DIRECTCSOUND AND VMCI: THE PARADIGM OF INTERACTIVITY

by Gabriel Maldonado

1. INTRODUCTION

Csound was initially thought of as a deferred-time synthesis language as were all the other synthesis languages, like MUSIC V, available at that time. That view of synthesis languages remained essentially unchanged from the beginning of '60s to recent times. This paradigm still continues to satisfy many composers' needs. And although this approach still has several important advantages over its real-time successors, it is impossible to ignore that the absence or real-time synthesis impedes composers' direct and immediate relationship with their sonic material.

In its original version, Csound couldn't be used in live performances because the processing speeds of the standard computers of that time (the 1980's) were insufficient for real-time synthesis. Consequently, the original Csound language didn't include any live-control functions. During the early 90's, Barry Vercoe (the creator of Csound) added some MIDI oriented opcodes to use for real-time control. At that time, the only machines capable of running Csound in real-time were Silicon Graphics machines and expensive UNIX workstations.

Nowadays, PCs with Intel processors have become fast enough to run Csound in real-time. DirectCsound, a real-time version of Csound, has filled a lot of the gaps in the original CSound interactive functionality by adding opcodes that allow live control of parameters. In this version, many new features have been implemented, such as control of MIDI input/output (which enables Csound to connect with external world). Also, there has been a reduction of the latency problems which appeared unsolvable when the first real-time versions of Csound were released.

The new functionality allows total control over synthesis parameters, all of which can be defined by the user in a very flexible way. This flexibility, together with unlimited synthesis power, makes DirectCsound superior to any hardware MIDI synthesizer available now. Just a few years ago, owning a workstation with real-time synthesis processing power at home was unthinkable. Such features were available only to machines costing hundreds of thousands of dollars.

But DirectCsound is free, and it is sufficient to have run it on a cheap PC with audio card. This minimal set up enables users to compose music interactively, and also, use CSound for live performances. It is now possible to think of Csound as a universal musical instrument.

The newest features of Csound will be explained in the following sections through several examples.

In last section, we will demonstrate some functions of VMCI (Virtual Midi Control Interface, a program that emulates several types of MIDI controllers). VMCI has been designed in order to provide a software tool which enables users to control DirectCsound in real time. It can even be used with other hardware/software MIDI stuff.

2. SPECIFIC FEATURES OF DIRECTCSOUND

2.1 Inputs and Outputs

With DirectCsound, a musician can use both MIDI IN and MIDI OUT ports to send and receive data. This feature allows real-time control of any synthesis parameter.

Regarding audio outputs, the user can choose two different types of *drivers*:

1. old *MME* (*Multi Media Extensions*) driver, which was introduced with Windows 3.1, and
2. new *DirectX* driver (*DirectSound APIs*), which allows very low latency, practically achieving the total absence of any perceptible delay in real musical situations. This allows to us to trigger DirectCsound from a MIDI keyboard, and thus retain the interactivity of any hardware MIDI synthesizer.

DirectCsound automatically recognizes all installed MIDI ports, as well as *DirectX* and *MME* audio ports.

DirectCsound can activate both audio DAC output and file writing of samples, allowing hard-disk recording of a real-time performance.

As for the number of audio channels, at the present time, DAC output can only handle mono and stereo. Whereas the new opcode ***fout*** allows composers to write to files with any number of channels. Notice that the ***fout*** opcode is totally independent from the ***out*** opcode. So, it is possible to record a real-time session to a multi-track file with four, eight, twelve tracks, etc. (Obviously, this is possible only if the hard-disk is fast enough. Otherwise it is possible to use a RAM-disk by copying the recorded file to the hard-disk at the end Csound real-time session). Presently, only monophonic or stereophonic monitoring is possible during live performance, but there is always the potential to play the multi-channel pre-generated files afterwards, and record them to another real-time multi-track format (like a digital 8 track recorder).

At last, DirectCsound permits a console window with up to 2050 lines of text (standard DOS windows only allow 25 lines). This makes orchestra and score errors faster to find.

2.2 Orchestra opcodes

DirectCsound has several opcodes which haven't been implemented in the standard version of Csound as of yet. (Most of them have already been ported to canonical Csound, and others will be ported soon). Below is a list of the opcodes that I've implemented so far (only some have been ported to the canonical version). For further information about each opcode, see the manual.

MIDI controllers
midic7, midic14, midic21, ctrl7, ctrl14, ctrl21
initc7, initc14, initc21
slider8, slider16, slider32, slider64, slider8f, slider16f, slider32f, slider64f, s16b14, s32b14 - return a signal according to incoming MIDI control-change messages.

Micro-tuning together with MIDI
cpstmid - enables definition of micro-tone scales. This opcode is designed to be used with MIDI.

MIDI generators
noteon, noteoff, ondur, ondur2 - send note-on and note-off messages to the MIDI-out port.
moscil, midion - send streams of note-on and note-off messages to the MIDI-out port. All the MIDI-message generating process is controllable by input arguments.
outic, outkc, outic14, outkc14, outipb, outkpb, outiat, outkat, outipc, outkpc, outpiat, outkpat - send the corresponding MIDI CHANNEL messages.
mclock, mrtmsg - send the corresponding MIDI SYSTEM REAL-TIME messages

General MIDI in/out message handling
midiin, midiout, midion2, nrpn - handle byte-level MIDI messages
mdelay - MIDI delay

MIDI note duration extension
xtratim, release - allows the extension of duration in MIDI-activated notes. It extends the note-off time.

Subroutine call
call, calld, callm, callmd - allows an instrument to call an another instrument. Equivalent to subroutine calls available in other programming languages such as Basic or C.
parmck, parmtk, parmca, parmta, rtrnck, rtrntk, rtrnca, rtrnta - allows the passing of arguments to the called subroutines as well as receiving return arguments, as in structured programming languages.

Signal wrapping
wrap, mirror - wrap input signal in two different ways (see manual).

Interpolators
ntrpol - linearly interpolates between two signals.

***WaveGuide* algorithms**
wguide1, wguide2 - Plucked string and struck-plate physical models
flanger - a flanger which can be totally configured by the user.

Exponential segment envelope generator
expsega - similar to expseg, but more precise with audio signals

16-bit tables related opcodes (save half RAM space)
loscil2 - similar to loscil, but can handle 16-bit samples stored into a table (together with GEN22), halving computer RAM needs.
fof3 - similar to fof2 but it can handle 16-bit samples and gives a better audio quality because of linear interpolation
lposcint - oscillator that allows the variation of the starting point and ending point of a table during reading operations at *k-rate*. Optimized to work with 16-bit sample tables.

Signal visualization
printk2 - prints k-rate signal (only when they vary).

High-precision oscillators
poscil - similar to oscili but allows a very high frequency resolution.
lposcil - oscillator that allows to vary the starting point and ending point of a table during reading operations at k-rate.
lposcint - like the previous opcode, but for 16-bit tables.

FM oscillator
foscili2 - similar to foscili, but allows different function tables for carrier and modulator.

Filters
lowres - resonant low-pass filter.
lowresx, tonex, atonex, resonx - banks of serially-connected filters (to obtain a steeper curve).
vlowres - bank of variable resonant low pass filters, serially connected.
resony - bank of variable second-order band-pass filters, connected in parallel.

Fast power of two
powoftwo, logbtwo - power of two or logarithm base two (faster than standard *pow* opcode)

Information about sample-tables
ftlen2 - returns the length of a table generated by GEN01 or GEN22, using deferred allocation.
nsamp - returns the number of samples actually loaded into a table from a file.
ftsr - returns sampling frequency of an audio file loaded into a table.

New GEN functions
GEN22 - loads a file of samples into a table, similar to GEN01, but it stores samples using 16-bit integers instead of 32-bit floating-points, halving computer RAM need.
GEN23 - reads values from an external text file and stores it into a table.
GEN24 - reads values of another table and scales them according to minimum/maximum limits defined by the user.

Trigger
trigger - generates a boolean true value (1 = true), when a signals meets a threshold defined by the user. In any other case, trigger generates false (0 = false). Value 1 (true) can be used by other opcodes to start any kind of events.

Multi-track file writing
fout - write an arbitrary number of audio signals to a multi-track file.
foutk - same as above, at *k-rate*.
fouti - same as above, at *i-rate*.
foutir - same as above, but takes care of duration of the corresponding note.
fiopen - create a file and enable it for reading or writing.
vincr - accumulator, increment an audio variable of an arbitrary value.
clear - clear an audio variable (set it = 0).

Multi-track file reading
fin - read signals from a multi-track file at a-rate.
fink - some as above, at *k-rate*
fini - some as above, at *i-rate*

Artificial foldover
fold - generates artificial foldover on an input signal.

2.3 Score opcodes and operators

New score opcodes
{ } - Loops with the possibility of nesting
F - score tables

Nested macros

New macro syntax that allows nesting.

Nesting means having more than one loop, one inside the other. For example, if we have four composed blocks called A,B,C,D and we want to repeat 4 times the first block, 3 times the second, 2 times the third and no repetitions for the fourth; in this case we will have a sequence of the following type:

```
A-A-A-A-B-B-B-C-C-D.
```

The first four repetitions of block A can be generated by a loop of 4 iterations; likewise the 3 repetitions of B and the 2 repetitions of C. But, suppose to wish to repeat the entire construction for four times, we will have the following sequence:

```
A-A-A-A-B-B-B-C-C-D-A-A-A-A-B-B-B-C-C-D-A-A-A-A-B-B-B-C-C-D-A-A-A-A-B-B-B-C-C-D.
```

This sequence can be easily generated by an external loop with four iterations that nests the three internal loops. If using nested loop for a verbatim repetition seems impractical because of the initial difficulty of learning this technique, remember that one can find many compositional situations in which the parameters of each iteration should be varied according to user-defined algorithms. In these cases, parameters must be different for each iteration. So, calculating each parameter by hand could be far more complex and boring than the previously described technique. It is more convenient to use nested loops because they create a more synthetic and readable score writing.

New macro operators
T - returns an element contained by a score table, by giving the corresponding index.
R - returns a random number
^ - power operator
% - remainder operator (modulus)

3. USING DIRECTCSOUND IN REAL-TIME

In this section some MIDI-controlled orchestra examples are analyzed.

In order to use DirectCsound in real-time, it is necessary to start it with the appropriate flags. Thes are not present in the canonical version at the present time.

