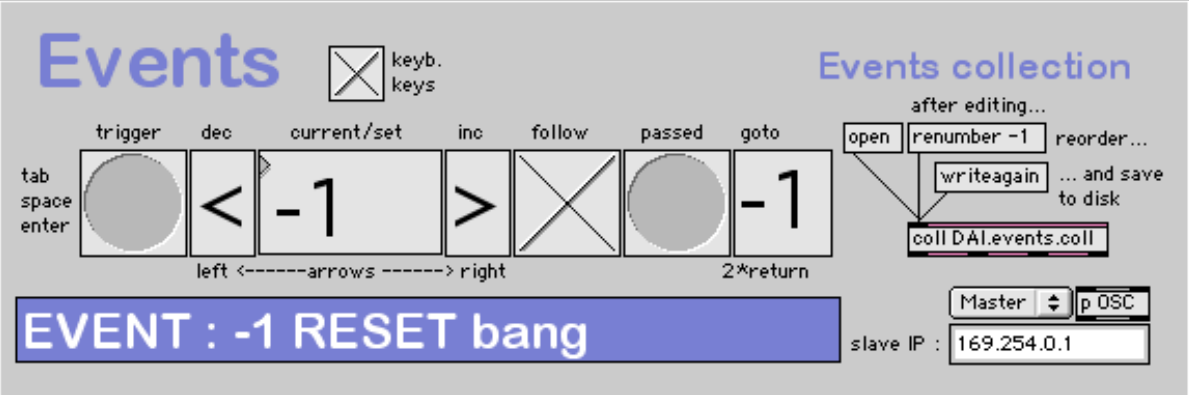
Using a MIDI pedal to trigger events

If the performer triggers the events, you can set the pedal parameters from the event patcher :



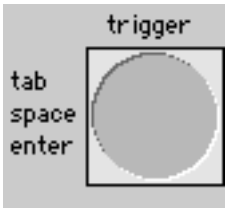
- enable : turn pedal action on/off
- ctrl : choose MIDI controller number (generally, trigger pedals output a controller 64)
- polarity : if necessary, inverse the polarity of the pedal : generally the event must be triggered when the pedal is pressed (some pedals have inversed polarities)
- port : select the MIDI port to which the pedal is connected
- info : click to refresh the list of available MIDI ports
- speedlim : set the minimum time between two events, in order to avoid mistakes due to wrong moves from the performer

Managing events

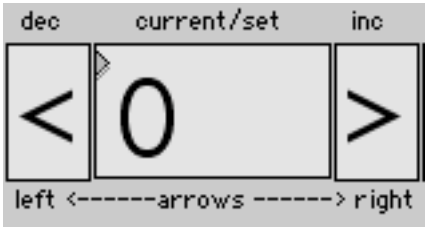


When you start the piece from event -1 (global reset), then event 0 (local init), the next events will trigger some soundfiles and open or close treatments.

To trigger each next event, either press the "trigger" button, or press the space, enter or tab key from the computer keyboard :

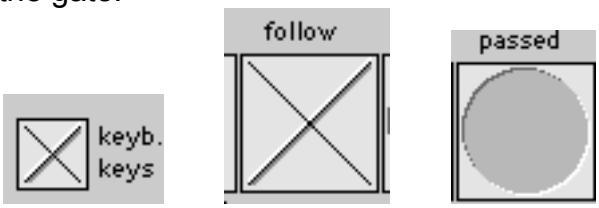


If you need to jump to a given event number, use the big numerical box to set the NEXT event to be triggered the next time you will press the "trigger" button - use the inc/dec buttons to scroll through the events (or use the left/right arrows from the keyboard) :

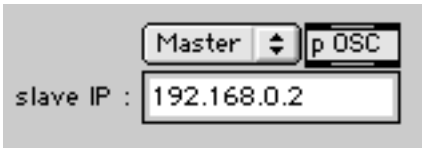


Using inc/dec buttons can be very useful, in case the performer misses a trigger - instead of trying to play the event (too lately), just set the next event to be triggered, then wait for the next pedal.

By default, "keyboard keys" and event follower toggles are turned on : "follow" operates as a gate when one tests the MIDI pedal, for instance : turn it off, and you'll see the events being increased by the pedal, but the events themselves won't be triggered - "keyboard keys" eventually prevents that no one accidentally triggers an event by touching the computer keyboard. The "passed" button indicates that an event had "passed" the gate.



