

Csound - Max/MSP : Implementing a framework for Serial Composition and Frequency Modulation in both environments

Reza Payami
reza_payami@yahoo.com

Introduction

This article explains how a certain problem can be solved in two different environments, e.g. Csound and Max/MSP, through providing a framework for serial composition and frequency modulation as a sample. It can be used as a basis for a workshop to explain and compare how to design and develop ideas in Csound and Max/MSP. So, different approaches for each environment to a certain problem can be compared with each other. Some user defined opcodes and instruments in Csound, as well as their counterpart patches in Max/MSP have been implemented and will be described throughout.

I. Problem Definition and Design Overview

As mentioned, the goal of this paper is to explain how to design and implement a specific problem in Csound and Max/MSP, regardless of suggesting serialism and frequency modulation as a composition and sound synthesis technique. Providing serialism together with frequency modulation is only a mean to compare these two computer music environments.

A composed piece based on this framework, can have a polyphonic texture by using multiple voices, each of which using series or tone row to specify the values for musical parameters, e.g. pitch, velocity and timbre. Moreover, frequency modulation is the tool used for synthesis of different timbres, having different harmonicities [4,5] as an FM parameter.

Based on what serialism would suggest, there should be a mapping and translation from each tone row element to the certain value of each musical parameter, e.g. pitch, velocity or timbre [2]. Having such an analogy, for instance, number 2 may different meanings considering each musical parameter, e.g. pitch 50, velocity 60 and harmocity 10. In other words, there is a formula for converting each tone row element to the value for a musical parameter. The `NextRowElement` opcode facilitates the general usage of a tone row, while `NextPitch`, `NextVelocity` and `NextHarmonicity` opcodes specialize `NextRowElement` considering each corresponding musical parameter. Calling these three opcodes, the next note for each voice can be obtained. `RestartRow`, and the specialized `RestartPitch`, `RestarVelocity` and `RestartHarmonicity` opcodes, are developed to provide the ability for initiating the traversal mechanism over a tone row. This tone row traversal or transformation mechanism may be of different types including original, inverse, retrograde or inverse-retorgade, together with a transposition,

The score structure pattern, Csound instruments and opcodes are explained in a top-down style according to what follows. The similar Max/MSP patches are also shown next to the Csound code.

II. The Score Structure Pattern

In the score, there may be two possibilities, the first one is to restart or initiate a tone row traversal mechanism for generating the next notes played by an instrument, and the other is to command the instrument to play the next note, using a specified attack and release duration. The former case is specified by passing duration, or p3 argument, as '-1', while the latter will use this parameter as the duration of the next note. In other words, whenever there is a '-1' value in p3 in the score, then p4, p5 and p6 will be used as transformation commands for restarting the row traversal to generate the next notes for the corresponding instrument.

The transformation command holds two parts, one for transformation type, and the other for transposition. The transformation types are defined as 0 (e.g. P or O) for using the row as original, 1 (I) as the inverse, 2 (R) for retrograde, and 3 (IR or inverse-retrograde). The succeeding number specifies the transposition value. For example, "2 3" means using retrograde and transposing the row elements by 3 semitones higher.

In case of having a positive duration, an event is sent to an instrument to play its next note. The attack and release parameters (p4 and p5) are regarded as the percentage of the whole duration. For example a value of 0.2 for attack means 20% of the whole note duration, causing the amplitude going from 0 to the value calculated based on the velocity of the note, which in turn is specified by the row and its transformation. For Max/MSP counterpart patch for developing the score, the "timeline" object in version 4.5 has been used.