DirectCsound is capable of handling both incoming and outgoing MIDI data. With regards audio, there is the option of using old *MME* drivers or new low-latency *DirectX* drivers. However, conflicting flags can't be used at the same time. The audio buffer is unique in the *DirectX* driver while it is possible to specify a number of buffers when using old MME driver. Notice that non-standard flags have characters '-+' as prefix, whereas the standard ones only have '-'.

User must use the following flags in order to run the examples:

-b <number> set buffer length (standard flag)
-+p < number > set buffer number of MME driver (don't use with -+X)
-+X activate DirectX (don't use with -+q)
-+q activate MME (don't use with -+X)
-+K activate MIDI IN port
-+Q activate MIDI OUT port
-+j < number > set the maximum number of lines of text visualized in DirectCsound console (useful for debug)

When more than one MIDI port is available, the DirectCsound console window will show a list of all the port names, and ask the user to select the required port.

In the following examples, a basic knowledge of MIDI protocol is taken for granted.

3.1 A simple example: sine.orc

This is the simplest example. It consists of an oscillator whose frequency is controlled by a MIDI note number sent by a MIDI keyboard.

```
; sine.orc
        sr      =   44100
        kr      =   441
        ksmps   =   100
        nchnls  =   1
gi1     ftgen       1, 0, 1024, 10, 1

        instr       1
ifreq   cpsmidi
iamp    ampmidi     10000
```

```
a1      oscili   iamp,  ifreq, 1
        out      a1
        endin
```

This is the score:

```
; sine.sco
f0 3600
```

This example must be activated with the following command line if the user has already installed DirectX on his computer:

csound.exe -+X -+K -b200 sine.orc sine.sco

...otherwise he must use old MME driver with the following command line:

csound.exe -+q -b500 -+p8 -+K sine.orc sine.sco

DirectCsound will prompt the user to enter the audio out port number, as well as, the MIDI out port number in the console window.

After the user has typed in that data, it is possible to play notes on a master-keyboard (connected via a MIDI-IN port) and to listen to sinusoids with frequency that corresponds to the pressed key. The timbre will not be interesting, but this orchestra is useful to verify if the system is set up correctly. If sound interruptions are present, it is necessary to increase buffer length by changing the number following flag *-b*.

In this case, the opcodes handling MIDI are standard, i.e. *cpsmidi*, and return the frequency of the pressed key. *ampmidi*, returns the amplitude.

Notice that score only contains the *f0 3600* statement. That allows Csound to compile in real-time for 3600 seconds. Actually, the audio table containing the sinusoid is generated directly inside the orchestra, by *ftgen* opcode. It is possible to terminate the Csound session at any moment by pressing CONTROL-C (after being sure that the console window has got the *focus*).

Adding an amplitude envelope to sine.orc

Let's modify instrument 1 of the previous orchestra as follows:

```
        instr    1
ifreq   cpsmidi
```

```
iamp    ampmidi    10000
kenv    linsegr    0, .1, 1, .3, .5, .2, 0
a1      oscili     iamp, ifreq, 1
        out        1*kenv
        endin
```

Notice, the *linsegr* opcode is used. This opcode handles the release stage when a current note receives a MIDI note-off message. The resulting action will extend normal note duration for the time the user has assigned to the corresponding parameter (that is second to last argument of line containing *linsegr*. The last argument contains the level value that *linsegr* returns at the end of release).

linsegr automatically extends note duration, and release stage consists of only one (exponential) segment. Notice that linsegr must be used only with MIDI. Otherwise it doesn't produce any extension of note duration.

What if a complex envelope is needed in the release phase consisting of more segments? (For example, a crescendo followed by a diminuendo.)

The next example will show how to implement it by means of two new opcodes: *xtratim* and *release*.

3.3 Extending the life of a MIDI-activated note: *xtratim* and *release*

In this example we suppose that the user wants a more complex envelope in the release phase of a MIDI-activated note.

Now, it is necessary to make a distinction between two types of orchestra instruments: those which are activated by the score and those which are activated by a MIDI note-on message. The reason we need this distinction is that sometimes some opcodes operate correctly only on instruments designed to be activated by MIDI. Most opcodes, however, can operate on both score-activated instruments and MIDI-activated instruments.

Actually, extending a note in a score-activated instrument is very easy. Duration is defined by the *p3* parameter of the i-statement of the score. To extend it, it is sufficient to assign a starting value to *p3* plus a value referring the wanted extension of duration, expressed in seconds:

```
p3      =    p3+1     ;adds 1 second to the duration of current note
```

This is true for score-activated instruments.

In MIDI-activated instruments, *p3* is meaningless because corresponding notes remain active until a note-off message is received.

The only ways to extend the duration of these notes is by

• using MIDI-oriented envelope opcodes (those terminating with an 'r', *linenr*, *linsegr*, *expsegr*, the 'r' being for 'release'), or
• by using two opcodes, specially designed to accomplish this task: *xtratim* and *release*.

Look at the following example:

```
        instr   1
inum    notnum
icps    cpsmidi
iamp    ampmidi   4000
;############## complex MIDI envelope #################
        xtratim 1                 ; extra-time, i.e. release duration
krel    init    0
krel    release                   ; outputs release-stage flag (0 or 1 values)
if      (krel > .5) kgoto  rel ; if in release-stage goto release section
;************ attack and sustain section ***********
kmp1    linseg  0, .03, 1, .05, 1, .07, 0, .08, .5, 1, .2, 50, .2
kmp     =       kmp1*iamp
        kgoto   done
;************ release section **********************
rel:
kmp2    linseg  1, .05, 4, .7, 0
kmp     =       kmp1*kmp2*iamp
done:
;################################################

a1      oscili  kmp, icps, 1
        out     a1
        endin
```

This instrument appears complex, but it isn't. The purpose of the *xtratim* opcode is to add extra time to the duration of instr 1. The extra-time amount is defined by the input argument (one second in the example). Notice that *xtratim* has no output. The purpose of the *release* opcode is to indicate when the corresponding note is in the normal stage and when it in at the extra time stage (the release time). In this case, there are two envelopes, performed in succession. Notice one important detail. In most cases, the

sustain envelope isn't performed in its entirety because it is impossible to foresee how much time the performer leaves the key pressed. So, the last value of variable *kmp1* could not be equal to the initial value of *kmp2*. In order to avoid discontinuities in the sound («tics»), it is necessary to multiply last value of *kmp1* by *kmp2*. (*kmp2* must set to value 1 at the beginning, in order to leave initial output argument of *linseg* unmodified.) The release-stage envelope acts as a multiplier for the last value from the envelope of the previous stage. In the example, initially we have an ascending release which after a second after descends toward zero.

3.4 Continuous controllers: varying amplitude and frequency of vibrato while playing notes.

```
;vibrato.orc
            instr       1
ifreq       cpsmidi
iamp        ampmidi     10000
kfrqvib     midic7      1, 0, 1
kampvib     midic7      2, 0, 1
kvib        oscili      kampvib, kfrqvib, 1
kenv        linsegr     0, .1, 1, .3, .5, .2, 0
a1          oscili      iamp, ifreq*powoftwo(kvib), 1
            out         a1*kenv
            endin
```

This example allows one to play a note on the master-keyboard, and varying both the frequency and the amplitude of vibrato simultaneously. Vibrato is generated by an oscillator (performing at k-rate),that generates a control signal which is multiplied by the base frequency of the audio signal oscillator. The frequency and amplitude of the oscillator generating vibrato can be varied continuously by means of MIDI control-change messages. The opcode used to handle these messages is *midic7* which allows scaling of the 7-bit raw values (incoming by the MIDI in port with a 0-127 range) into a minimum-maximum range defined by the user.

There are three *midic7* input arguments plus an optional one: MIDI control message number (for example 1 is corresponding to modulation-wheel, 2 to breath-control etc.), minimum and maximum values, used to re-scale the output.

Notice the use of *powoftwo( )* function, which gives a multiplier varying exponentially instead than linearly: if we used a direct sum of vibrato signal and the note base frequency, the vibrato would generate large frequency variations in the low frequency range and little variations in high frequency range. In this case, multiplying by an

exponent enables us to express the variation in octaves instead of cps. Although the mathematical reasoning behind this concept isn't too complex, it is beyond the scope of this chapter. Assuming that the range of the second instance of *midic7* opcode (which controls vibrato amplitude) is expressed in octaves, variations in the range of 0 to 1 octaves are sufficient.

3.5 More complex vibrato, delay and tremolo, controllable in real-time

The previous example is not optimized for speed. If the performer plays more than one note at the same time, several instances of *midic7* opcode are activated that generate an identical signal. This is a useless waste of processing time.

It is necessary to think a little about how Csound instruments and opcodes work.

A Csound instrument is a template used by Csound when it activates the corresponding note.

When this happens, an '*instance*' of that instrument, together with its data, is generated. Csound instruments are polyphonic. So, more than one note of the same instrument can be activated with different parameters at the same time. Each of these notes is actually an *instance* of the corresponding instrument.

In a similar way, multiple instances of the same opcode can be called by the same instrument (in the previous example this happens with *midic7* opcode). So, we have a hierarchy of instances. Several notes of the same instruments generate several instances of that instrument, each containing more instances of the same opcode. Csound variables represent signals which can be control signals, audio signals or initialization parameters (the last choice remains constant for all note duration). It is important to notice that the variables contained inside an instrument are local. This means that variables with the same name, placed in different instruments, are different in all respects. Furthermore, even considering the same instrument, variables with the same name are different in each new instance of that instrument, and they contain different but independent values for each parameter that correspond to activating notes.

Let's return to the previous example. When the performer activates a chord consisting of 3 notes, the two instances of *midic7* contained by instr 1 are actually multiplied by three, creating a total of 6 instances, which are useless because they all produce the same values. This is a waste of processing time. It is better to use the MIDI-control opcodes in a separate instrument which is activated only once, and remains active for the duration of current Csound session. These instruments are activated by the score instead of by a MIDI controller.

In order to make signal produced by such an instrument visible to other devices (and in particular by the MIDI activated instrument which needs to access the values produced by controllers), it is necessary to use *global variables*. Global variables have

orchestra scope, not instrument scope. This means that they are common and visible by all instruments of the orchestra. Furthermore, unlike local variables, they are common to all instances of all currently active instruments.