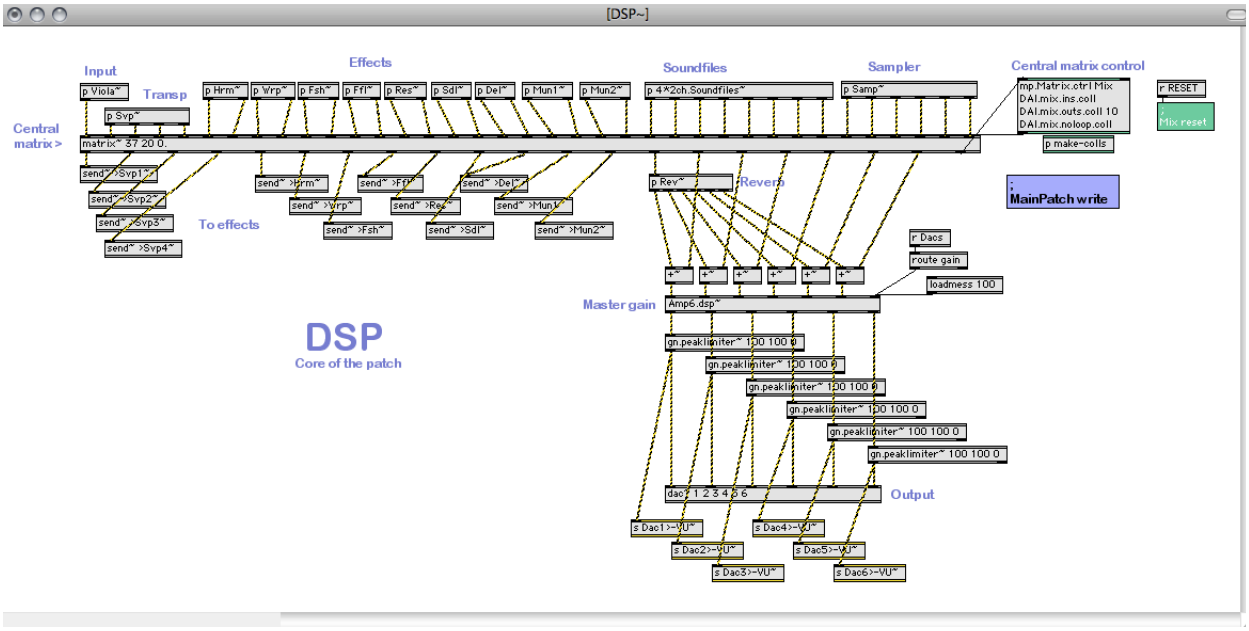
If you're using a parallel spare machine, event triggers can be sent in parallel from the master machine to another one using UDP via network. In both machines, select its status (master or slave). In the master machine, enter the IP adress of the target spare machine (visible in the System Preferences - sharing section). Then save the patch again, in order to validate the target IP adress :



How the patch works

Below is a description of the structure of the patch, which shows how the piece should be performed, and what to take care of.

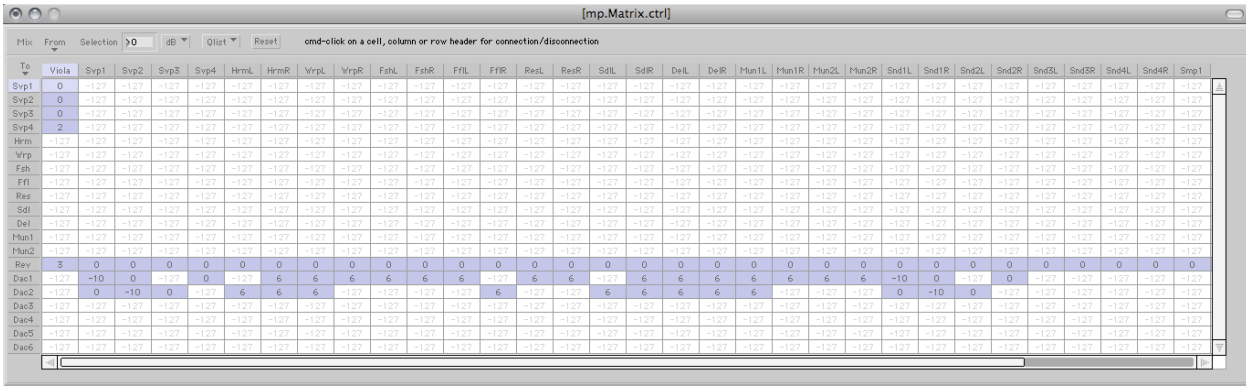
The core of the patch is an audio matrix, which allows the routing the audio input to any effect or output. It stands, as well as all the modules connected to it within the **DSP~ subpatcher**, and looks like this :



Basically, the Viola~ Adc will be routed to the other modules (Hrm~, Wrp~, Fsh~...) through the matrix and using the send~ objects. In parallel, the output of the effect modules will be routed to the dacs through the matrix. There's also a global 6 channels reverb module, which will be mixed together with the output channels, 4*2 channels soundfiles (DFD) players, and a 6 channels sample player (RAM).