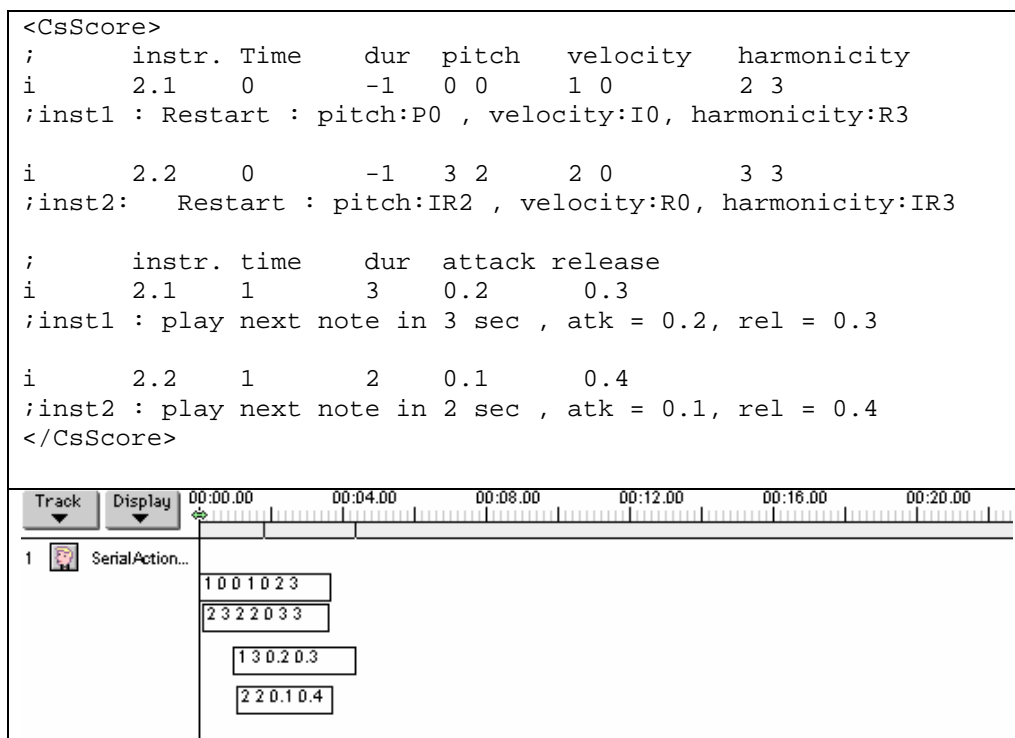


Figure 1 – Csound score / Max/MSP timeline

So, the start time, duration, and volume envelope parameters for each note are specified in the score where different voices are played at the same time. The Row, Reference Pitch and Harmonicity values, can be specified for each score differently,. They may also change during a single piece

III. Instrument 1 : Frequency Modulation

The first instrument uses frequency modulation and the FM opcode, working as the sound generation engine. This opcode implements Chowning`s idea [4,5] for having harmonicity and modulation index for defining timbre. Among the opcode input parameters, p4 and p5 specify pitch and velocity of the note, while p6 and p7 define harmonicity and modulation index. The former parameter, or p6, affects the timber and would result in a harmonic or inharmonic sound, based on its value. The latter parameter has a value between 0 and 1, and the higher it is the brighter the generated sound would be. As mentioned previously, p8 and p9 parameters are attack and release duration percent related to the whole note duration.

In this context, as score is the only caller to FM, i-rate variable opcode have been implemented and used. For extending this opcode to be reusable in other situations, a k-rate opcode can be implemented as well.

<pre>opcode FM, a, iaai iCarrFreq, aCarrAmp, iHarmonicityRatio, iModulationIndex xin iModFreq = iCarrFreq * iHarmonicityRatio iModAmp = iModFreq * iModulationIndex aModulator poscil iModAmp, iModFreq, 2 aCarrier poscil aCarrAmp, iCarrFreq + aModulator, 2 xout aCarrier endop</pre>	
--	--

Figure 2 – FM opcode/patch

<pre>instr 1 iKey = p4 iVelocity = p5 iHarmonicity = p6 iModIndex = p7 iAttackPercent = p8 iReleasePercent = p9 iFrequency = cpsmidinn(iKey)</pre>

```

iAmplitude = iVelocity / 127
aEnv linseg 0, p3 * iAttackPercent, iAmplitude, p3 * (1 -
iAttackPercent - iReleasePercent), iAmplitude, p3 * iReleasePercent, 0

aCarrier FM iFrequency, iAmplitude * aEnv, iHarmonicity, iModIndex

outs aCarrier, aCarrier
endin