Let's look at the example below:

```
;orchestra

        sr      =   44100
        kr      =   441
        ksmps   =   100
        nchnls  =   1

gi1 ftgen 1, 0, 1024, 10, 1, .2, 0, 0, .1, 0, 0, .05       ; audio table
gi1 ftgen 2, 0, 129, 7, 0, 4, -1, 64, -1, 4, 0, 56, 0          ; tremolo table

;## vibrato functions
gi1 ftgen    50, 0, 513, 10, 1 ;sine
gi1 ftgen    51, 0, 513, 7,  1, 511, -1 ; falling saw tooth
gi1 ftgen    52, 0, 513, 7,  -1, 511, 1 ; rising saw tooth
gi1 ftgen    53, 0, 513, 7,  0, 128, 1, 256, -1, 128, 0 ;triangle
gi1 ftgen    54, 0, 513, 7,  1, 256, -1, 255, 0 ;square
gi1 ftgen    55, 0, 513, 7,  0, 170, 0, 0, 1, 170, 1,  0, -1, 170, -1, 0, 0 ;three steps
gi1 ftgen    56, 0, 513, 7,  0, 128, 0, 0, 1, 128, 1, 0, 0, 128, 0, 0, -1, 128, -1, 0, 0 ;4 steps
gi1 ftgen    57, 0, 513, 7,  1, 128, 1, 0, 0, 128, 0, 0, -1, 128, -1, 0, 0, 128, 0 ;4 steps 2
gi1 ftgen    58, 0, 513, 7,  -1, 128, -1, 0, 0, 128, 0, 0, 1, 128, 1, 0, 0, 128, 0 ;4 steps 3
; fourth, fifth and octave
gi1 ftgen    59, 0, 513, 7, 0, 128, 0, 0, 5, 128, 5, 0, 7, 128, 7, 0, 12, 128, 12
; octave, fourth, fifth and unison
gi1 ftgen    60, 0, 513, 7, 12, 128, 12, 0, 5, 128, 5, 0, 7, 128, 7, 0, 0, 128, 0
;4 glissando
gi1 ftgen    61, 0, 513, 7,  -1, 90, -1, 38, 0, 90, 0, 38, 1, 90, 1, 38, 0, 90, 0, 38, -1

gi1 ftgen 100, 0, 8193,  5,  .001, 8193, 1     ; exponential curve for slider mapping

gk1     init        0
gk2     init        0
gk3     init        0
gk4     init        0
gk5     init        0
```

```
gk6     init        0
gk7     init        0
gk8     init        0
gaout   init        0

;////////////////////////////
        instr       1
;////////////////////////////
kvib    oscili      gk1, gk2, i(gk3)+.5
atrem   oscili      gk4, gk5, 2
ifreq   cpsmidi
iamp    ampmidi     10000
aenv    linsegr     0, .05, 1, .2, .2, 10, .2, .2, 0
a1      oscili      aenv*iamp*(1+atrem), ifreq*powoftwo(kvib),1
        vincr       gaout, a1
        endin

;////////////////////////////
        instr       100
;////////////////////////////
        initc7      1, 3, (50-50)/(61.5-50)
gk1     ctrl7       1, 1, 0, 1              ; vibrato amplitude
gk2     ctrl7       1, 2, .5, 20, 100       ; vibrato frequency
gk3     ctrl7       1, 3, 50, 61.5          ; vibrato table
gk4     ctrl7       1, 4, 0, 1              ; tremolo depth
gk5     ctrl7       1, 5, 2, 20, 100        ; tremolo frequency
gk6     ctrl7       1, 6, 0, 1              ; wet/dry ratio
gk7     ctrl7       1, 7, 0, 1              ; delay feedback
gk8     ctrl7       1, 8, 1, 1000           ; delay time
gk8     tonek       gk8,2
a8      interp      gk8
a1      init        0
a1      vdelay      gaout+a1*gk7,a8,1000
        out         a1*gk6+gaout*(1-gk6)
        clear       gaout
        endin

;score
i100 0 3600
```

As you can see, the score doesn't need the *f0* statement because the duration of the real-time session is already set to 3600 seconds by the only note activating ëglobal' instr 100. We will see the purpose of instr 100 later.

Instrument 1 has three oscillators. The first and the second oscillator generate vibrato and tremolo, whereas the third one generates the audio signal. Some remarks about instr 1 follow.

1. Vibrato amplitude is defined externally from instr 1, as well as, vibrato frequency, vibrato table number, tremolo depth and tremolo frequency. These parameters are supplied to the oscillators by means of the global variables *gk1, gk2, gk3, gk4* and *gk5*. We will see the storage locations of these variable later.
2. *out* opcode is not present inside instr 1, because its output is assigned to the *gaout* global variable. The reason we use a global variable and do not send the output signal directly to the *out* opcode is that the signal is reused by another instrument that adds an effect (a delay) and balances the wet/dry ratio before it is assigned it to the *out* opcode. When sending an audio signal to the *gaout* global variable, simply assigning it is not sufficient because a global variable is common to all orchestra instrument instances. So, if instr 1 is polyphonic and its output is assigned directly to *gaout*, the signal generated by any concurrent instances of instr 1 would be replaced by the one immediate instance. In that case, when *gaout* is connected to the *out* opcode (in instr 100), the signal would be monophonic (not polyphonic) because it would contain only the output of last instance of instr 1 (last voice). In order to avoid this drawback, we have to mix output of instr 1 with the content of *gaout* instead of simply assigning the raw value directly. Here this operation is accomplished with the *vincr* opcode.
   vincr does this:
   ```
   gaout = gaout + a1
   ```
   ...but is faster. It is an accumulator, especially designed to sum a signal and a mixing line. After connecting *gaout* to the *out* opcode (placed in instr 100), it is necessary to clear *gaout* by setting it to zero. Otherwise, the variable will 'explode' because values would continue accumulating endlessly.
   To set *gaout* to zero, the following line would be sufficient:
   ```
   gaout = 0
   ```
   ...but the *clear* opcode is used instead. It is designed specifically to complete this task quickly (in this example the increasing in speed is not particularly evident, but it is apparent, in complex mixing lines when several variables have to be zeroed at the same time. See manual).

Instrument 100 is called by the score only once per Csound session. This instrument has three functions:

1. MIDI control messages handling, by filling global variables. Such variables have the following purposes:
   gk1 - vibrato amplitude
   gk2 - vibrato frequency
   gk3 - select vibrato table number
   gk4 - tremolo depth
   gk5 - tremolo frequency
   gk6 - wet/dry delay ratio
   gk7 - delay feedback (to allow echo)
   gk8 - delay time

2. generation of a delay effect (that is unique for all notes of all instruments)

3. master output control.

Let's examine this instrument more closely. Noticed that there is no *midic7* instance in instr 1. In fact, the MIDI control-change messages are controlled by several instances of *ctrl7*, an opcode similar to *midic7* but with the flexibility to choose the MIDI channel. Consequently, it can be used in score-activated instruments. Assigning *midic7* to a score-activated instrument causes CSound to crash. (The *midic7* channel is implicitly implied and has the same MIDI channel number as the instrument that activates it.)

Notice that global variables are initialized before any instruments are declared in the header section of orchestra. If global variables are not declared first errors appear caused by using variables before they are initialized.

The purpose of *initc7* opcode is to initialize the first output value of the corresponding MIDI control-change handling opcode (works both with *midic7* and *ctrl7*, see manual).

3.6 Non-linear distortion, micro-tuning and slider banks

Let's analyze the following orc/sco pair:

```
;**** distortion.orc
        sr      =   44100
        kr      =   441
        ksmps   =   100
        nchnls  =   2

ga1     init        0
gk12e4  init        882
```

```
;////////////////////////////////////////////////////
        instr       1
;////////////////////////////////////////////////////
kbend   pchbend     -1,1
i1      cpstmidi(gk12e4)
k2      linsegr     0, 1.5, 1, 6, .15, 1, .15, 1.2, 0
a1      oscili      k2, i1*powoftwo(kbend), 1
        vincr       ga1, a1
        endin

;////////////////////////////////////////////////////
        instr       2
;////////////////////////////////////////////////////
gk12e1,gk12e2,gk12e3,gk12e4,gk12e5,gk12e6,gk12e7,gk12e8   slider8f 1,\
\;      ctl     min     max     init    func    icutoff
        1,      .1,     5,      .1,     0,      5,      \;1- distortion
        2,      2,      500,    20,     92,     5,      \;2- filter cutoff
        3,      .05,    8,      1,      92,     5,      \;3- filter resonance
        4,      880,    888.5,882,      0,      10000,\;4- micro-tuning table
        5,      0,      8,      0,      0,      5,      \;5- amp. of dist.modulation
        6,      .1,     10,     1,      92,     5,      \;6- freq. of dist.modul.
        7,      0,      200,    0,      0,      5,      \;7- amp. of filter modulation
        8,      .1,     10,     1,      92,     5       ;8- freq. of filter modulation
gk12e9 ctrl7    1, 9,   870,    878.5                   ;9- distortion table

kosc    oscili      gk12e5,gk12e6,879
kosc    tonek       gk12e1+kosc,5
aosc    interp      kosc

amod    tableikt    ga1*aosc, int(gk12e9),1, .5, 0

kfilt   oscili      gk12e7, gk12e8, 879
kfilt   tonek       gk12e2+kfilt, 5
kres    tonek       gk12e3, 1.5

amod    lowres      amod, kfilt, kres
amod    =           amod*6400
adel    delay       amod,.3
        outs        amod, adel
```

```
        clear       ga1
        endin
;////////////////////////////////////////////////
;***** distortion.sco

;## audio table ##
f1 0 1024 10 1 ;sinusoid

;## exponential slider mapping ##
f92  0  4097  5  .01 4096 100

;## distortion tables ##
f870 0 4097 9 .5 1 90 ;sigmoid
f871 0 4097 8 8 2032 1 32 -1 2032 -8 ; cubic spline 1
f872 0 4097 8 0.3 1024 1 2048 -1 1024 -0.3 ; cubic spline 2
f873 0 4097 8 0 256  1 3584 -1 256  0 ; cubic spline 3
f874 0 4097 8 -1 512 1 512 -1 2048 -1 512 1 512 -1
f875 0 4097 8 -1 512 1 512 -1 2048 1 512 -1 512 1
f876 0 4097 8 -1 512 6 512 -4 2048 1 512 -1 512 1
f877 0 4097 8 7 128 8 128 6 128 7 128 5 128 6 128 4 128 5 128 3
        128 4 128 2 128 3 128 1 128 2 128 0 128 1 128 0
        128 -1 128 0 128 -2 128 -1 128 -3 128 -2 128 -4 128 -3
        128 -5 128 -4 128 -6 128 -5 128 -7 128 -6 128 -8 128 -7
f878 0 4097 8 12 128 8 128 6 128 7 128 5 128 6 128 4 128 5 128 3
        128 4 128 2 128 11 128 1 128 2 128 0 128 6 128 0
        128 -6 128 0 128 -2 128 -1 128 -3 128 -2 128 -4 128 -3
        128 -5 128 -4 128 -6 128 -5 128 -7 128 -13 128 -8 128 -12

;## positive sinusoid ##
f879 0 1024 19 1 1 0 1

;## tuning tables ##
;# equal temperament  #
;       numgrades       freqbase            scaleRatio (eq.temp.)
;               interval        basekeymidi
f880 0 16 -2    12      2       261       60
            1                       1.059463094359      1.122462048309      1.189207115003
            1.259921049895      1.33483985417       1.414213562373      1.498307076877
            1.587401051968      1.681792830507      1.781797436281      1.887748625363
```

```
;# diatonic pure #
f881 0 16 -2  12    2    261.62   60  1  1.04166667  1.125  1.171875  1.25  1.3333  1.40625  1.5  1.5625  1.66666
1.777777777777  1.875

# harmonic progression  1( 36 steps )#
f882 0 64 -2  36   2   30.5    24   1  2  3  4   5  6  7   8   9   10 11  12 13 14  15 16  17 18 19  20  21   22 23  24
25 26  27 28  29 30 31  32  33   34 35  36

;# harmonic progression 2( 24 steps)#
f883 0 32 -2  24   2   30.5   24   2  3  4  5  6  7  8  9  10  11 12  13 14 15  16 17  18 19 20  21  22  23 24  25

;# harmonic progression 3( 24 steps)#
f884 0 32 -2  24   2   15.25   24   4  5  6  7  8  9  10  11  12  13 14  15 16 17  18 19  20 21 22  23  24  25 26  27

;# harmonic progression 4( 12 steps)#
f885 0 16 -2  12   2   61   60   4  5  6  7  8  9  10  11  12  13 14  15

;# harmonic progression 5( 12 steps)#
f886 0 16 -2  12   2   61   60   8  9  10 11  12 13 14  15  16  17 18  19

;# harmonic progression 6( 24 steps)#
f887 0 16 -2  12   2   61   60   16 17  18 19  20 21 22  23  24  25 26  27 28 29  30 31  32 33 34  35  36  37 38  39

;# 'detuned' scale #
f888 0 16 -2  12   2   261.62   60   1  1.111111111 1.142857143 1.25 1.285714286 1.388888889 1.428571429 1.5
1.607142857 1.666666667 1.714285714 1.928571429

;## note ##
i2 0 3600
e
```