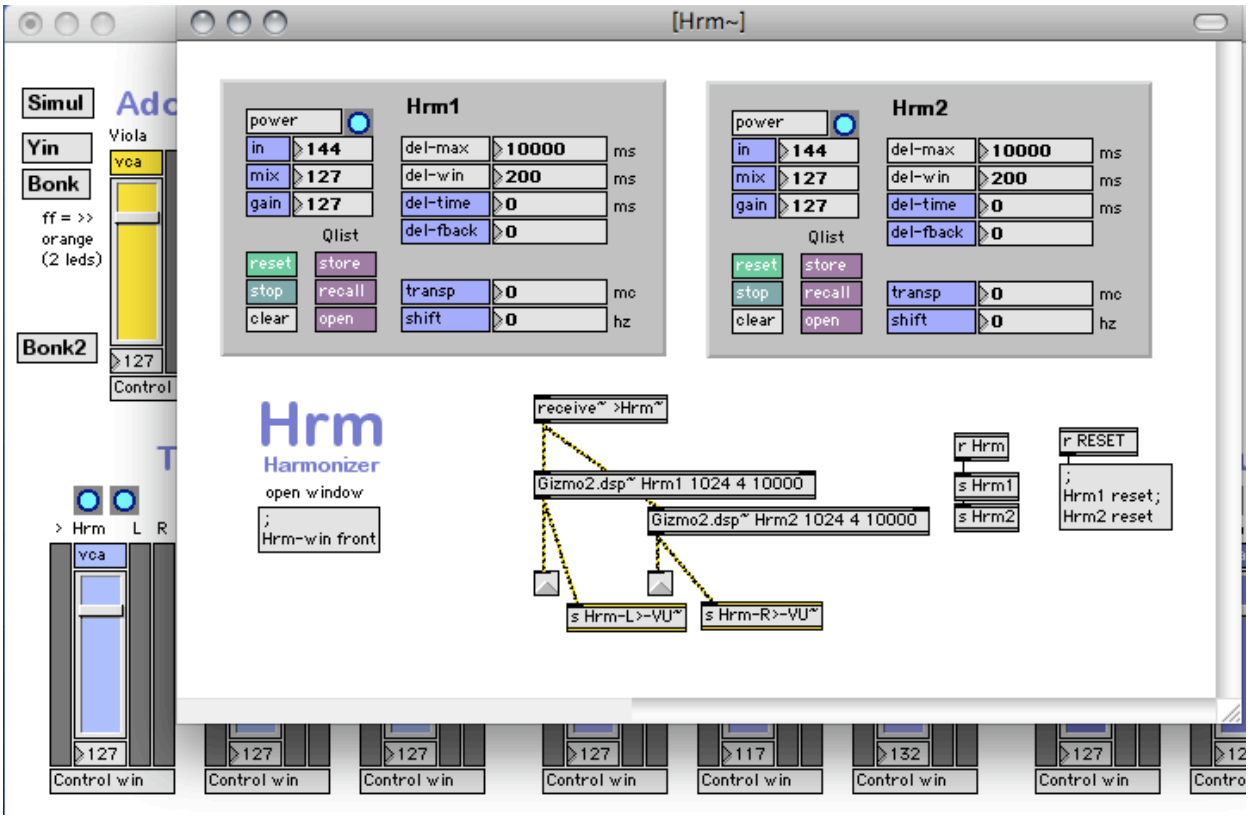
The routing of the matrix is controlled and monitored using a graphic interface (accessible here in the mp.Matrix.ctrl object). One can display that interface from within the main patch by clicking in that button :

which will open the Matrix window :



Horizontally, you see the outputs of the modules to be routed to the inputs of other modules, vertically. If you cmd-click on a cell, you will toggle between a connection level (by default 0db) and a disconnection level (by default -127 dB). During the performance, all connections are made automatically, so you shouldn't use that interface but for monitoring what's going on in the patch.

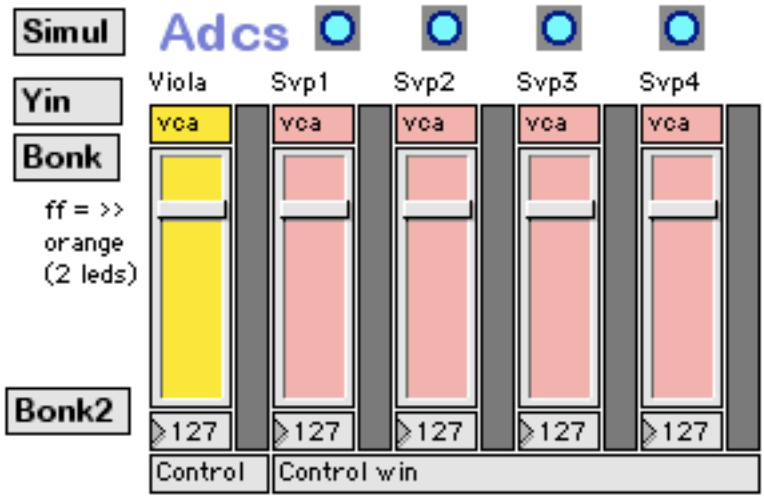
Each of the modules is accessible either by double-clicking on the objects within the DSP~ subpatcher, or by clicking the "Control win" shortcut buttons within the main patcher. Here, if we click on the "Control win" button below the "Hrm" gain slider at the left bottom part of the patch, the window that contains the Hrm module opens up :



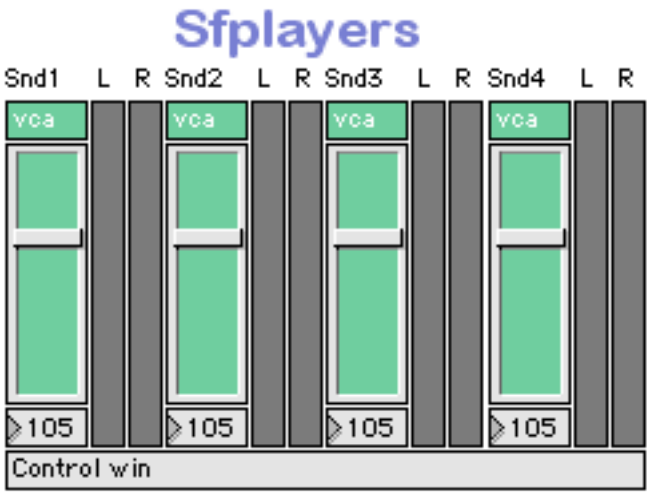
Each module has a dsp object (here 2 "Gizmo2.dsp" objects), with a receive~ object (that is remotely linked to the matrix), and two "VU" send objects (used to monitor the signal within the main patcher). The dsp objects have their corresponding interface, which allows you to display and control the parameters.