```

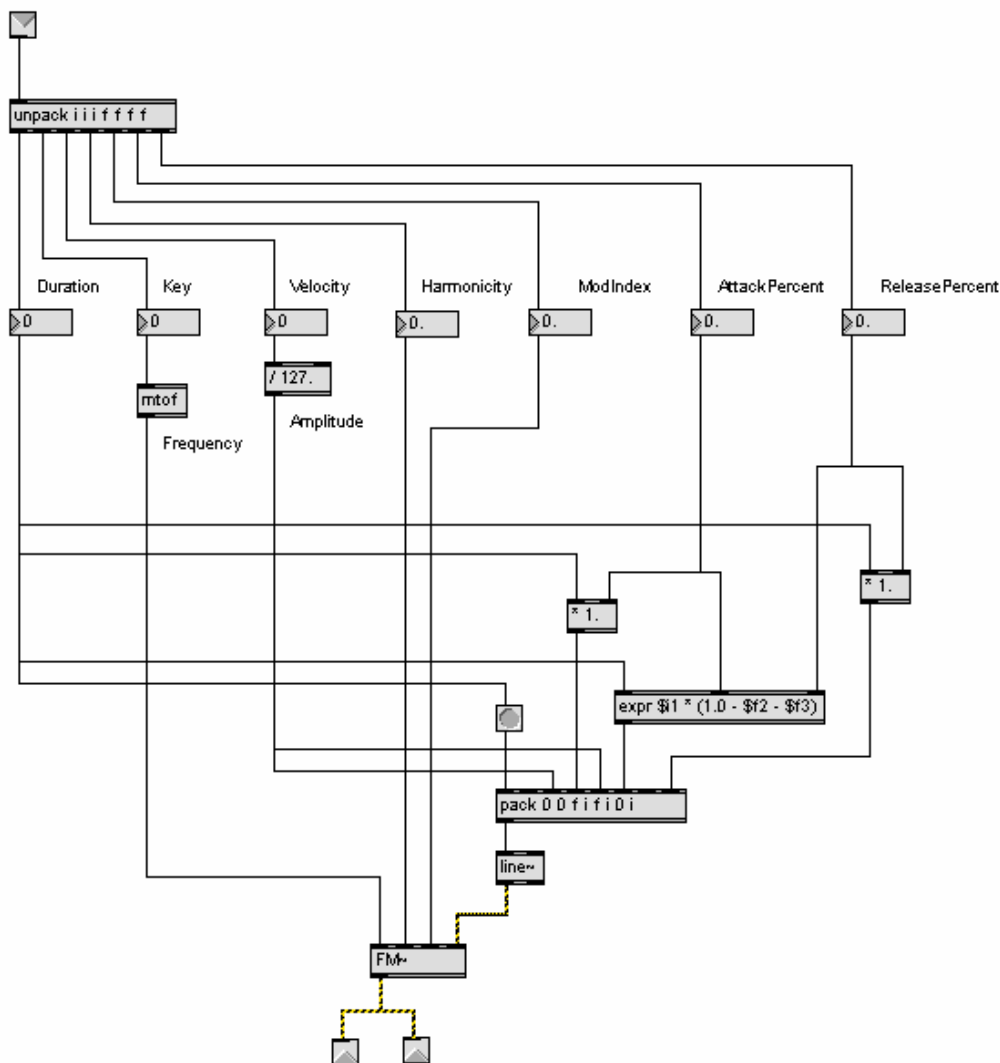


Figure 3 – instr1

IV. Instrument 2 : Applying Serialism

The second instrument, as described before, acts differently based on the duration value. A '-1' duration value would lead to a row traversal restart, while the positive values result in playing the next note for a voice.

Through a polyphonic texture, there are different (4) voices which can traverse the rows differently at the same time. Assuming that the instrument number is 2, the voice number can be specified by

using “p1 * 10- 20” expression. For example “2.3” would lead to having the value 3 as iVoice and “2.4” will specify the 4th voice.

When a ‘-1’ value in the duration occurs, the traversal of all three aspects are initiated based on the transformation command parameters. So, RestartPitch, RestartVelocity and RestartHarmonicity is called. In order to hold two transformation parts, e.g. transformation type and transposition in a single integer, a formula has been used, by multiplying transposition type by 100 and then adding the result with transposition value.

In the other case, the next note for the specified note is generated and played using instrument1. So, calling NextPitch, NextVelocity and NextHarmonicity for the corresponding voice, the next note parameters are calculated and the next note will be played using frequency modulation.