This is an example of non-linear distortion with a shaping-table unique for all voices of polyphony. This produces an electric-guitar-like, chord effect.

Let's start with instr 1.

pchbend opcode controls MIDI pitch-bend messages. In this example, the output values of this opcode cover a range of -1 to 1. This corresponds to pitch shifts down or up an octave as defined by the *powoftwo( )* function.

cpstmid opcode controls table parameters containing user-defined tuning systems. *cpstmid* is similar to *cpsmidi*, because both are designed for use with MIDI-activated instruments and require a table number as the input argument. The table is filled with parameters and a set of frequency ratios (for more information see the manual). This example shows different tuning systems can be selected by means of MIDI control-change messages during the performance. In fact the *gk12e4* global variable is sampled and held in the initialization stage *(i( )* function). So, its value is available as input argument to *cpstmid*, and can change for each notes. (It remains constant for the entire duration of one single note.) Notice that the *gk12e4* variable is initialized in the header section of orchestra. As mentioned earlier, an error appears if a variable is used before it is defined.

linsegr opcode defines the amplitude envelope of a sinusoidal oscillator, and the output of such oscillator is then distorted by instr 2. Oscillator output increments global audio variable *ga1*, which is reused by instr 2. *ga1* had been defined in the header section - before it is called.

Instrument 2 starts with the *slider8f* opcode, that generates a bank of 8 sliders, giving the possibility to set MIDI channel (a unique channel for all controllers), MIDI control number, minimum and maximum values, starting value, an optional mapping table number (if not used, set the corresponding argument to zero), and cutoff frequency of a low-pass filter placed before the output, in order to smooth discontinuities due to low resolution of 7-bit MIDI data. Such parameters must be defined for each controller, for this reason the argument of slider8f opcode are split into eight text lines by using '\' character, interpreted by Csound as line continuation. Notice that after the "\" character it is possible to put comments starting with "";" character.

As controllers needed in this orchestra are nine, and *slider8f* opcode can handle only eight of those, another line containing *ctrl7* opcode is added.

Variables interpreting the MIDI messages are:

gk12e1 - amount of distortion (corresponding to the amplitude of the signal generated by the oscillator)
gk12e2 - cutoff frequency of the low pass resonant filter, used to make distorted signals more musically, resulting very harsh to hearing otherwise
gk12e3 - filter resonance
gk12e4 - select the number of micro-tuning table (read by instr 1)
gk12e5 - amplitude of distortion-amount-modulating oscillator

gk12e6 - frequency of distortion-amount-modulating oscillator
gk12e7 - amplitude of cutoff-frequency-modulating oscillator, connected to resonant low pass filter
gk12e8 - frequency of cutoff-frequency-modulating oscillator, connected to resonant low pass filter
gk12e9 - select the number of table containing the wave-shaping function

The next oscillator is used as an amplitude-modulator for the *ga1* signal (generated by instr 1). This is equivalent to modulating the amount of distortion. The oscillator output is summed with the output of the slider controlling the level of distortion (*gk12e1*). The resulting value is then filtered to smooth the stepped shape of the output caused by the low resolution of MIDI. This filtering stage prevents 'tics'.

Filter output (*k-rate* signal) is then converted to a-rate (*aosc* variable) with the *interp* opcode. It linearly interpolates between current *kosc* value and the previous one, again, smoothing the signal.

Then, we can see *tableikt* opcode, which is similar to *tablei* (linearly-interpolated table reading), but changes to the table number at *k-rate*. This opcode distorts the input signal (which is *ga1* in this example) in accordance with the corresponding waveshaping table. In this example, this table can be changed during the performance of a note which allow us to hear how the sound output changes according to the type of functions used by current waveshaping table. Notice that the level of distortion modulates with the *aosc* signal as it is multiplied by the *ga1* signal.

Next oscillator modulates cutoff frequency of the lowpass resonant filter (wah-wah effect). *kfilt* signal, generated by such oscillator, is then summed to an offset (*gk12e2* signal, controlled by a slider), and filtered by a low pass filter in order to avoid tics. The signal that controls the amount of resonance (*gk12e3*) is filtered, too.

Then, the output of the waveshaping function (*amod* variable) is sent to the resonant low pass filter (*lowres* opcode) which is controlled in frequency and in resonance amount by *kfilt* and ***kres***. At last, the signal going to right channel is delayed of 0.3 seconds, in order to enrich output sound.

clear opcode zeroes *ga1* variable in order to avoid new values to continue being summed endlessly.

