Csound Synthesis Approach for the iVCS3 iOS App
Eugenio Giordani(*) – Alessandro Petrolati(**)

(*)**LEMS**
Laboratorio Elettronico per la Musica Sperimentale
Conservatory of Music G. Rossini – Pesaro - Italy
(**)**apeSoft**

Introduction

This article is about the process of creation of an iOS based app for the digital emulation of the famous electronic synthesiser VCS3 by EMS using a Csound orchestra as sound engine. Despite the out of standard features of the original instrument, we have attempted to clone a large amount of details regarding the sound generation, the connectivity, the look and its specific original ergonomy. We will try to illustrate the general synthesis strategies adopted in creating this musical synth application and in particular, we will focus on Csound code of the main sound modules and their relative individual modeling approach.

I. The origins of the project

The recreation of a vintage sythesizer involves a lot of problems related to the difficulty in designing a virtual analog sound machine with digital means. Each design approach cannot ignore that the main goal is to produce waveforms with extended frequency content while minimizing the side effects of aliasing artifacts. The first challenge was to realize a synthesis engine using as much as possible the original Csound opcodes library maintaining a reasonable level of code complexity.
The early attempts were made during a conservatory class exercize in the middle of 2007 using MacCsound frontend and already in that first implementation was also added to the sequencer module trying also to mimic the control interface (see fig.1)
One of the strengths of the original EMS-VCS3 resides in the great potential offered by the matrix of connections compared to other machines, similar to what concerns the sound generation but much more simplified in regard to the general connectivity.

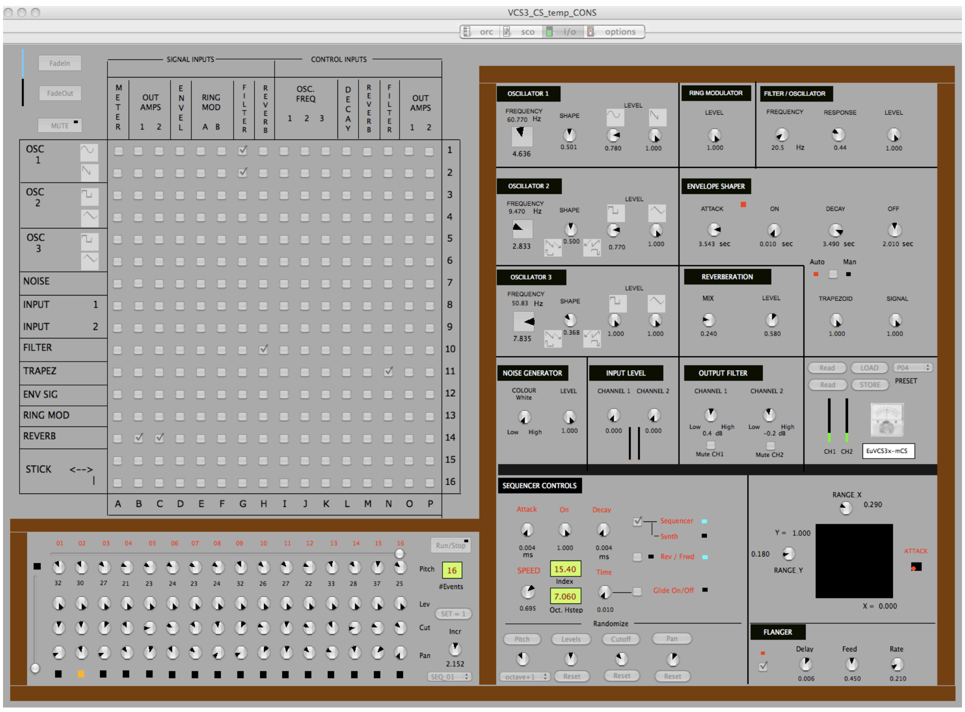

Figure 1. The early GUI of iVCS3 realized with MacCsound (2007)

II. The sound generation engine

As it was anticipated, the sound synthesis and controls has been designed with a Csound orchestra consisting of about 2200 lines of code including some comments. The orchestra includes the followings principal structuring elements: 28 instruments, 18 UDO custom opcodes, 13 macro definitions, 17 function tables, a 16 global audio variable vector, a 16x16 global control variable matrix and 12 general use global audio/control variable.
We adopted CsoundQt frontend and Csound6 for develop and test each individual sound module.

III. Modeling the VCS3 sound modules

The EMS VCS3 synth is essentially based on subtractive synthesis model that includes:

- 2 wide range audio VCO (1Hz – 10 Khz) and one LFO VCO (0.025Hz – 500 Hz) plus a White Noise module as sound sources
- a Ring Modulator unit
- a VCF (Filter/Oscillator) unit with cut off frequency range from 5Hz to 10 Khz
- an Envelope Shaper based on a trapezoid shape
- a Spring Reverberation Unit
- Input/Output amplifiers
- the connections Matrix

The main approach taken in the realization of the sound generation has been to use a modeling approach based mainly on the effects combined with a high coherence of the individual parameter range and consistency of all the external user parameters. For this reason, we have spent a long time carefully analyzing the profile of the voltages in many patches and in particular by analyzing the mapping of each user parameter including some known idiosyncrasies such as 0.32 V/oct for the equal tempered pitch scale or the negative performance of the envelope function.