Here is a list of the modules used :

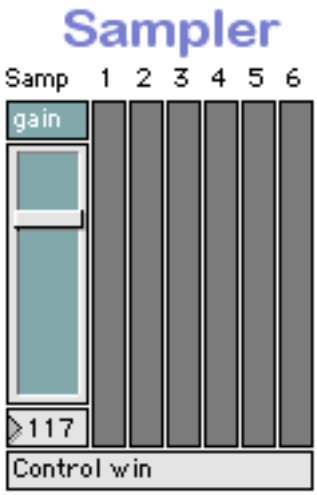
Viola : the level control of the input signal - additionally, the signal is measured by three detection modules : "yin" for pitch and amplitude detection, and 2 "bonks" for attack detection.
Simul : can be used to play an input soundfile, in order to feed the patch and simulate the action of the performer.
Svp 1 to 4 are "virtual" adcs, which transpose the input sound. These transposers are used to simulate the presence of a virtual string quartet. They can be transformed by the effects, just like the real sound :



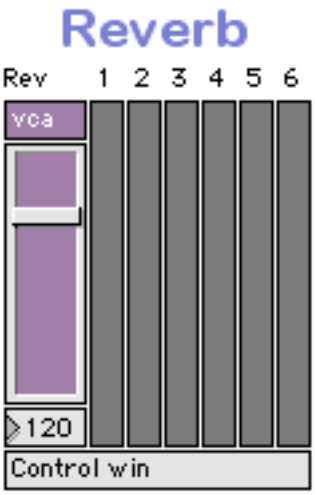
Sfplayers : 4 stereo DFD soundfile players that will play some pre-recorded soundfiles :



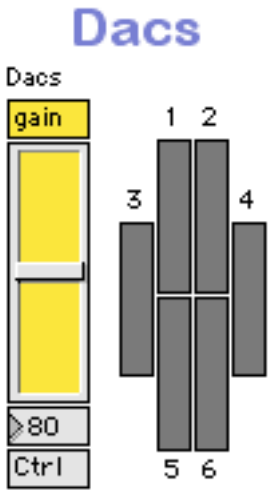
Sampler : a 6 channels RAM sample player, whose samples are triggered by the attack detection in Section 3.



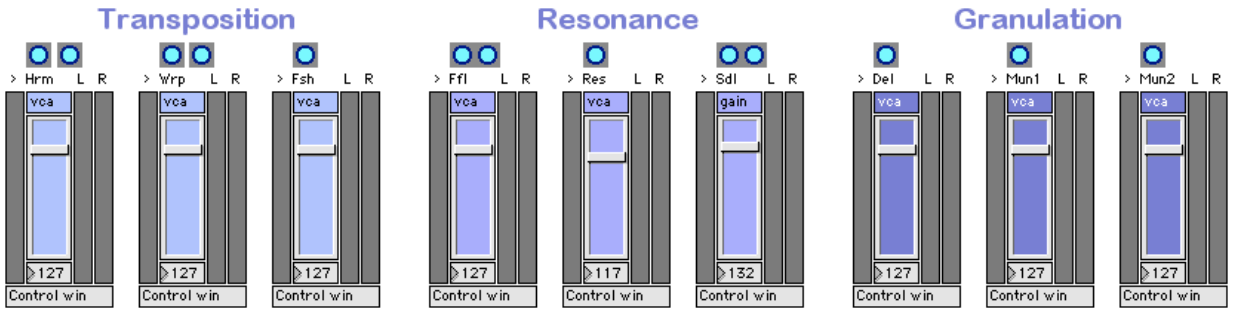
Reverb : a 6 channels reverberator, that reverberates the overall electronics sound



Dacs : the overall dacs control



Effects : these effects transform the input sound and the "virtual" SVP string quartet
Hrm : a stereo FFT harmonizer/pitch shifter with feedback delay
Wrp : a stereo FFT "frequency warper" (compresses/expands the signal spectrum)
Fsh : a stereo frequency shifter
Ffl : a stereo FFT 250 bands filter with feedback delay
Res : a stereo FFT "resonator" (which is like the frequency warper with additional features)
Sdl : a stereo FFT spectral delay
Del : a stereo multitaps delay
Mun1 & Mun2 : two stereo granulators



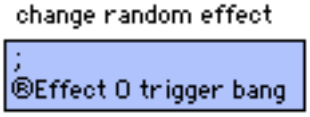
The aim of the patch

The main idea behind the patch is that the live electronics should remain "capricious". Thus, a set of Max objects (in addition of the dsp modules) were specially developed in order to fit that idea. These objects perform some random walks of dozen of parameters for effects and matrix routing. They react to attack detection, and generate rhythms which are relative to a global tempo. They're located in the Init subpatcher in the Events window :