3.7 Granular synthesis

In this section a granular synthesis example will be presented, which allows real-time control of almost every parameters.

```
; granular.orc
sr      =   32000
kr      =   320
ksmps   =   100
nchnls  =   2

garev1  init 0
gaout1  init 0
gaout2  init 0

;- - - - - - -

;/////////////////////////////////////////
        instr   100
;/////////////////////////////////////////

gk_1, gk_2, gk_3, gk_4, gk_5, gk_6, gk_7, gk_8,\
gk_9, gk_10, gk_11, gk_12, gk_13, gk_14, gk_15, gk_16   slider161,\
\;      ctl     min     max     init    func
        1,      1,      20.5,   1,      0,      \;1 audio table number
        2,      0,      13.5,   1,      0,      \;2 play speed
        3,      .5,     5,      3,      0,      \;3 grain repeat speed
        4,      0,      2,      1,      0,      \;4 phase scanning speed of current sample
        5,      0,      1,      0,      0,      \;5 end frequency of glissandos
        6,      .005,   4,      .5,     100,    \;6 total duration of grains
        7,      1/15,   1,      .3,     0,      \;7 attack time of grains
        8,      1/15,   1,      .3,     0,      \;8 decay time of grains
        9,      0,      .12,    0,      0,              \;9 reverb send level
        10,     300,    8000,   3200,   0,      \;10 reverb low pass filter
        11,     0,      2,      0,      0,      \;11 random amount of grain frequency
        12,     0,      3,      0,      0,      \;12 random amount of grain duration
        13,     0,      .2,     .05,    0,      \;13 left/right channel phase difference
        14,     0,      .5,     0,      0,      \;14 random amount of starting grain phase
        15,     0,      3,      0,      0,      \;15 octaviation factor
        16,     .05,    2,      1,      0       ;16 global volume

gk_17, gk_18, gk_19, gk_20, gk_21, gk_22, gk_23, gk_24 slider8  1,\
\;      ctl     min     max     init    func
        17,     102,    105.5,  102,    0,\;17 harmonic tuning table number
```

```
        18,     0,      8.5,    0,      0,      \;18 amount of random changes in harmonic tuning table
        19,     1,      1.5,    1,      0,      \;19 left/right deviation of grain repetition speed
        20,     0,      1,      0,      0,      \;20 stating phase offset of grains
        21,     0,      15,     0,      0,      \;21 random amount of grain starting time
        22,     0,      1,      0,      0,      \
        23,     0,      1,      0,      0,      \
        24,     0,      1,      0,      0

;************ REVERB and OUTPUT *************
arevb   tonex   garev1, gk_10,3
arev    reverb2 arevb, 9, .05
        outs    gaout1 + arev, gaout2 + arev
        clear   gaout1, gaout2, garev1
        endin

;/////////////////////////////////////////////
        instr   1
;/////////////////////////////////////////////
ifmidi  cpsmidi
iamp    ampmidi    1

ifna        =   int(i(gk_1))        ;** left channel sample table
ifna2       =   ifna                ;** right channel sample table
imemlen     =   ftlen(ifna)         ;** length of allocated table
ilen        =   nsamp(ifna          ;** number of samples contained into the table
ilnDmem     =   ilen/imemlen        ;** sampling length/table length ratio
ifsr        =   ftsr(ifna)/sr       ;** sampling frequency of audio sample up sr
isrDmem     =   sr/imemlen          ;** necessary to calculate actual frequency
isrDdur     =   ftsr(ifna)/ilen      ;** sampling frequency of audio sample up number of samples

krdharm         linrand  gk_18      ;** random harmonic variation amount
krdharm2        linrand  gk_18
krdharm         table    krdharm, i(gk_17)
krdharm2        table    krdharm2, i(gk_17)
kform           table    gk_2, 151
kform       =   (ifmidi/127*.98) *kform*isrDmem*ifsr
krndpch     trirand  gk_11      ;**  random pitch variation of grains
krndpch2        trirand  gk_11
kform1      =   kform*powoftwo(krndpch)*krdharm   ;** right chan. freq.
```

```
kform2          =   kform*powoftwo(krndpch2)*krdharm2 ;** left chan. freq.
krnd                trirand  gk_21
kfund           =   2^(gk_3+krnd)       ;** speed of grain repeats

kphsrate        =   gk_4*isrDdur

krndphs         linrand  gk_14      ;** random phase amount

kphs        init     0
kphs        phasor   kphsrate
kphs            =    (kphs+krndphs+gk_20) * iInDmem
kphs        wrap     kphs, 0, iInDmem

kphs2       init     0
kphs2       phasor   kphsrate, i(gk_13)  ;** right channel phase can be varied for stereo effect
kphs2           =    (kphs2+krndphs+gk_20)*iInDmem
kphs2       wrap     kphs2, 0, iInDmem

kgliss          =    gk_5            ;** grain glissando
krnddur     linrand  gk_12           ;** random amount of grain duration
kdur            =    (1+krnddur) * gk_6
kris            =    kdur * gk_7
kdec            =    kdur * gk_8
iolaps          =    10

kampenv     linenr   iamp*gk_16, 0,.1,.03

;**            xfund        koct     kris       kdec         ifna     itotdur      kgliss
;**      xamp          xform      kband      kdur       iolaps       ifnb        kphs
;**      ------------------------------------------------------------------------
a1      fof3     kampenv, kfund,  kform1, gk_15, 0, kris, kdur, kdec, iolaps, ifna, 101, 3600, kphs, kgliss
a2      fof3     kampenv, kfund*gk_19, kform2, gk_15, 0, kris, kdur, kdec, iolaps, ifna2,101, 3600, kphs2, kgliss

        vincr    gaout1, a1 * (1 - gk_9) ;** left main output
        vincr    gaout2, a2 * (1 - gk_9) ;** right main output
        vincr    garev1, (a1 + a2) * gk_9     ;** reverb send
        endin

;********** Granular.sco ****************
```

```
f1  0 262144  -22  “c:\csound\icmc\voceGabEng.aif” 0 0 1
f2  0 524288  -22  “c:\csound\icmc\aiff\fm15 a(tema 2).aif” 0 0 1
f3  0 524288  -22  “c:\csound\icmc\aiff\fm15 m(arrich spett arm acute).aif” 0 0 1
f4  0 524288  -22  “c:\csound\icmc\aiff\fm15 t(arricchimento spettrale).aif” 0 0 1
f5  0 1048576 -22  “c:\csound\icmc\aiff\fm15 z(grave con arr spettr).aif” 0 0 1
f6  0 524288  -22  “c:\csound\icmc\aiff\fm15-f(iperarmonici).aif” 0 0 1
f7  0 131072  -22  “c:\csound\icmc\aiff\Gest9 (ottava gliss bassa).aif” 0 0 1
f8  0 524288  -22  “c:\csound\icmc\aiff\GestB (trilli glissati).aif” 0 0 1
f9  0 65536   -22  “c:\csound\icmc\aiff\GestD (batterie glissate).aif” 0 0 1
f10  0 65536   -22  “c:\csound\icmc\aiff\GestF (acciaccature).aif” 0 0 1
f11  0 524288  -22  “c:\csound\icmc\aiff\gyuto1 flang2 (pizzicati e raschiati).aif” 0 0 1
f12  0 262144  -22  “c:\csound\icmc\aiff\gyuto1 uuuamiu.aif” 0 0 1
f13  0 524288  -22  “c:\csound\icmc\aiff\Pigmei A(voce e flauto).aif” 0 0 1
f14  0 524288  -22  “c:\csound\icmc\aiff\Vid Nooo (E3).aif” 0 0 1
f15  0 262144  -22  “c:\csound\icmc\aiff\Vid uuuu(dissonanza).aif” 0 0 1
f16  0 262144  -22  “c:\csound\icmc\aiff\Clar PigmeiTema alto.aif” 0 0 1
f17  0 262144  -22  “c:\csound\icmc\aiff\Clar coppia temi paralleli3.aif” 0 0 1
f18  0 262144  -22  “c:\csound\icmc\aiff\Clar Tema2 retrogrado.aif” 0 0 1
f19  0 262144  -22  “c:\csound\icmc\aiff\Clar coppia temi veloce4.aif” 0 0 1
f20  0 131072  -22  “c:\csound\icmc\aiff\Clar coppia temi veloce6 cromatica2.aif” 0 0 1


f100 0 8192     5  .001 8192 1       ;** exponential curve for slider mapping
f101 0 8192     19  .5 1 270 1       ;** sigmoid curve

;**              unis / third / fifth / Min7th / octave / oct+3rd / oct+5th / oct+7th / double oct
f102 0 32   -2   1    1.25    1.5    1.75    2      2.5      3        3.5        4

;**              unis / -4 / 5  / -8  / 8  / -8 -4  / 8 +5  / -2oct / 2oct
f103 0 32   -2   1   .75   1.5   .5    2    .375     3     .25      4

;**            major just scale
f104 0 32   -2   1   1.125   1.25  1.33333  1.5  1.66666  1.875  2 2

;**            harmonic progression
f105 0 32   -2   1   1.125  1.25 1.375 1.5 1.625 1.75 1.875 2

;**       frequency table
f151    0  32   -2   .5   1   .625  .75  .875  1  1.125  1.25   1.375   1.5   1.625   1.75   1.875  2
```

```
i100 0 3600
e
```

The orchestra is quite complex. So, we'll work backwards, starting from the audio output located at the bottom of instrument, up to controller's inputs located at the top.

NOTE: We recommend that you analyze complex orchestras from output back to input because, normally, the audio output consists of one, easy-to-locate variable (or two in stereo, four in quadraphonic, etc.). Whereas the number of inputs usually is much larger. In this case, there are 21 continuous controllers, plus inputs for the note-number and the velocity of each note. Distinguishing at a glance the variables derived directly from physical inputs in contrast to the intermediate variables is not an easy task. The audio output is like the root of a tree, while input signals are like as branches or leaves of the tree.

Orchestra consists of two instruments:

- instr 100, containing MIDI-control-related opcodes (slider8 e slider16), main output and reverb line;
- instr 1, containing granular-synthesis-related code.

Instrument 100 is easy to read. Starting from the bottom toward the top (i.e. from the root toward the branches), notice that the global audio variables contained in the left and right channels (*gaout1* and *gaout2* variables) and the reverb signal (*arev* variable), are routed to the stereo output. These variables are then zeroed (together with *garev1* reverb line) with the *clear* opcode. The *arev* signal is generated by the *reverb2* opcode. *garev1* (global reverb line) is connected to the reverb after being filtered by a low pass filter (*tonex*). The filtering makes the reverb sound more pleasant when hearing it. The cutoff frequency in this filter is adjustable with slider 10 (*gk_10* global variable). Notice that *tonex* opcode consists of a bank of first-order filters, serially connected. This filtering produces a steeper cutoff curve which generates a warmer sound.

The arguments of *slider8* and *slider16* opcodes are split into several text lines which makes it more legible. (Remember that '\' character indicates a division of a single CSound statement into multiple lines that is recognized as single statement). Our orchestra has a total of 20 signals (adjusted by MIDI controllers). We used two opcodes. The first contains 16 sliders. The second has 8 sliders. So, four sliders remain unused, and can add additional variable parameters in the future by modifying the orchestra.

The following is a list of the global variables corresponding to parameters adjustable by MIDI sliders:

gk_1 select the table containing audio samples (this orchestra can use different samples at the same time);
gk_2 pitch offset, that can be selected by changing the index of a table containing a series of frequential ratios;
gk_3 grain repeat speed, i.e. number of grains per second;
gk_4 phase scanning speed of current sample. This parameter allows to change the duration of current sample without changing its pitch and vice-versa;
gk_5 end frequency of glissando of each grain (see *fof2* and *fof3* manual);
gk_6 total duration of each grain, scaled according to grain repeat speed;
gk_7 attack time of grains, scaled according to their total duration;
gk_8 decay time of grains, scaled according to their total duration;
gk_9 reverb send level;
gk_10 cutoff frequency of low-pass filter used by the reverb output;
gk_11 random amount of grain frequency;
gk_12 random amount of grain duration;
gk_13 left/right channel phase difference;
gk_14 random amount of starting grain phase;
gk_15 octaviation factor (see *fof2* and *fof3* manual);
gk_16 main volume control;
gk_17 harmonic tuning table number;
gk_18 amount of random changes in harmonic tuning table;
gk_19 left/right deviation of grain repetition speed;
gk_20 stating phase offset of grains. If the phase scanning speed of current sample is set to zero (*gk_4* slider), this parameter allow to 'scrub' sampled sound manually.
gk_21 random amount of grain starting time;
gk_22 not assigned;
gk_23 not assigned;
gk_24 not assigned;

Now, let's read instr 1, starting from the bottom.

The three lines of code containing the *vincr* opcode assign left/right channels and the reverb line to the corresponding global variables (*gaout1,gaout2,grev1*) which will be used as arguments in the main audio output located in instr 100.

Notice that reverb line (*garev1*) is obtained by mixing the left and right channels, and by multiplying the result by the current value of slider 9 (*gk_9*) or adjusting wet/dry ratio. Slider 9 also affects the level of direct signals (*a1* and *a2*) placed in the global variables *gaout1* and *gaout2*.

a1 and *a2* variables are generated by *fof3* opcodes. These are the heart of the granular synthesis engine. There are two calls to *fof3* which process a stereo file using the granular synthesis technique.

Even a mono sample can be granulated using two *fof3* modules with slightly different parameters. This produces a stereo effect.

fof3 opcode is derived by *fof2* which is derived by *fof*.

Initially, *fof* was not designed for granular synthesis, but for vocal synthesis using formants derived from IRCAM's *Chant* program (Xavier Rodet et al.).