IV. The Matrix

To illustrate the methods used, the example below shows a small 4 x 4 matrix (see fig. 2) which communicates with *Csound* in a simple and safe way: through the 'score message'.
They have been implemented two versions: the first use the '*zak patch system*' of *Csound* while the second (default) uses the *pull* mechanism to sum the audio (variable '*a*') of the connections. The second version uses the '*Array Opcodes*', introduced since *Csound*6. We could consider an array as N-mixer with N-channels, where the mixer and channel numbers depends from MATRIX_SIZE. In the example we will have 4 mixer to 4 channel, we can think the columns as the mixers and rows as the mixer channels. Every mixer produces the sum of own 4 channels, then are required 4 units of sum for each mixer. The matrix of the app iVCS3 is 16 x 16 and thus requires 16 additions, in this case also 16 multiplications are needed because each channel can be scaled according to the type of connecting pins on the matrix (white or green pins). In theory, when all connections in the matrix are active, we will have to calculate 32 operations for every mixer (16 sum + 16 mul) x 16 mixer, i.e. 512. We will see how the calculation is optimized to reduce CPU load .
The UI is implemented in *CoreGraphics,* we have two delegates (*callback*) for the notification of connections and disconnections (pins). Through these two functions it is possible to activate and deactivate the *Csound* instrument which implements the matrix functions.
The UI array (matrix) returns values in the range 0 ÷ 15 through the delegates.

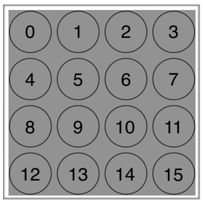

Figure 2. A 4x4 example matrix

Using the zak ystem the matrix communicates with *Csound* through a 'score message' in order to activate (or deactivate) the 'matrix instrument' (i.e. *instr* 1). Practically, we activate N-instances for N-connections of 'matrix instrument'.
The focus point is that the activation of *Csound* instrument, occurs with a fractional *p1* (e.g *instr 1,000, instr 1,001, insert 1,015 etc.*) according to the index of the matrix

The activations happen in the Objective-C *matrixConnected* delegate method, please focus on the p1-field:

```
;p1         p2    p3    p4    p5
i 1.000     0     -1    0     0
i 1.004     0     -1    1     0
.
i 1.015     0     -1    3     3
```

The fractional *p1* is the univocal reference to the activation number that we will use to turn off the instrument when is disconnecting a *pin*.

In the example, *p1* of 1.000 (i.e. 1,000) identifies the connection *p4_0* in *p5_0*; 1.004 (i.e. 1,004) the connection *p4_1* in *p5_0 and* 1.015 the *p4_3* in *p5_3*.

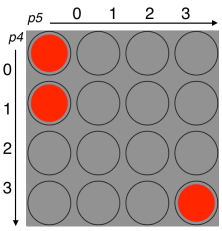

Figure 3. Some matrix connections activated

Note also that *instr* 1 is activated with a negative *p3* which involves an infinite duration, In fact, this instrument performs the sums (and multiplication) of connections every *ksmps,* therefore it must remain active throughout the performance of *Csound*.

To turn off the instrument (disconnect a pin), we need to call it, again with the *instr* num (i.e. fractional *p1*) used for activation but with negative sign, this is done on the *matrixUnconnect* delegate:

```
;p1         p2    p3
i -1.000    0     0
i -1.004    0     0
.
i -1.015    0     0
```

For convenience, it was necessary to identify each connection on the cartesian axes X and Y, these values are expressed as *p4* and p5 (i.e. Y and X axes). We may think that *p4* is the channel number of the mixer *p5*. The values are calculated in the Objective-C *matrixConnected* delegate, as follows:

```
int mixer   = pinNum % MATRIX_SIZE;
int channel = pinNum / MATRIX_SIZE;
```

To better understand the matrix, we need to focus on the first instrument (i.e. *instr* 1) and, for this specific case, on the USE_ZAK block.

```
;--------------------------------------------------------------
instr 1 ; MATRIX PATCHBOARD
;--------------------------------------------------------------
; Y = p4 Channel
; X = p5 Mixer
; p6 = 1 Matrix connected; 0 Matrix un-connected
iChannelNumber init p4
iMixerNumber init p5
iMatrixState init p6

#ifdef USE_ZAK
      ain    zar  iChannelNumber
      zawm ain, iMixerNumber + $MATRIX_SIZE, 1
#else

      gkMatrix[iMixerNumber][iChannelNumber] = iMatrixState
      turnoff
#end
endin
```

The instrument consists of a few lines of code but hides a fairly complex operation, we reads the *zak* memory from the *p4* channel (*iChannelNumber*) which will is copied in the *p5 + $* MATRIX_SIZE (*iMixerNumber*).
The connections in the matrix, involve the consequent activation of the instrument 1 (in proportional numbers) which sums all the channels (rows) connected to the mixer (column), this operation is performed by the *zawm* opcode.

Since we have an one-dimensional array, the offset $MATRIX_SIZE assures the copy of the mixer master on the second part of the array:

Indexes for reading the channels:
0 ÷ $MATRIX_SIZE-1

Indexes to write the mixer sum (master)
$MATRIX_SIZE ÷ ($MATRIX_SIZE*2)-1

Now, we understand why *zak* has been initialized with a twice value of $MATRIX_SIZE*:*

```
zakinit  $MATRIX_SIZE*2, 1
```

The mixer-master is accessible, for the instruments (such ‘out L’, ‘out R’, reverb and flanger), through the *GET_MIXER_MASTER* UDO (i.e. User Defined Opcode).

Please focus on the USE_ZAK block-code:

```
;-------------------------------------------------
      opcode GET_MIXER_MASTER, a, i
;-------------------------------------------------
ichannel  xin

#ifdef USE_ZAK
ichannel += $MATRIX_SIZE
      asum  zar ichannel
      zacl ichannel, ichannel
#else
aSumOfChannels = 0
kndx = 0
loop:
      if (gkMatrix[ichannel][kndx] > 0) then
            aSumOfChannels += gaMixer[kndx]; * gkMatrix[ichannel][kndx]
      endif
loop_lt kndx, 1, $MATRIX_SIZE, loop
#end
xout aSumOfChannels
xout asum
      endop
```

For convenience the indexes are expressed in the range 0 to $MATRIX_SIZE-1, achieved by re-introducing the offset as in the writing process, at last we clean the *zacl* array.