The whole process can be summarized as following :

The attack detector will try to follow the performer's impacts onto the instrument, in order to perform some constraint random choices among the collection of sound treatments. So, ideally, each time (s)he would hit the instrument, we could decide wether not to route the incoming audio signal to such or such sound treatment. So the result for each part of the piece would be always slightly different, given some constraints : at that part, the signal might be routed to the harmonizer, or to the delay, or to none of them. The parameters that describe the behaviour of the harmonizer and the delay should also "move" given some constraints. There are several sections, for instance, which potentially play with all effects, with "moving" parameters for each of them. What we tell to the program in most of the events of the piece, is the constraint, rather than a precise settings. We've extended that behaviour to many aspects of the audio rendering, for instance : for each impact, any input (adc or effect) of the matrix will be routed or not to the reverberator. Another use of the attack detection is to route randomly the outputs of the modules towards the speakers : not "which" speaker, but rather "when" ("which" is described below).

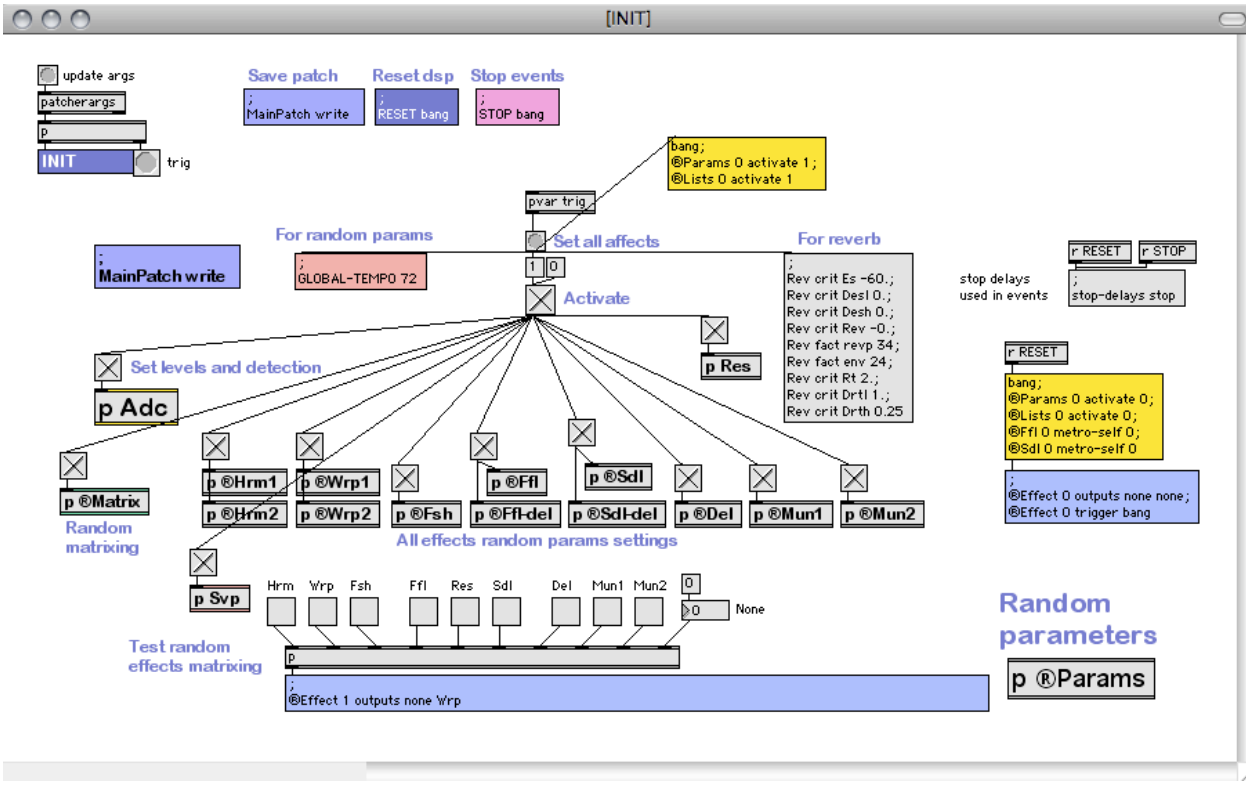
Note that all along the piece, not all input sound might trigger enough attack detections. So, all random parameters have internal metronomes that can force them to change the state of the effects. When an attack is finally detected, it will restart each metronome, thus leaving the "dynamic" aspect of the interaction intact. If one thinks that a given situation is not sounding very well, one can always trigger a new random choice by clicking that button, and monitor its effect within the Matrix window :



The amplitude follower (Yin) is used for two main tasks, according to the above : tell the matrix which loudspeakers the outputs of the effect modules should be sent to, and how fast the random effect changes should occur. The constraint is easy : the louder the input sound is, the "wider" the live electronic space will be, and the more effect changes you get. The spatialization of the effects is calculated by loudspeakers pairing - basically : piano sound will send the electronics to the front speakers, mezzo forte sound will send them both to the front and side speakers, and forte sound will send them into all speakers. The effect one should obtain is that the performer "projects" the electronics across the room according to how loud (s)he plays.