Thanks to the internal structure, *fof, fof2* and *fof3* can be used in granular synthesis, as well. Compared with *fof* and *fof2*, *fof3* can handle 16-bit integer samples stored into a CSound table (whereas *fof* and *fof2* only handle 32-bit floating-point samples). For this reason, *fof3* is more efficient and wastes less memory. It stores double the amount of samples with the same amount of RAM. Furthermore, *fof3* uses linear interpolation to read samples, while *fof* and *fof2* don't. So, the audio quality is much better.

Let's analyze the input arguments of the first line of code for the *fof3* opcode (arguments on the second line are practically the same):

```
a1      fof3 kampenv, kfund,  kform, gk_15, 0, kris, kdur, kdec, iolaps, ifna, 101, 3600, kphs, kgliss
```

kampenv Amplitude envelope. In this case, signal is generated by the previous line of the *linenr* opcode. In this instrument the amplitude varies according to the note-on velocity received by DirectCsound from the MIDI port (see the line containing *ampmidi* opcode).

kfund When using *fof3* in granular synthesis (instead of vocal synthesis), this parameter indicates the grain repeating speed. During processing, *kfund* is variable and enables control over the temporal distance between grains. If *kfund* doesn't vary, or varies slowly and smoothly, we refer to the process as synchronous granular synthesis. If *kfund* varies chaotically, we refer to it as asynchronous granular synthesis. In this orchestra, the base speed of grain repetitions is adjusted by slider 3 (*gk_3* variable). The random variations in this base speed are adjusted with slider 21 (*gk_21* variable).

kform When *fof3* is used for granular synthesis instead of vocal synthesis, this parameter contains the pitch-transposition of the current grain. The value of *kform* is sampled at the start of grain and remains constant for the duration of the entire grain - even if *kform* varies in the meantime. The concept is the same for the initialization parameters which are 'sampled' at the start of a note and remain constant for entire note duration. The only difference is that,

in this case, *kform* remains constant only inside a single grain. So, if it varies during a note, the next grain will get a new frequency value, but the current grain continues to use the old value. *kform* is expressed in cps which indicates the number of times per second the entire table containing the audio samples is read. It should be clear that assigning a frequency value expressed in cps to this parameter is correct only when a single cycle of the sound waveform is stored into the table (for example, a sinusoid). When the table contains a complex sample, it is better to refer to the period of each cycle. (Period is the inverse of frequency.) In our example, the period corresponds to the duration of sampled sound contained in the table. Suppose to have a sample with a length of one second that contains a tone of 440 cps. If we want to play this sampling at 880 cps, we have to read the table twice fast by assigning 2 for a value of *kform*. If we want to play it at 220 cps, we have to assign 0.5 etc. If our sample has a duration of 1.5 seconds and we want to produce a tone of 440 cps, (the sampling frequency of the original) we have to set the period = 1.5 seconds. But since this value must be expressed in cps, (a frequency unit, not a duration unit) we have to assign the inverse of that period (1/1.5 = 0.6666). If we want to transpose up an octave, *kform* must be set to 1/(1.5/2) = 2/1.5 = 1.3333 (i.e. the inverse of half period, that is the double of frequency) and so on.

So it is important to take into account the total duration of the sample contained in the table, as well as, its original frequency (pitch) and original sampling frequency. We pay particular attention to sounds recorded at sampling frequencies that are different from the orchestra's sampling frequency. In our orchestra, the pitch transposition of grains can be adjusted by slider 2 (*gk_2* variable) and by a MIDI note number (*cpsmidi* opcode) while the random amounts of pitch variation of grains can be adjusted by slider 11 and 18 (*gk_11* and *gk_18* variables). Slider 18 allows a table with micro-tuning ratios to control the frequency of grains randomly. Each grain will assume a frequency value that corresponds to a ratio contained in the current micro-tuning scale.

gk_15 in vocal synthesis, this argument contains *octaviation index* which attenuates the level of odd-numbered grains. In vocal synthesis, this index produces an octave-lowering effect. In synchronous granular synthesis, it produces a rhythmic effect when this index is greater than zero. In our orchestra this parameter can be controlled by slider 15.

0 This argument is not used in the granular synthesis of this orchestra. In vocal synthesis, it expresses the bandwidth of the formant in Hertz.

kris Attack time of the trapezoidal envelope of grains. It can be varied in different grains. In our orchestra attack time can be adjusted by slider 7 (*gk_7* variable).

kdur Total duration of each grain. It can be varied in different grains. In our orchestra attack time can be adjusted by slider 6 (*gk_6* variable). Also, it is possible to adjust random variation of grain duration with slider 12 (*gk_12* variable).

kdec Decay time of the trapezoidal envelope of grains. It can be varied in different grains. In our orchestra decay time can be adjusted by slider 8 (*gk_8* variable).

iolaps Number of overlapping grains. If the time duration of the grains exceeds the time interval between grains, the grains will overlap. This parameter defines the maximum number of overlaps. If the number of overlaps surpass this number, CSound crashes. In this orchestra maximum number of overlaps is set to 10. It is recommended that number to this parameter is not too high. High numbers waste of processing time and degrade real-time performance.

ifna In granular synthesis this parameter must be filled with the number of the table containing the sampled sound.
In *fof* and *fof2*, this table is, usually, filled using *GEN01*. In *fof3* (as in the previous example), it is filled using *GEN22*. (It is identical to *GEN01* except that it fills table with 16-bit samples). In our orchestra it is possible to select a different table for each note by varying slider 1 value (*gk_1* variable).

101 Number of the table containing the shape of the attack and decay segment. It is usually a straight-line segment or a sigmoid. The sigmoid can be obtained by using a cycle of a positive sinusoid shifted 90 degrees out of phase with the first and last quarter of cycle are cut off (by using *GEN19*).

3600 Maximum note duration (not grain duration). In this orchestra is set to 3600 seconds.

kphs This parameter tracks the phase of the table storing the sampled sound. It start with zero and ends with 1. These numbers refer to the start and the end of table. To accomplish this task, normally the *phasor* opcode is used, which generates the phase for a table in which period (i.e. the inverse of frequency) coincides with the duration of table scanning.
By varying the frequency of *phasor*, we obtain different reading speeds. So, it is possible to *stretching* time without affecting the pitch of a sampled sound. If the frequency of *phasor* is set to zero, that is if the *kphs* parameter remains constant, the sampled sound will 'freeze'. In our orchestra, the *phasor* frequency (i.e. table scanning speed) can be adjusted with slider 3 (*gk_3* variable). It is also possible to set an offset defining the initial point of

table scanning. By varying such parameter and setting *phasor* frequency to zero, it is possible to 'scrub' manually by changing the corresponding slider value. In our orchestra this value can be adjusted by slider 20 (*gk_20* variable).

kgliss - In *fof2* and *fof3*, a glissando can take place in each grain. This parameter indicates terminal frequency of the transposition. It is expressed as a scale factor. Consequently, assigning 1 to such parameter leaving the frequency of grain unchanged; assigning 2 doubles the frequency at the end of the glissando (the transposition moves up an octave); assigning 0.5 halves the last frequency (i.e. it transposes down an octave down). If the grain's duration is very short, the glissando will sound like a timbral effect, or a sort of detuning. In our orchestra, this parameter can be adjusted with slider 5 (*gk_5* variable).

ampenv signal, contains the amplitude envelope which is generated by *linenr*. The absolute amplitude can be adjusted with slider 16 (*gk_16*), the main volume control. *iolaps* is set to 10. *kris* and *kdec* (grain attack and decay) are calculated as fractions of *kdur* these fractions can be adjusted by slider 7 and 8 (*gk_7* and *gk_8*).

Grain duration *kdur* is controlled by slider 6 (*gk_6*) and its the random variation can be adjusted by slider 12 (*gk_12*).

The frequency of *kgliss* (grain glissando) can be adjusted with slider 5 (*gk_5*).

The table that contains the sampled sound is scanned separately for the left and right channels (*kphs* and *kphs2*) with the two lines of code that call the *phasor* opcode.

The *phasor* input argument controls the speed of sampled sound scan. This argument (*kphsrate*) can be adjusted with slider 4 (*gk_4*). When *kphsrate* defaults to the value 0, sampled sound scanning is stops on a single point, and all grains are read from only one point of sampled sound table. It is possible to modify the scanning point manually, in this example, with means of slider 20 (*gk_20*). This offsets the current point in the scanning. It is, also, possible to modify the phase-difference between the left and right channel by adjusting slider 13 (*gk_13*). Offsets in the grain reading points of sampled sound table can be implemented using random jumps. The random offset (*krndphs*) is generated by *linrand* opcode and the random influence on sampled sound reads can be adjusted with slider 14 (*gk_14*). Actually, *phasor* generates a signal moving from zero to one like the numbers that represent the phase at the start and end of the table containing the sampled sound. Remember the table length must be a power of two, but the number of table entries needed to store a sampled sound is rarely a power of two. So, the number of sound samples and the total number of elements of the table are usually different. (The table length will be greater or equal to the number of sound samples). For this reason, the *nsamp( )*

function is used. This function returns the number of samples actually read from sample file.

The last part of table remains unused, and is filled with zeroes. If the entire table is scanned, we would hear sound only in the first part of table. For the remaining portion we wouldn't hear anything. To prevent this, the phase of table must not rise all the way to 1, but only to the last which holds a sound sample. (i.e. a number between 0.5 is half the table.) The scale factor *ilnDmem* represents the ratio between the sampled sound's length and the table's length.

In our example *phasor* operates at *k-rate*, but the *kphs* and *kphs2* variables must contain values at the initialization stage so that *fof3* can legally call those variables. For this reason, the two variables are initialized with zero values. The phasor frequency (*kphsrate*) is obtained by the multiplication of the output of *slider* 4 with *isrDdur* which is a scaling factor.

The frequency of granule repetitions (*kfund*) is controlled with slider 3 (*gk_3*) in combinations with a random offset (generated by *trirand* opcode). The value of this offset can be adjusted with slider 21. When the value is not zero, rhythm of the grains will vary.

The frequency of each grain is obtained by two variables *kform1* (left channel) and *kform2* (right channel).