Matrix with Array

We employ two arrays: *gaMixer* (one-dimensional) and *gkMatrix* (two-dimensional) to send and receive the signals between the Orchestra's instruments. Both arrays are initialized with the maximum number of connections of the matrix (i.e. $MATRIX_SIZE ^ 2).

```
gaMixer[] init $MATRIX_SIZE
gkMatrix[][] init $MATRIX_SIZE, $MATRIX_SIZE
```

The *gkMatrix* contains the state of the matrix and *gaMixer* contains the output signals of the instruments of the Orchestra.
In this case the instrument (*instr* 1) perform one only task and, unlike the *zak* system, it must be turned off. from the matrix's delegates, we turn on the instrument for a minimum time in order to copy the status of the matrix in *gkMatrix*. The minimum time required is 1/*kr* (i.e. one *ksmps* block), which is enough to copy *iMatrixState* in *gkMatrix*. The vector indexes are the mixer (column *p5*) and channel (row *p4*).

```
gkMatrix[iMixerNumber][iChannelNumber] = iMatrixState
turnoff
```

Unlike *zak*, the bulk of the work is done da *GET_MIXER_MASTER*, that performs a loop on all connected mixer channels (column), and accumulates them on *aSumOfChannels* variable.The construct 'if', it is necessary in order to optimize the performance since the unconnected channels are skipped.

```
aSumOfChannels = 0
kndx = 0
loop:
      if (gkMatrix[ichannel][kndx] > 0) then
            aSumOfChannels += gaMixer[kndx]
      endif
loop_lt kndx, 1, $MATRIX_SIZE, loop
```

The instruments (such 'out L', 'out R', reverb and flanger) that require the audio in input, will get the signal from the own mixer:

```
instr 14; OUTPUT
;receive signal from mixer 0
inputSignal init 0
aIn_L = GET_MIXER_MASTER(inputSignal)

;receive signal from mixer 1
inputSignal init 1
aIn_R = GET_MIXER_MASTER(inputSignal)
outs aIn_L, aIn_R
endin
```

For instance, by connecting both oscillators (1 sine and 2 saw) on the 'Output R' input slot, the *aIn_R* will contain the sum of the two oscillators (i.e. mixer-master 1).
Concluding, in the *zak* case the mixer's channel accumulation are performed on the 'matrix instrument', while for the second case are performed by *GET_MIXER_MASTER* UDO.
The *zak* system is more faster than the second case but less accurate, it is suitable only to convey the control signals (variable '*k*'). This second case is currently used for the iVCS3 app's implementations.

V. The Sound Sources

In early stage of development we start to implement the ocillator starting from the BLIT (Band Limited Impulse Train) [1] [2] approach that essentially try to reproduce digitally the classical set of synth waveforms as a combination and integration over time of band limited impulse trains. The following lines illustrate this basic method:

```
aBLIT_0 gbuzz .5, kcps, knh, 1, kmul, 1   ; generate band limited impulse train (BLIT)
aBLIT_0_AC dcblock aBLIT_0                ; DC block it
adel interp kpwm + kper_milli/2     ; convert k-rate to a-rate variable (PWM control)
aBLIT_180 vdelay3 -aBLIT_0, adel, 1000 ; generate out of phase BLIT according to PWM
aBLIT_180_AC dcblock aBLIT_180            ; DC block it
aRAMP integ aBLIT_0_AC              ; generate RAMP via direct BLIT integration
aSQUARE integ aBLIT_0_AC + aBLIT_180_AC   ; Generate SQUARE via direct sum BLIT in and
out of phase
aTRI_0 integ 0.025 * aSQUARE        ; Generate TRIANGLE via integration of SQUARE
aTRI_AC dcblock aTRI_0              ; DC block it
aTRI balance aTRI_AC, aRAMP         ; Balance TRIANGLE wave amp with respect of RAMP
```

where *kcps* is the frequency, *knh* the maximum number of partials and *kmul* the multiplier in the series of amplitude coefficients that is reduced of a factor of two when sine wave is selected. In figure 3 it can be seen the clean spectrum of the ramp waveform at a 2 KHz fundamental pitch.

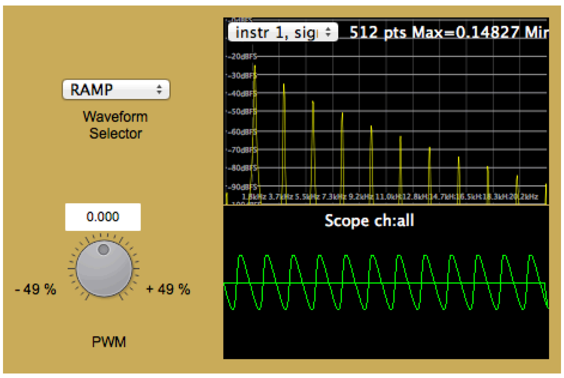

Figure 4. CsoundQt prototype test. Ramp waveform generated with the BLIT algorithm. Note the absence of aliased frequencies.

Despite the quick solution we noticed two different drawbacks. The first was some lack of bandwidth and other it is important to remember to say that in its digital reconstruction has tried to reproduce the most obvious characteristics and to mediate between different needs. The lack of waveform coherence at very low frequency (we remeber that the oscillators works also as modulation sources) required us to using an hybrid approach with sampled waveform and the opcodes *vco2ft* and *oscilikt* together with an interpolation process.