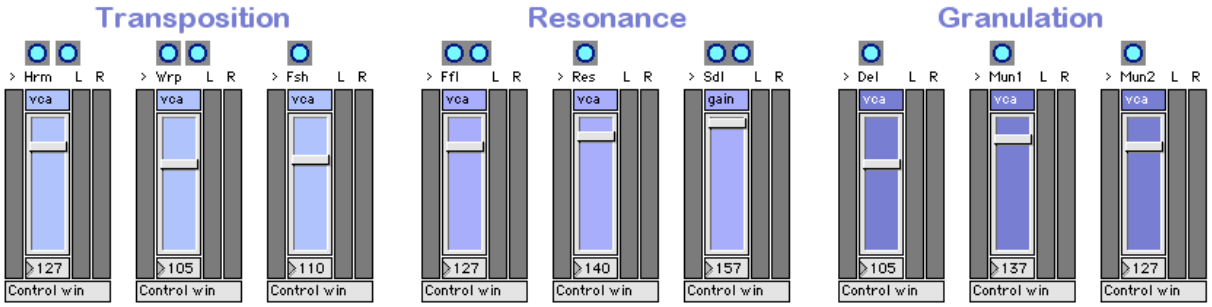
Test it

In **the Events/INIT subpatcher**, you find all the settings that control the expected behaviour described above. In order to perform the test, turn DSP on and click in the button on the left.



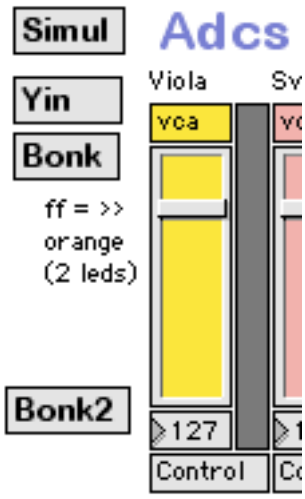
Here, you can see the overall settings for the whole piece. The Adc subpatch will set the interactive part (attack and amplitude detection actions), the ®Matrix subpatch will set the routing behaviour for the matrix, and other subpatches will set the random paths of the parameters of each effect module.

Together with the INIT being triggered, all effect modules get activated - and remain so, which explains the need for powerful machines :



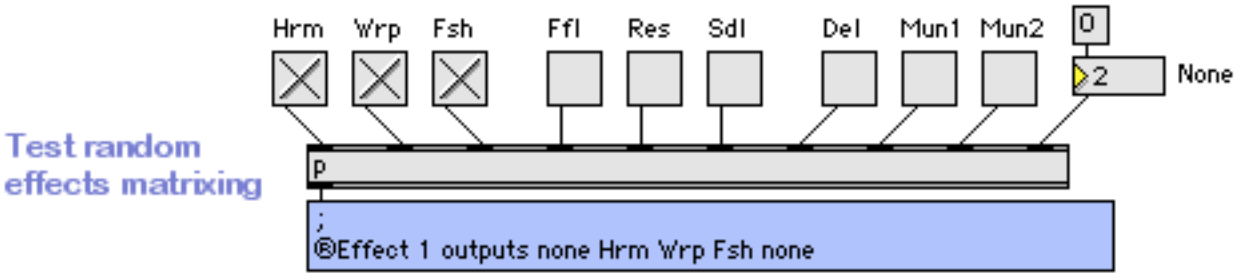
Calibrate the input sound

The detection and random treatments processes rely on the level of the input signal. You should adjust the level of the adc in order to have at least 2 orange leds displayed in the corresponding VU meter, when the performer plays fortissimo :