Frequency values are influenced by several factors: MIDI note number, slider 2 (*gk_2*, that initiates a transposition), random generators which vary the frequency of each grain which is adjustable with slider 11 (*gk_11*), and random generators that choose an interval ratio stored in a table storing a scale which is adjustable with slider 18 (*gk_18*).

imemlen, ilen, ilnDmem, ifsr, isrDmem and *isrDdur* are re-scaling factors which affect various parameters. We avoid to go into mathematical details in order to don't annoy the reader, who can study their purpose for himself in any case.

ifna and *ifna2* are numbers of the tables containing the sampled sound (left and right channel). These numbers can be changed with slider 1 (*gk_1*).

iamp used to adjust the amplitude of notes, depending by MIDI note velocity.

ifmidi affects the grain frequency (as well as other factors). It is dependent by MIDI note number.

At the beginning of score several tables (1 to 20) are filled with sampled sounds using GEN22.

Table 100 contains a segment of exponential curve and is accessed with the *slider16* opcode. It is used to create a more natural response when adjusting some sliders.

Table 101 contains a sigmoid curve. Its purpose is to shape the attack and decay segment of each grain. The sigmoid curve is created by shifting a positive sinusoid 180 degrees out of phase 180 and then removing the first and the last quarter of period.

Tables 102, 103, 104 and 105 contain frequency ratios for micro-tuning scales.

Table 151 contains transposition factors.

At last, instrument 100 is activated for 3600 seconds, in order to allow an hour of Csound real-time performance.

At this point we end the lecture section regarding DirectCsound real-time examples.

How do we control all MIDI parameters in real-time? We could connect a MIDI fader box (several such mixers are available commercially) to the MIDI input port of the computer to provide the user with a number of physical sliders (potentiometers) which transmit MIDI control-change messages as they are moved (and are then recognized by the orchestra). Normally, MIDI fader boxes or MIDI mixers have only 16 sliders while the adjustable parameters of orchestras are more numerous. Our orchestra, for example, has 21 parameters.

An alternative to hardware devices is VMCI is a program designed to be used in conjunction with DirectCsound under Windows. This is more convenient in many aspects, as we will see in next session.

4. VMCI (VIRTUAL MIDI CONTROL INTERFACE)

VMCI GUI interface can send most kinds of MIDI messages (all VOICE messages: *note-on, note-off, poly after-touch* and *channel after-touch, control-change, program-change* and *pitch-bend*) to any MIDI port installed on the computer. VMCI is totally configurable and gives the user total control over MIDI message definitions. VMCI can be used in place of a MIDI keyboard or any external MIDI control device. It is possible to control DirectCsound via MIDI even without having any MIDI interface card installed on the computer. This program was specifically designed to control DirectCsound, but it can be used to control any MIDI instrument, hardware and software, as well.

4.1 VMCI modes.

There are two versions of VMCI:

1. *VMCI*
2. *VMCI Plus (commercial version with the unique hyper-vectorial synthesis and other enhanced features).*

In this lecture we only deal with the first version that has two modes:

- *Lite mode*
- *Pro mode*

LITE mode is *freeware*.
PRO mode is *shareware*. This mode can be only activated with a 'magic' code given to persons who pay for the software.

In order to control DirectCsound with VMCI, it is necessary to install «*Hubi's LoopBack device*», a virtual MIDI program which provides MIDI ports without any hardware MIDI interface. The «*Hubi's LoopBack*» package is freeware. It is possible to connect several devices to the same MIDI port. Thus, it is possible to control DirectCsound both with VMCI and a hardware master-keyboard at the same time.

VMCI Pro provides the following tools:

- four *7-bit slider panels*, up a total of 256 configurable sliders that can send any MIDI control-change message;
- two *14-bit sliders panels*, up a total of 64 configurable sliders that can send messages of with a double resolution;

- two *virtual joystick panels*, up a total of 16 configurable joystick areas, each one of these controlling two parameters at a time;
- a *virtual keyboard* panel capable to handle a total 960 different MIDI messages of any kind, according to user configuration. Any one of these message can be associated to a key of the alphanumeric keyboard;
- CSound editing and starting buttons, which makes Csound related operations easier;
- *copy board*, that allow to copy and paste parameter values from VMCI to a CSound orchestra o score.

4.2 The ACTIVATION BAR

When starting VMCI, the activation bar appears:

FIG.1 Activation bar

The *activation bar* is the main window and has the following purposes:

- Open and save setup files (with ***.stp*** extension). Setup files contain most settings made during the working sessions. The main settings are slider and joysticks positions (see below), text labels for every parameters and virtual keyboard configuration.
- Show and hide panels (containing sliders, joysticks, and virtual keyboard), by clicking the corresponding check-boxes.
- Open the «MIDI Setup» dialog-box, that allows to select MIDI out port (in «Plus» version it is possible to activate the MIDI in port too).
- Show and hide panel toolbars in order to gain more space and to enlarge vertical slider length.
- Enable automatic sending of note-on and note-off messages each time a slider is moved, in order to initialize a «global instrument» (i.e. a Csound instrument which handles global variables used by other instruments).

- Enable/disable visualization of numeric values of parameters when they change. Since updating visualization subtracts a lot of CPU processing time, this option is useful to enhance Csound realtime performance by freeing processing resources.
- Visualize and clearing «Copy Board» that contains all current slider positions.
- Csound related buttons («*Csound settings*», «*Run Csound*», «*Edit orc*» and «*Edit sco*» that allow to run up to two instances of Csound at the same time, and to edit corresponding orc/sco pairs. «*Csound Settings*» button allows to setup all parameter corresponding to Csound calling inside VMCI).

4.3 Slider-panels

Slider-panels contain a group of scroll-bars (the number of available sliders is different in LITE and in PRO mode). By moving these scroll-bars MIDI control-change messages are sent to the MIDI out port. Each panel emulates the functionality of a MIDI fader box or a MIDI mixer. Two different resolutions are possible in sending parameters: 7-bit messages covering a range of 0 to 127, and 14-bit messages (consisting of a pair of 7-bit messages) covering a range of 0 to 16383.

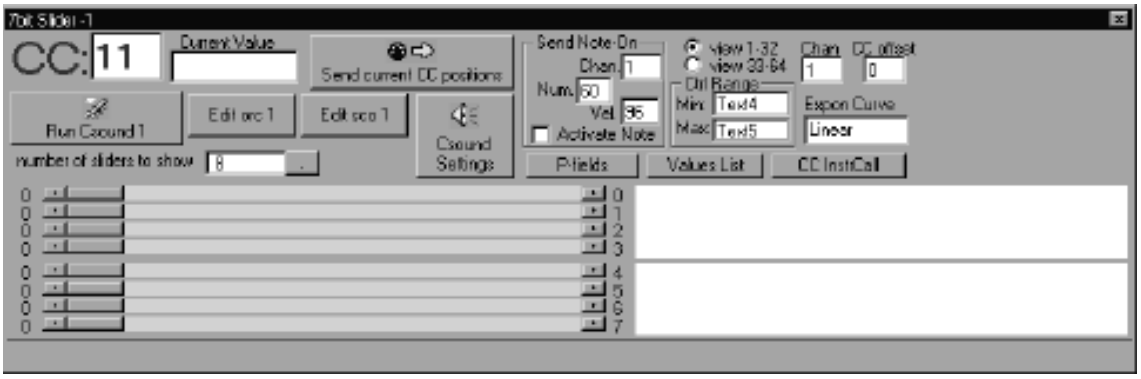

FIG.2 7-bit Slider Panel (horizontal)

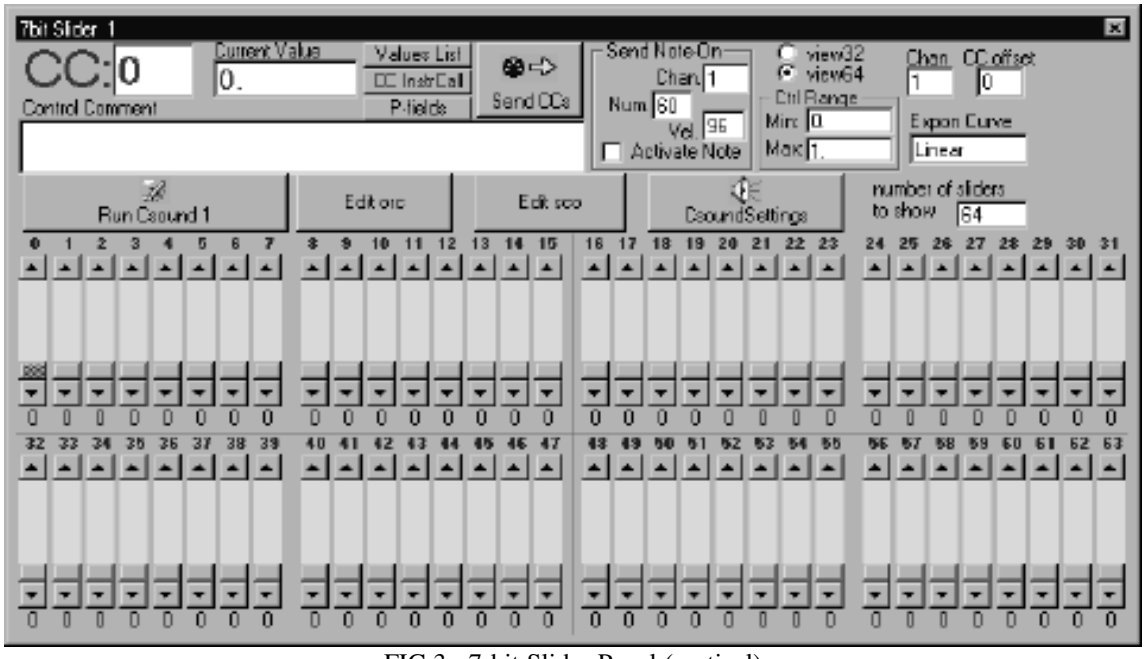

FIG.3 7-bit Slider Panel (vertical)

VMCI Pro mode allows to use up to six slider panels at the same time: two horizontal 7-bit slider panels (fig.2), two vertical 7-bit slider panels (fig.3) and two 14-bit slider panels (fig.4).

Each 7-bit panel contains a total of 64 sliders. The horizontal sliders can only display up to 32 slider at a time whereas the vertical sliders can display all 64 sliders. On the other hand, the horizontal ones display all the slider comments at the same time.