VI. The Ring Modulator and Reverberator

These two units was designed in different ways and we decided to let the user choose between different alternatives and actually through the options you can choose which one to use.
In digital world, the RM is implemented by a trivial multiplication between two audio signals but in the analog domain, things are a little more complicated. In the original VCS3 the modulator is based on the Gilbert circuit whose transistors mismatches are responsible in circuit asymmetries and related spurious components that are generated for that reason. Our implementation derives from the simplified digital approximation described by R.Hoffman-Burchardi and it essentially consists of a simple expression then includes the main non linearity (tanh) that derives from the circuit analysis.

```
aCAR = tanh(aCAR_1) ; carrier shaped by a non linear function
aRM_VCS3 = (aMOD + k1*aCAR) * (aCAR + k2*aMOD) + (k3 * aCAR) + (k4 * aMOD)
aout = aRM_VCS3
aout atone aout, 16     ; sub audio components hi-passed
```

where k1=k2=k3=k4 <= 0.01

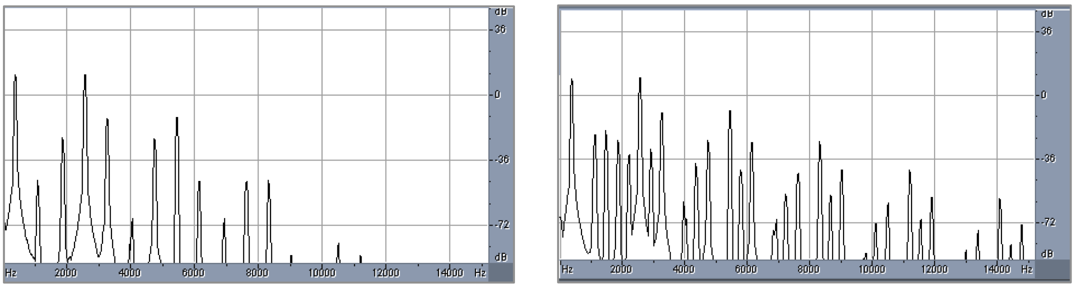

Figure 5. The two diagrams show the spectrum differences between a raw implementation and the digital emulation of the Gilbert circuits. Noticeable spurious frequencies appears inside the spectrum.

Another special feature of the VCS3 sound is the well known spring reverberator. In order to reproduce the characteristic howling metallic sound produced by the springs, we have been adopted also in this case several alternatives for its emulation but in this article we will present only one related to what looks like a physical modeling approach.
This implementation of the spring reverb was loosely inspired by the model proposed by Parker [5]. In this model, the impulse response presents a double sequence of *chirped* echoes, the first below the frequency threshold of 5Khz and the second up to the entire audio band with different distribution of time arrivals of each group of components. In order to create necessary dispersion of the frequencies and produce therefore the chirped signal, a first-order allpass filters have been connected in series. The output was further processed with unit delays and filtering to reproduce convincingly the original response. For this reason, the development of this module (as the other modules of the synthesizer) benefited from the easy way in which you can, using CsoundQt as fast prototyping tool, set and test each parameters to find and freeze into the target application.

```
aAP_1       alpass ainput, irvt, idel     ; chirp filter dispersion section start
.                        .
.                        .
aAP_7       alpass aAP_6, irvt, idel
aAP_8       alpass aAP_7, irvt, idel      ; chirp filter dispersion section end
;
aAP_DEL           delay aAP_8, 0.059      ; 59 ms is the VCS3 reverb pulse repetition
aSPRING_LOW_SEC   tonex aAP_DEL, iFC-500, 8
aSPRING_DIFFUSE   nreverb aSPRING_LOW_SEC, 4, 0.15
aSPRING_HI        atonex aAP_DEL, iFC, 3
aSPRING_HI_DEL    delay aSPRING_HI, 0.002

aSPRING_HI_SEC    = aSPRING_DIFFUSE*0.2 + aSPRING_HI_DEL
arev              = (aSPRING_LOW_SEC + aSPRING_HI_SEC*0.3) * 2.0
```

VII. The Filter

The heart of every synthesizer is represented largely by the filter that is able to define his characteristic sound mark. In the case of VCS3 this statement is doubly true because the filter has some peculiarities that make it unique. It is important to say that in its digital reconstruction we have tried to reproduce the most obvious characteristics and to mediate between different needs. The main choice was to adopt the “moogladder” opcode based on the work of Antti Houvilainen [4].
In some historical 'patch' (i.e. doopesheet) of musicians and engineers, we can understand many things about how it was used in this sense. With an high Resonance the filter produces a pseudo-sinusoidal signal and may be used as FM module (i.e. FM Frequency Modulation). For instance, it could be used as carrier oscillator which is modulated through the cutoff parameter, or vice versa.
The Csound opcode used for the realization of the filter is the **moogladder** by Victor Lazzarini (based on the work of Antti Huovilainen). Our implementation simplifies and lightens the code eponymous UDO (always by Lazzarini). See the Resources for the download link of the UDO.
From this base, we have also tried to develop a parametric model that took into account two characteristic behaviors of the device: the first concerns the behavior of the frequency response when the resonance is increased while the second concerns the transformation of the filter in a real oscillator when the resonance exceeds a certain threshold. These two behaviors were implemented by simply adding a high-pass filter placed in series and an sine oscillator s with frequency equal to the cut-off actual frequency and with amplitude controlled by a function dependent on the amount of resonance.

```
ares        interp kres  ; Change k-rate resonance value  into a-rate variable
amp_exp     tablei ares, 7, 1  ; Scale auxiliary whistle oscillator amp with kres
                               ; value (table 97)
aosc oscil ireson_OSC_amp*amp_exp, acut, 6      ; generate aux whistle
aHP atone afil, acut_glide                      ; 1st order hi-pass filter
aFILMIX = aHP * ares + afil * (1-ares)          ; cross-fade of moogladder and HP
afil =  (0.72 - amp_exp) * aFILMIX  + aosc      ; add oscillator (aosc)
```