The implementation of the mentioned Opcodes and the translated Max/MSP patch will be described and shown in the following sections.

```
instr 2

  iPitch init 0
  iVelocity init 0
  iHarmonicity init 0
  iVoice init p1 * 10 - 20

  if p3 == -1 then
    RestartPitch iVoice, p4 * 100 + p5
    RestartVelocity iVoice, p6 * 100 + p7
    RestartHarmonicity iVoice, p8 * 100 + p9
  else
    iAttackPercent = p4
    iReleasePercent = p5

    iPitch, iPitchIndex NextPitch iVoice
    iVelocity, iVelocityIndex NextVelocity iVoice
    iHarmonicity, iHarmonicityIndex NextHarmonicity iVoice

    if iPitch != -1 then
      event_i "i", 1, 0, p3, iPitch , iVelocity, iHarmonicity, 0.5,
      iAttackPercent, iReleasePercent
    endif

  endif

endin
```

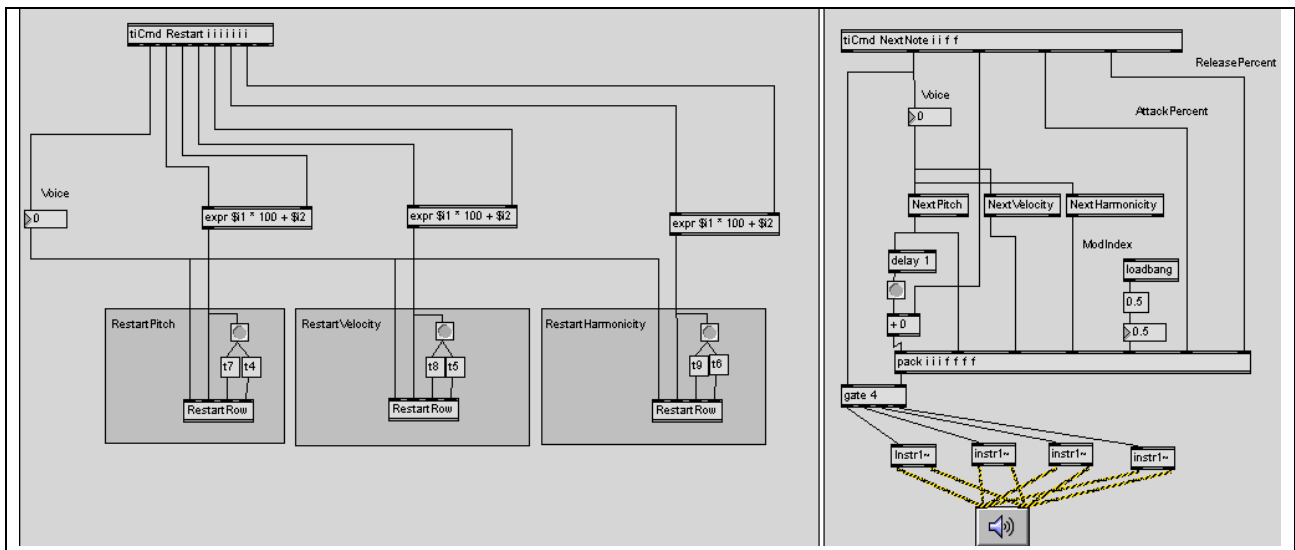


Figure 4 – instr2

Data Structure : Global Tables

The values for configuring the scores are defined in some global variables, including `giRow` which represents a row, `giSineWave` as sine wave for FM, `giHarmonicities` and `giReferencePitch`.

```

giRow ftgen 1, 0, -12, -2, 0, 1, 6, 7, 2, 5, 4, 3, 11, 9, 8, 10
giSineWave ftgen 2, 0, 1024, 10, 1
giHarmonicities ftgen 3, 0, -12, -2, 1, 10.3, 43.7, 60.7, 120.6, 230.5, 340.3,
450.4, 562.2, 673.43, 789.43, 1004.345