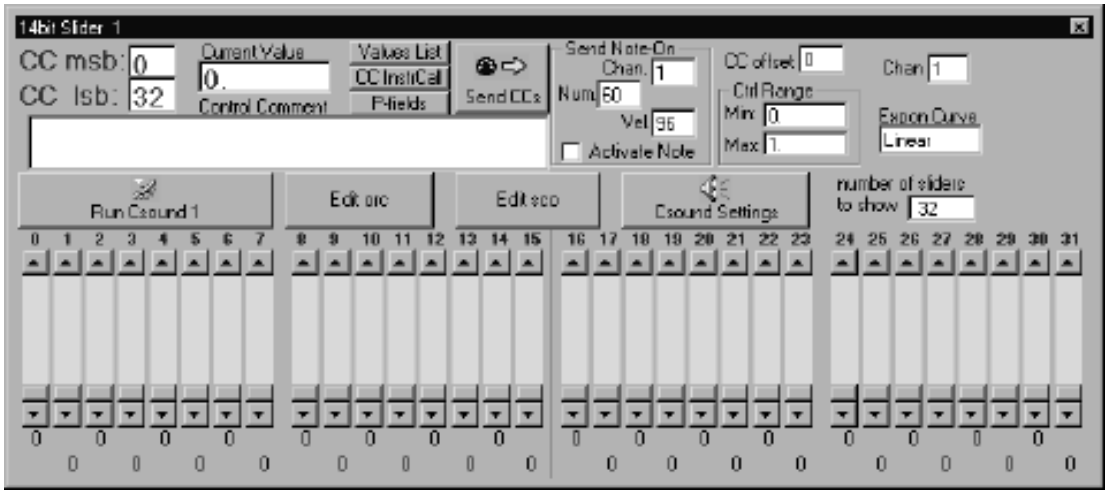

FIG.4 14-bit Slider Panel

When dragging a slider with the mouse, a stream of MIDI control-change messages are sent, and the following information is displayed in accordance with the slider position:

- control-change message number, corresponding to its actual configuration (that can be modified by the user);
- control-change message data value sent to the MIDI out port;
- re-scaled data, as it is interpreted by CSound (default re-scaling range is 0 to 1, and can be reconfigured by the user). Such value is useful to monitor what actually happens in the corresponding CSound instrument;
- slider comment (in horizontal slider panel the comments of all sliders are visualized at the same time), that can be defined by user for each slider.

VMCI allows to save all current slider positions of all panels in the setup file, and to send all MIDI messages corresponding to the present state of a panel by pressing «*Send current CC positions*» button. This restores the identical sonic configuration in a different CSound session, containing many parameters. This feature is normally not present in hardware MIDI fader boxes.

4.4 Virtual joystick panels

VMCI includes two panels, each one contains eight squares which emulate an analog joystick. By dragging a mouse over these areas, two different MIDI control-change messages are sent simultaneously. The first corresponds to the horizontal position of the mouse. The second corresponds to the vertical position. So, it is possible to control two parameters with a single mouse movement. An action which is impossible with sliders (fig.5).

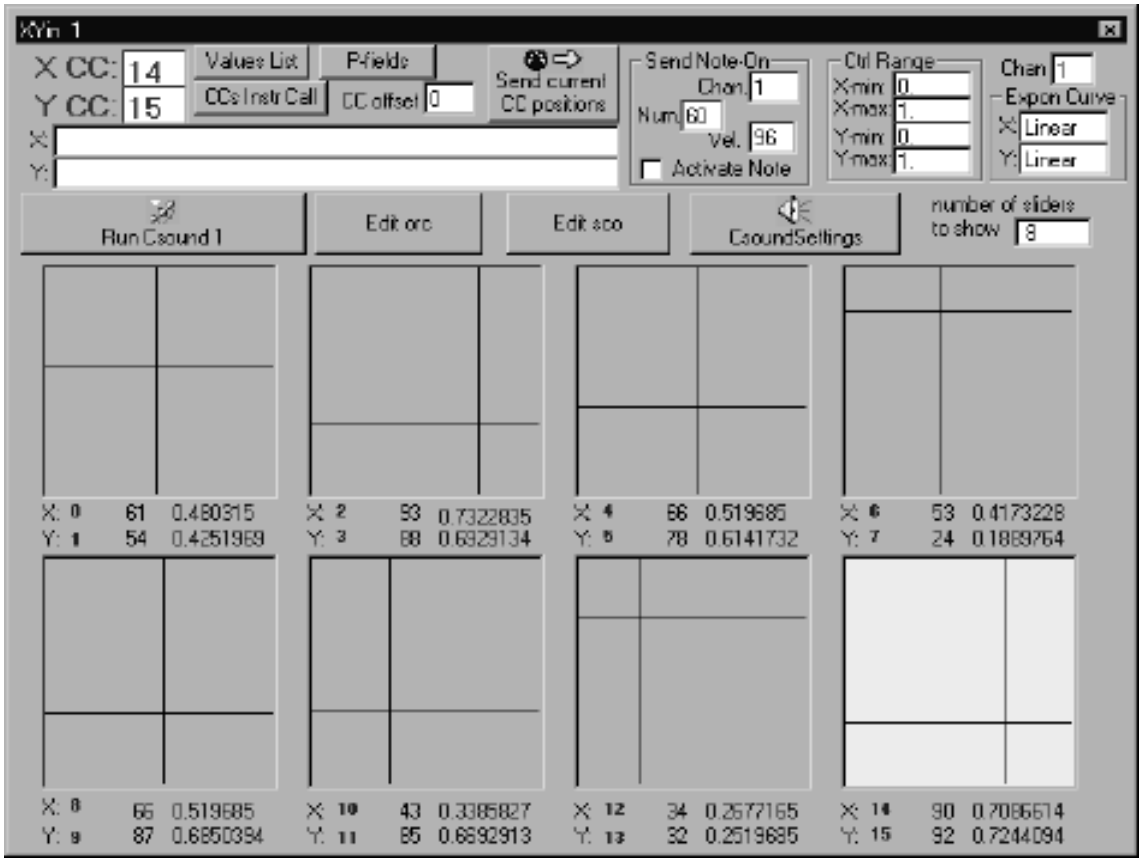

FIG.5 Virtual Joystick Panel

4.5 Virtual keyboard panel

Virtual keyboard panel (fig.6) sends any kind of MIDI VOICE messages, not only note-on and note-off messages.

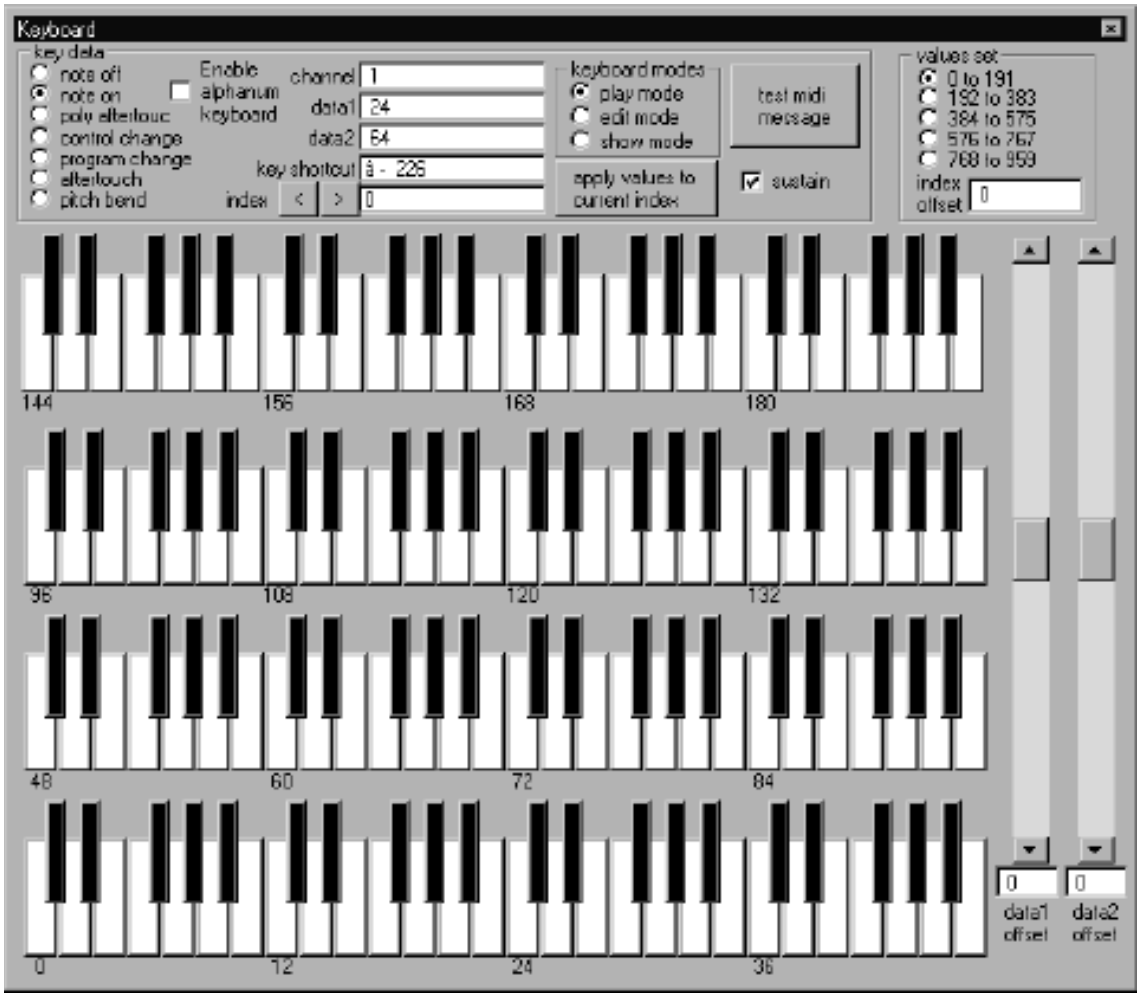

FIG.6 pannello della tastiera

Each button appears as a piano-key and can generate a note-on/off message, as well as program-change, pitch-bend, control-change or after-touch. Each key is totally configurable by the user and can send any VOICE message the user assigns to it. Furthermore, it is possible to assign each key of the alphanumeric keyboard to one of the piano-like buttons of the panel, making it possible to play the computer keyboard as it

was a piano. Up to 959 MIDI messages can be assigned to the piano-like keys and current user configuration can be saved in the *.stp* file. Also, it is possible to add (or subtract) an offset to all data bytes in the current configuration by means of the two sliders located at the right of the piano-like buttons.

(Translated from the Italian by the author)