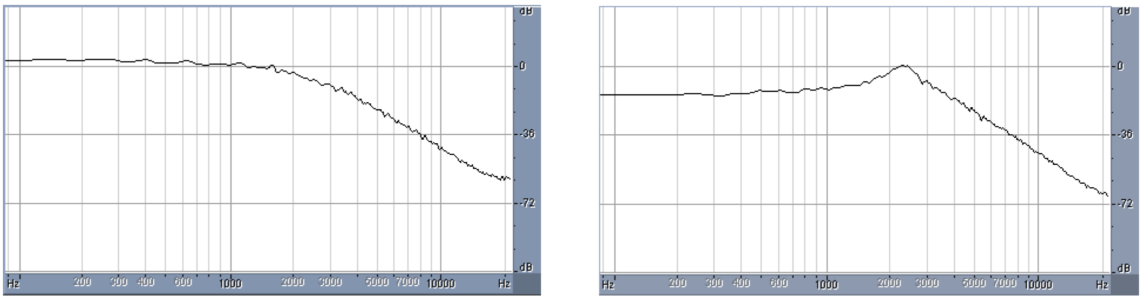

Figure 6. Frequency responce of the VCF with cutoff at 2.5 Khz and no resonance at all (left) and medium resonance (right). Notice the hi-pass effect on the left sido of the spectrum.

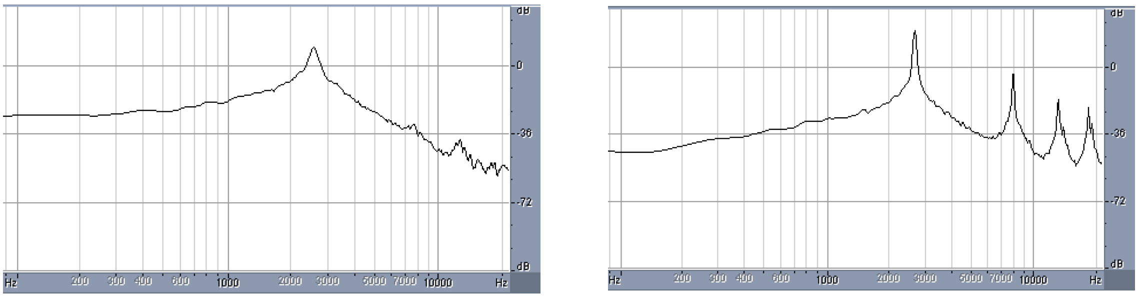

Figure 7. Frequency responce of the VCF with cutoff at 2.5 Khz and hi resonance (left) and very hi resonance (right). Notice the hi-pass effect on the left side and the morphing into a real oscillator responce with a certain amount of related harmonics.

VIII. The Envelope Shaper

The implementation of the Envelope Shaper module has demanded us a very accurate phase of analysis and study, In fact, the features of this module, as you can see from the figures 8 and 9, are not usual and inherently hide a series of difficult behavior (expensive) to implement it.

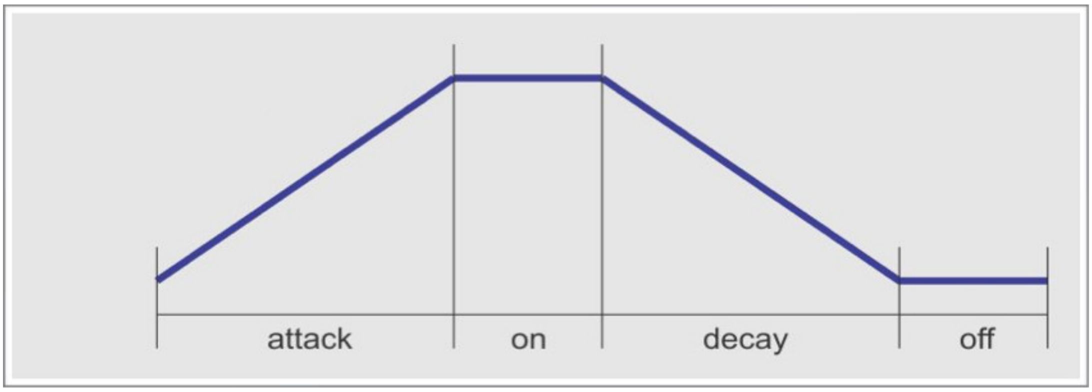

Figure 8. VCS3 Basic envelope (Trapezoid)

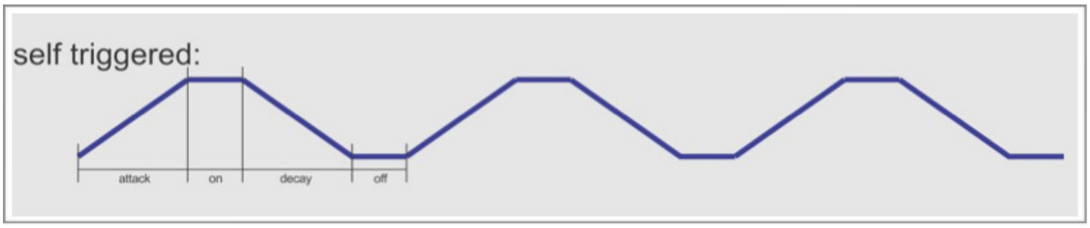

Figure 9. Self triggered envelope.

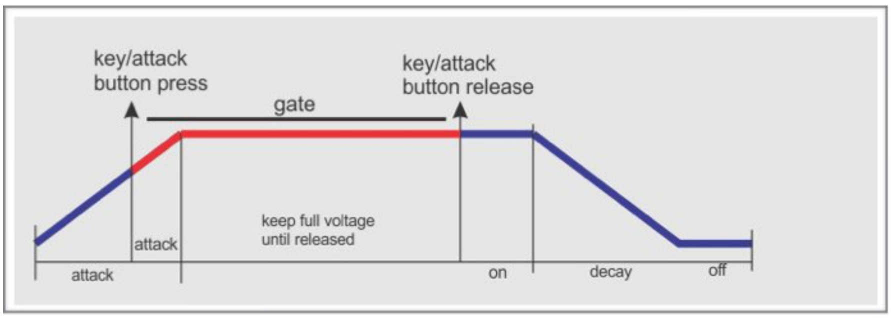

Figure 10. Envelope re-trigger when it is in Attack phase: the Attack phase will continue from the current voltage it had when triggering.