```

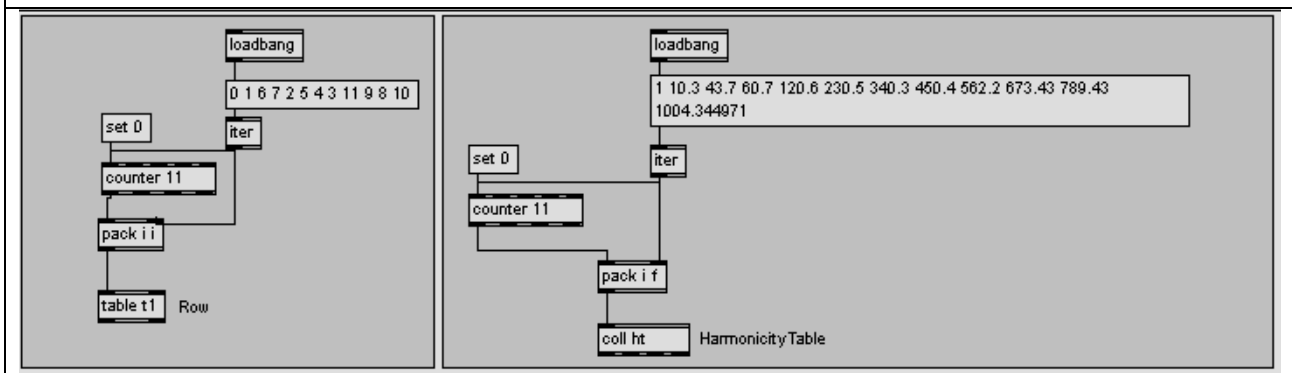


Figure 5 – global tables

As there may be multiple (assumed up to '4') voices in the score, a data structure for each voice is required to store index values for pitch, velocity and harmony. Moreover, the current transformation value (e.g. R3) for each voice, set by the last called Restart opcode should also be stored somewhere for each voice. This can be implemented by using different tables, with their size equal to the number of voices. According to figure 6, table 4, 5, 6 hold the index values, while table

7, 8, 9 keep the transformation values. Each element of a table represents the related value (e.g. pitch transformation for table 7) for a specific voice.


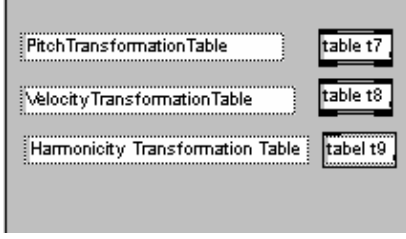
<pre>giPitchIndex ftgen 4, 0, -4, -2, 0, 0, 0, 0 giVelocityIndex ftgen 5, 0, -4, -2, 0, 0, 0, 0 giHarmonicityIndex ftgen 6, 0, -4, -2, 0, 0, 0, 0</pre>	
<pre>giPitchTransformation ftgen 7, 0, -4, -2, 0.0, 0.0, 0.0, 0.0 giVelocityTransformation ftgen 8, 0, -4, -2, 0.0, 0.0, 0.0, 0.0 giHarmonicityTransformation ftgen 9, 0, -4, -2, 0.0, 0.0, 0.0, 0.0</pre>	

Figure 6 – tables for each voice

According to what mentioned so far, in order to store two different parts of transformation value, e.g. transformation type and transposition (for instance R4 has two parts : ‘R’ as transformation type and ‘4’ as transposition), as a single integer, an encoding is required. We can simply assign P with 0, I with 1, R with 2 and IR, with 3, multiply it with 100 and add it with transposition value. So, for example R4 would be converted to $2*100 + 4$ or 104.

Therefore, in order to manipulate the values for each voice `RestartPitch`, `RestartVelocity` and `RestartHarmonicity` opcodes are implemented. They only hide the fact that which table number is related to which musical parameter, providing a higher level opcode to be called in instrument 2. In Max/MSP there would be no need to implement an independent patch due to its simplicity and they are only a part of the main Action patch.

```
opcode RestartPitch ,0, ii
  iVoice, iPitchTransformation xin
  RestartRow iVoice, iPitchTransformation, 7, 4
endop

opcode RestartVelocity ,0, ii
  iVoice, iVelocityTransformation, xin
  RestartRow iVoice, iVelocityTransformation, 8, 5
endop

opcode RestartHarmonicity ,0, ii
  iVoice, iHarmonicityTransformation xin
  RestartRow iVoice, iHarmonicityTransformation, 9, 6
endop
```