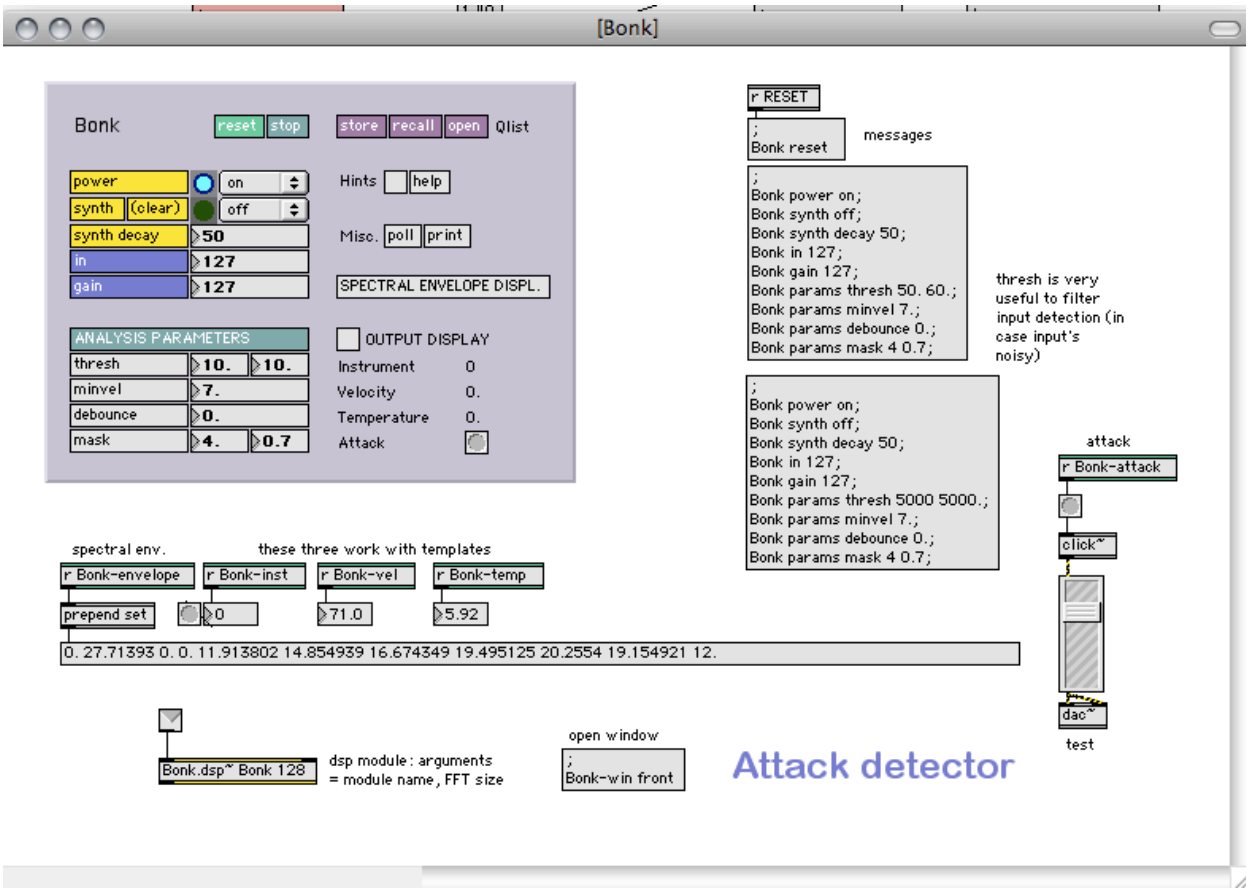


Test random effects

Back to our INIT patch, by enabling the effects, you allow the program to route the audio input to any of them, at "attack detection" time. You can also insert "none" selections, which means that the program might sometimes not route the signal to any effect.



Back to the DSP patch, click the "Bonk" button next to the Adc slider and open the attack detection subpatch :

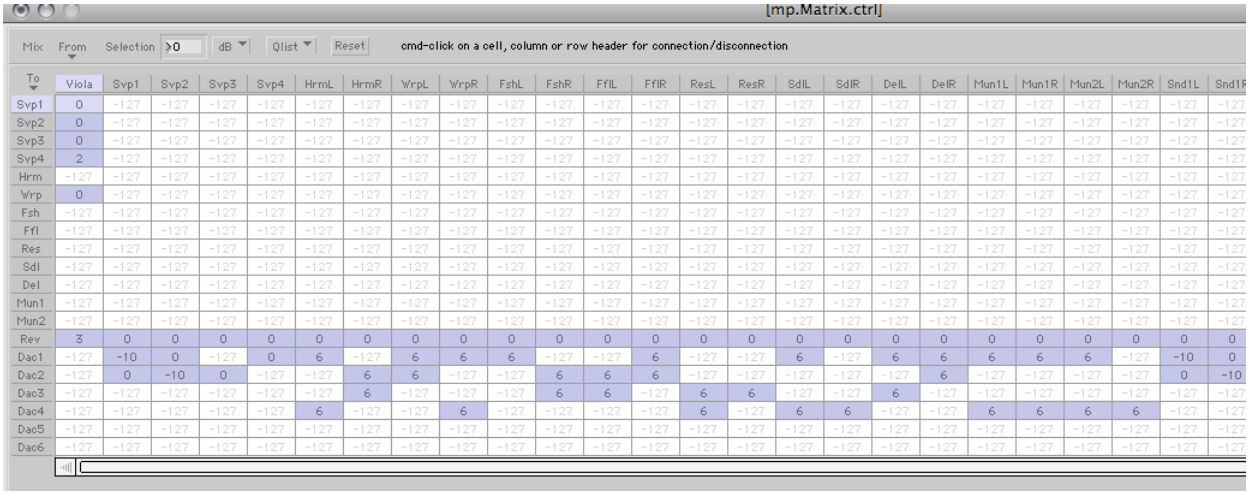


On the rightmost part, you see the detected attacks triggering a click sound. Adjust the level in order to monitor those attacks on front speakers (then set it back to 0 !).