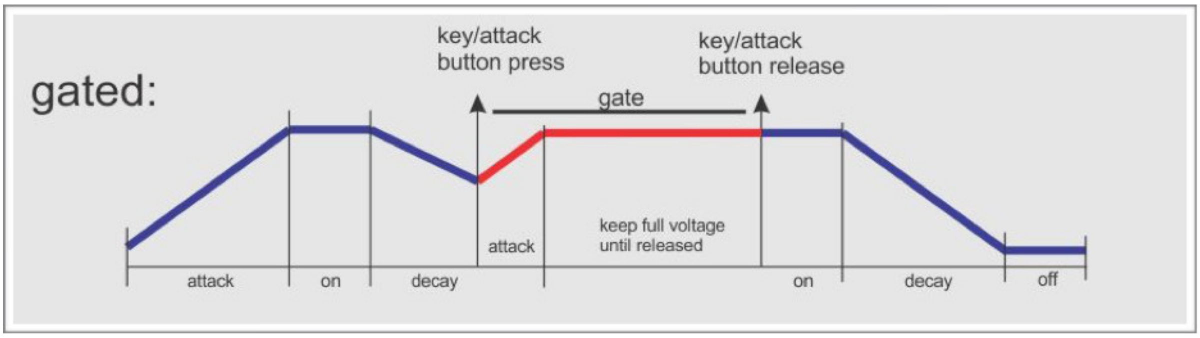

Figure 11. Envelope re-trigger when it is at the Decay phase. The instant you re-trig the envelope with the keyboard (or ATTACK button), it will finish the Attack phase from the current voltage value and then it will stay at full amplitude until you release the key (or button).

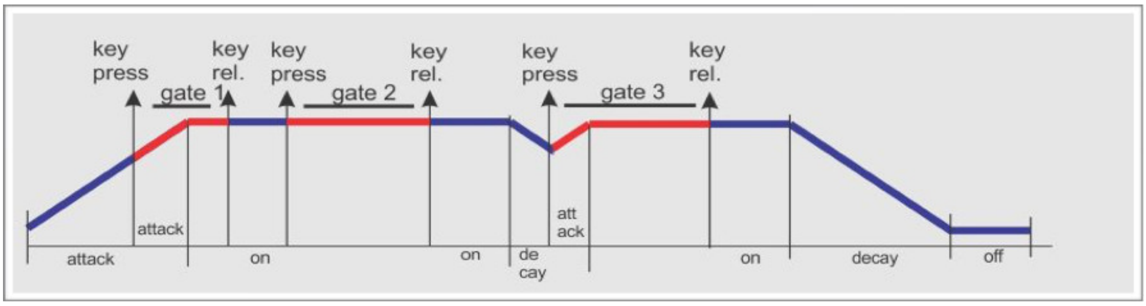

Figure 12. The first gate is the same as fig. B, but now notice the second gate behaviour: since the envelope is at full Voltage (still at hold phase) when you push the key/ATTACK, there will be not a 'retrigger' for the Attack but the Voltage will hold for as long as you keep the key/ATTACK pressed. Gate 3 is the same as fig A.

When you trigger the Envelope from either keyboard or ATTACK button, the Envelope always goes to the Attack phase of the Envelope, but it will begin from the current vaulue. Once released the key or ATTACK button, the Envelope continues from the end of the Attack phase so it will go to the On (steady state), and then Decay and Off.
The Envelope Shaper is a particularly important module of VCS3, which deserves particular attention. The *Csound* code is based on the UDO, the implementation needs a strong use of controls on the audio variables '*a*' and the UDO architecture makes it possible, since it allows to set the local *ksmps* to 1. Infact when *ksmps* is set to one, the variables '*k*' and '*a*' are sampled with the highest rate (i.e. sampling rate), unfortunatley this approach is devastating from the point of view of the CPU load, because of the heavy overhead of function calls that is introduced. This implementation did not allow to run the app on first generation devices, such iPad 1, 2.
See the Resources for the download link of the *Csound* resources for this text, and focus on the EnvelopeApe UDO in the ***VCS3_Envelope.csd*** file.

To overcome this limitation, we had to implement the Envelope, 'outside of *Csound*' and add a new opcode to the list of opcodes with the following *Csound* API:

```
/* Append External Csound Opcodes */
csoundAppendOpcode(cs, "VCS3Envelope", sizeof(VCS3ENVELOPE_OPCODE),
                    0, 3, "a", "kkkkk",
                    iVCS3Envelope,
                    kVCS3Envelope,
                    aVCS3Envelope);
```

At the moment the VCS3Envelope code is a simple adjustment to the language 'C' of *Csound* UDO that it might to be optimized in the next **iVCS3** updates.

IX. Voltage to Amplitude Mapping

To make the emulator as close as possible to the original, all the audio signals of the various modules have been "tuned" according to the amplitudes of the original. Using the standard digital normalized range we have to consider the values in the range -1 to 1 and all the signals have been amplified or attenuated according to the voltages in volts of the original VCS3.