Now, open the matrix interface :

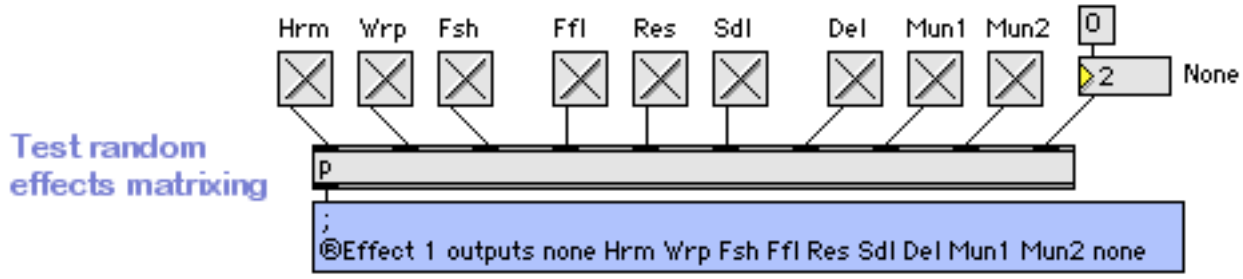
```
;
Mix window front
```

You might see something like this :

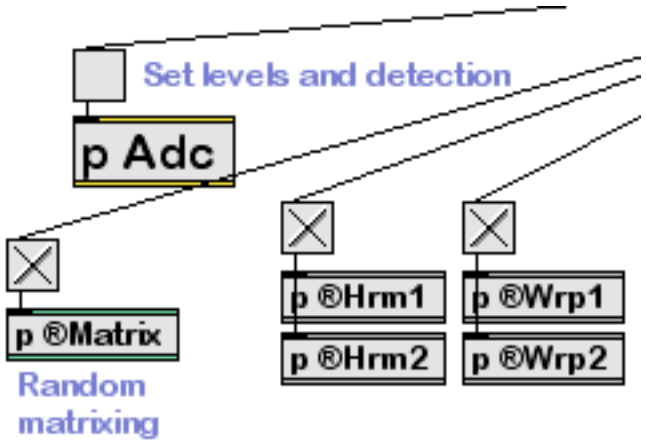


This can be analyzed like this : the Adc was (randomly) routed to the Wrp effect, is re-verberated, and that the effects are (randomly) routed to all dacs - which means that the player might have played with quite a lot of energy, as the sound is diffused over all loudspeakers.

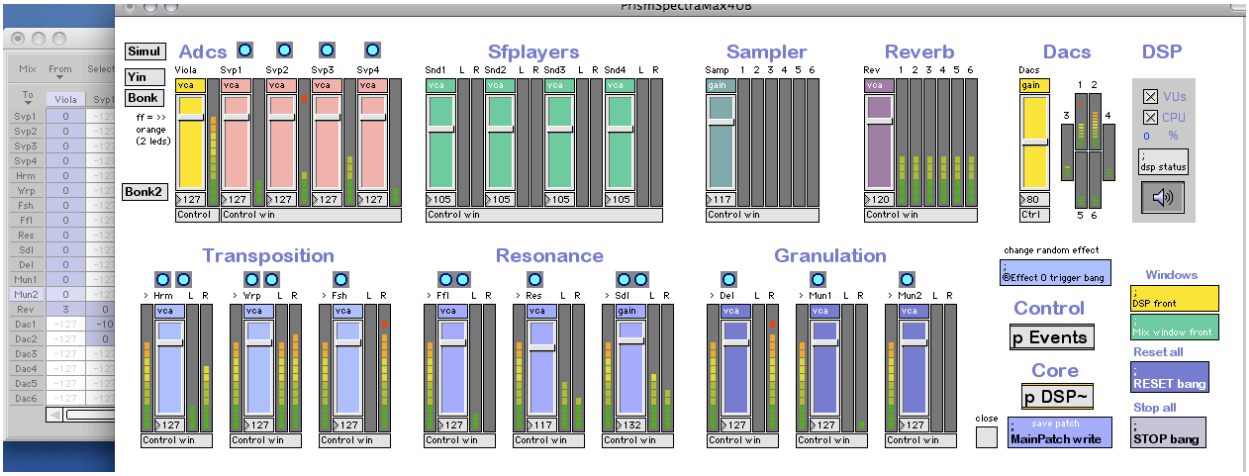
Selecting all effects here... :



... is a good way to perform the gain level for each effect. If you would need to adjust them more precisely, simply de-connect the Adc detection in the INIT patch :



And route the signal by hand in the matrix UI :



Feedback

All along the piece, it is rather difficult to predict which effect will be chosen at which time. You may encounter, in very extreme conditions (when using an aerian microphone for instance, which is not recommended) some feedback problems, especially when the player is NOT playing : as the attack detection changes the routing of the audio input to such or such effect, the system might not have the "time" to get into a feedback loop. But if the player stops playing in the middle of a section, you might get a feedback, a little bit like a guitar player letting the strings resonate in front of his amplifier. To avoid any "bad" feedback, the sound engineer should keep a finger on the adc level (rather than the dacs) during the performance, and raise it to 3/4 dBs when he "feels" that the sound is getting "too much" into a loop, then gently going back to the original level. According to the system, the feedback shouldn't remain more than one or two seconds, and thus, as it is hopefully nicely processed over effects, the feedback itself becomes a musical part of the concept. During the premiere, we had "nice" feedbacks controlled by the sound engineer - nice, because it participated to the dynamic/interactive overall aspects of the system.

Description of events

All events are briefly described internally within the corresponding subpatchers in the Events window. We can't describe them all here, but here are some global indications :

Section 1 : transpose the input sound, and process it through effects
Section 2 : trigger soundfiles
Section 3 : settings for attack-detected RAM sample triggers, trigger DFD soundfiles
Section 4 : trigger soundfiles
Section 5 : process the input sound through effects
Section 6 : like Section 1, plus also process the transposed sound