	Max. Output V p-p	Range Min. (Hz)	Range Max. (KHz)	Sensitivity V/Octave
Oscillator 1	3			
	4	1	10	0.32
Oscillator 2	4			
	6	1	10	0.32
Oscillator 3	4			
	6	0.025	0.5	0.26
Filter	5	5	10	0.20

Figure 13. Original VCS3 oscillators / filter voltages (Vpp) vs. frequency reference and sensitivity (V/octave)

As we can see from the figure 13 (from the original VCS3 manual), the maximum value of the AC voltage is 6 Vp-p. Therefore the value of 'peak' is 3, since the emulator's digital audio modules return normalized values in the range -1 to 1, the attenuation/amplification factor is calculated in *Csound* as follow:

```
#define     MAX_VOLT_REF #3.0#
instr 11 ; VCO 1
//…
      /* Max. Out. 3V p-p  ossia 3/2 = 1.5V peak*/
      iampSine init 1.5 / $MAX_VOLT_REF

      /* Max. Out. 4V p-p  ossia 4/2 = 2V peak */
      iampSaw init -2 / $MAX_VOLT_REF
//…
endin
```

For what concerns the input signals, instead:

```
iVoltPerOctave init -0.32 / $MAX_VOLT_REF
```

In the example of the VCO 1, the 0.32 value refers to the sensitivity in Volts per octave. It means that summing 0.32 Volt at the frequency value in Volt of the VCO, we will produce a octave-up and subtracting it we will produce an octave below. The negative sign (-0.32) is justified by the fact that the VCA (i.e. Voltage-controlled amplifier) modules of the VCS3 produce a reverse current.
Finally, the POWEROFTWO (UDO) calculates the factor to raise/lower in frequency according to the Volt amount.

```
apower POWOFTWO aControlIn/iVoltPerOctave ; 2^(aControlIn/iVoltPerOctave)
acps   mac kcps, apower
```

All of these details make the emulator very close to the original, in terms of "playability" and feedback.

X. Knobs

For the programming the Knobs, has been reserved the maximum attention to the non-linearity of the original VCS3 machine. Every widget was designed to follow finely the curve according to the voltages of the original that, in some cases, becomes particularly discontinuously. Since the standard math shapes as exponential or logarithm, they could not adequately approximate the original curve, it was decided to use the technique table look-up.
In this modus-operandi, the values of the Knobs are used as indexes of tables to 11 points, the tables are filled at compile-time with 11 values (samples) measured on the original machine. Intermediate values are obtained through a process of linear interpolation.
The algorithm is easily solvable in *Csound* environment, using the tables (*gen* 2) and the opcode *tablei* which serve to read the interpolated values as a function of an index. However, we wanted to keep a correspondence with the absolute values of the UI (ie user interface) and for this reason, it was necessary to implements the feature in Objective-C.

Below, an example from the Decay Knob of the Envelope module, please observe the discontinuity values of this parameter:

```
/* Real VCS3 Values for Envelope Decay */
[_Decay setTableCurve:0.007 forIndex:0];
[_Decay setTableCurve:0.010 forIndex:1];
[_Decay setTableCurve:0.028 forIndex:2];
[_Decay setTableCurve:0.116 forIndex:3];
[_Decay setTableCurve:0.425 forIndex:4];
[_Decay setTableCurve:1.600 forIndex:5];
[_Decay setTableCurve:4.400 forIndex:6];
[_Decay setTableCurve:7.500 forIndex:7];
[_Decay setTableCurve:9.500 forIndex:8];
[_Decay setTableCurve:11.000 forIndex:9];
[_Decay setTableCurve:15.00000 forIndex:10];
```

Therefore, the function for the interpolation calculation:

```
-(float)valueFromIndex:(float)phi {

    //Linear Interpolation (two points)
    short index = (short) phi;
    float dif = phi - (float) index;
    float sample1 = _tableCurve[index];
    float sample2 = _tableCurve[index + 1];
    float RESULT = sample1 + (sample2 - sample1) * dif;

    return RESULT;
}
```

This approach provides an additional level of optimization, in fact the interpolations are calculated as a result of a user action on the Knob. The mechanism is known as 'event-driven programming', a delegate-function (callback) is called only if necessary, unlike *Csound* that continuously performs a *pull* over the control (i.e. 'k') variables.

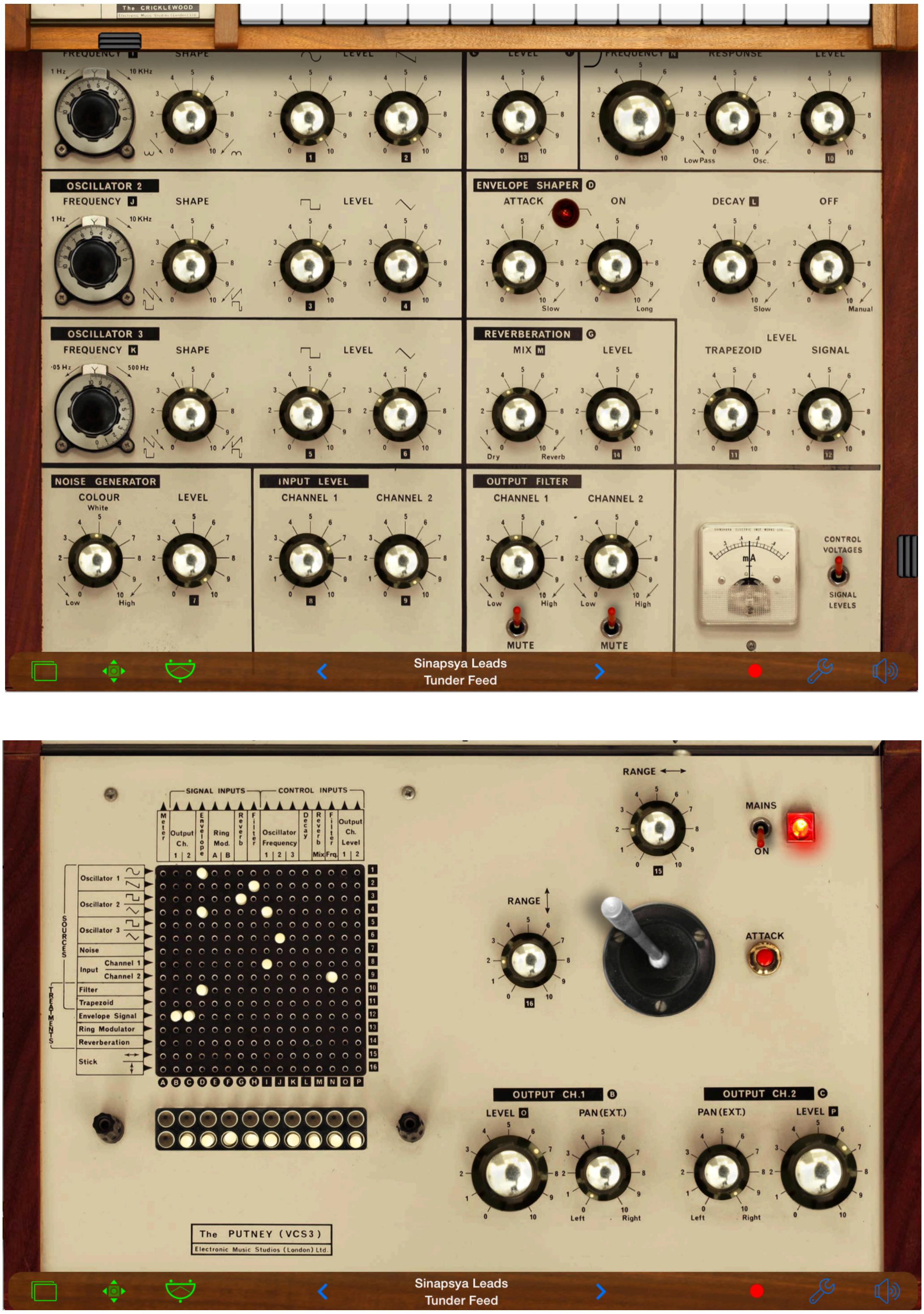

Figura 14. Two screenshots that show all the controls on the front panel of the iVCS3.

XI. Conclusions

The experience of emulate a vintage synth using the standard Csound opcodes (and the good performance in terms of number of the app downloads) shows the power of Csound as an incredible sound development tool not only for experimental and didactical purpose but also for semi-professional and professional use.
This experience also showed again that the success of the emulation of a music machine depends in large part from the good synergy between the various modules and control parameters, even greater extent the exact reconstruction of each individual component. In addition, the programming style of Csound accelerates the adaptation of the various modules to the more general context that characterizes the environment and development tools for iOS system applications.

XII. Aknowledgements

Many thanks to LEMS (Conservatory of Music G. Rossini - Pesaro) to let us to analyze and study in detail the behavior of a real VCS3 (1969).
Special thanks to Peter Zinovieff (the inventor of the original VCS3) for his appreciation of our work and its valuable suggestions.
A sincere thanks to Stefano Zambon for sharing with us the guidelines of his work and experiences on the digital emulation of the VCS3 filter. A special thanks to Josue Arias for the invaluable help in the envelope programming and for his deeply knowledge of the EMS Synthi.
www.kimatika.com

References

[1] Tim Stilson, Julius O.Smith "Alias-Free Digital Synthesis of Classic Analog Waveforms" CCRMA [Online] Available: https://ccrma.stanford.edu/~stilti/papers/blit.pdf
[2] Gary Scavone "Bandlimited Synthesis" MUM 307 – (2004-2015) McGill Univerity [Online] Available: http://www.music.mcgill.ca/~gary/307/week5/bandlimited.html
[3] Richard Hoffmann-Burchardi "Asymmetries Make The Difference: An Analysis Of Transistor-Based Analog Ring Modulators" Proc. of the 12th Int. Conference on Digital Audio Effects (DAFx-09), Como, Italy, September 1-4, 2009 - [Online] Available: http://dafx09.como.polimi.it/proceedings/papers/paper_24.pdf
[4] Antti Huovilainen "Non-Linear Digital Implementation Of The Moog Ladder Filter" " Proc. of the 12th Int. Conference on Digital Audio Effects (DAFx-04), Naples, Italy, October 5-8, 2004 [Online] Available: http://www.mirlab.org/conference_papers/International_Conference/DAFx%202004/Proc/P_061.pdf
[5] Julian Parker "Dispersive Systems in Musical Audio Signal Processing " Doctoral Dissertation 158/2013 – Aalto University [Online] Available: http://lib.tkk.fi/Diss/2013/isbn9789526053684/isbn9789526053684.pdf
[6] EMS VCS3 / Synthi A Emulator by Steven Cook - stevencook@appleonline.net

Additional Resources

VCS3 Users Manual – EMS London
The Canonical Csound Reference Manual – B. Vercoe et altri - MIT
http://karim.barkati.online.fr/cours/supports/csound/csound5_manual.pdf
R. Boulanger "The Csound Book" – MIT Press
R. Bianchini – A. Cipriani " Il suono virtuale" – Ed. ContempoNet
E. Giordani "Stria 2.70" – from CsoundQt official synth category examples
S. Zambon – F. Fontana "Efficient Polynomial Implementation Of The EMS VCS3 Filter Model" Proc. of the 14th Int. Conference on Digital Audio Effects (DAFx-04), Paris, France, September 19-23, 2011 [Online] Available: http://recherche.ircam.fr/pub/dafx11/Papers/99_e.pdf
V. Lazzarini's implementation of *moogladder* UDO
http://www.csounds.com/udo/cache/Moogladder.udo

The Csound resources for this text: www.alessandro-petrolati.it/cs/icsc_2015.zip