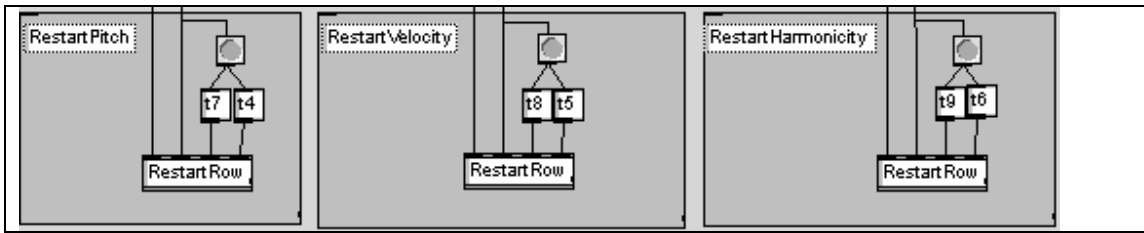


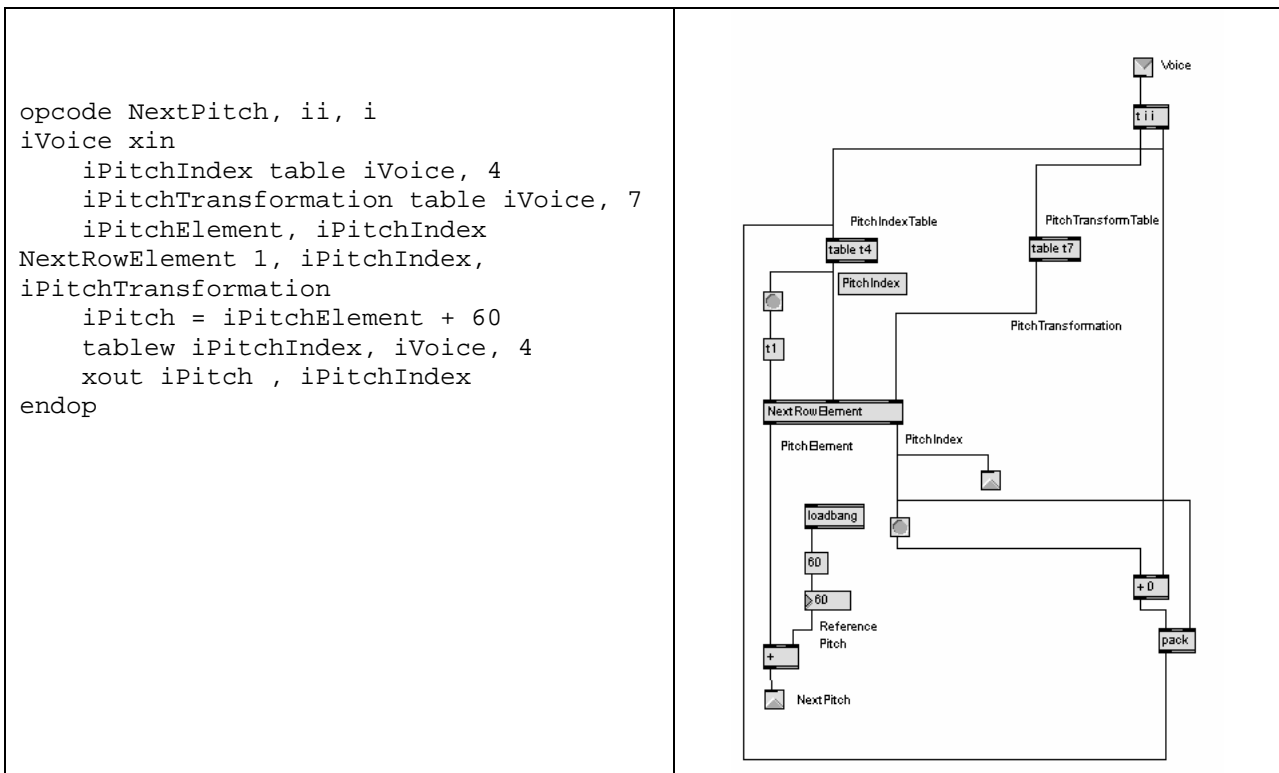
Figure 7 – RetartPitch, RestartVelocity and RestartHarmonicity opcodes / patches

NextPitch, NextVelocity and NextHarmonicity opcodes, also contain the required mapping for analogy in serialism.

- For pitch we have: $iPitch = iPitchElement + giReferencePitch$

- For velocity: $iVelocity = iVelocityElement * 3 + 80$

- And for harmonicity: $iHarmonicity$ table $iHarmonicityElement, 3$



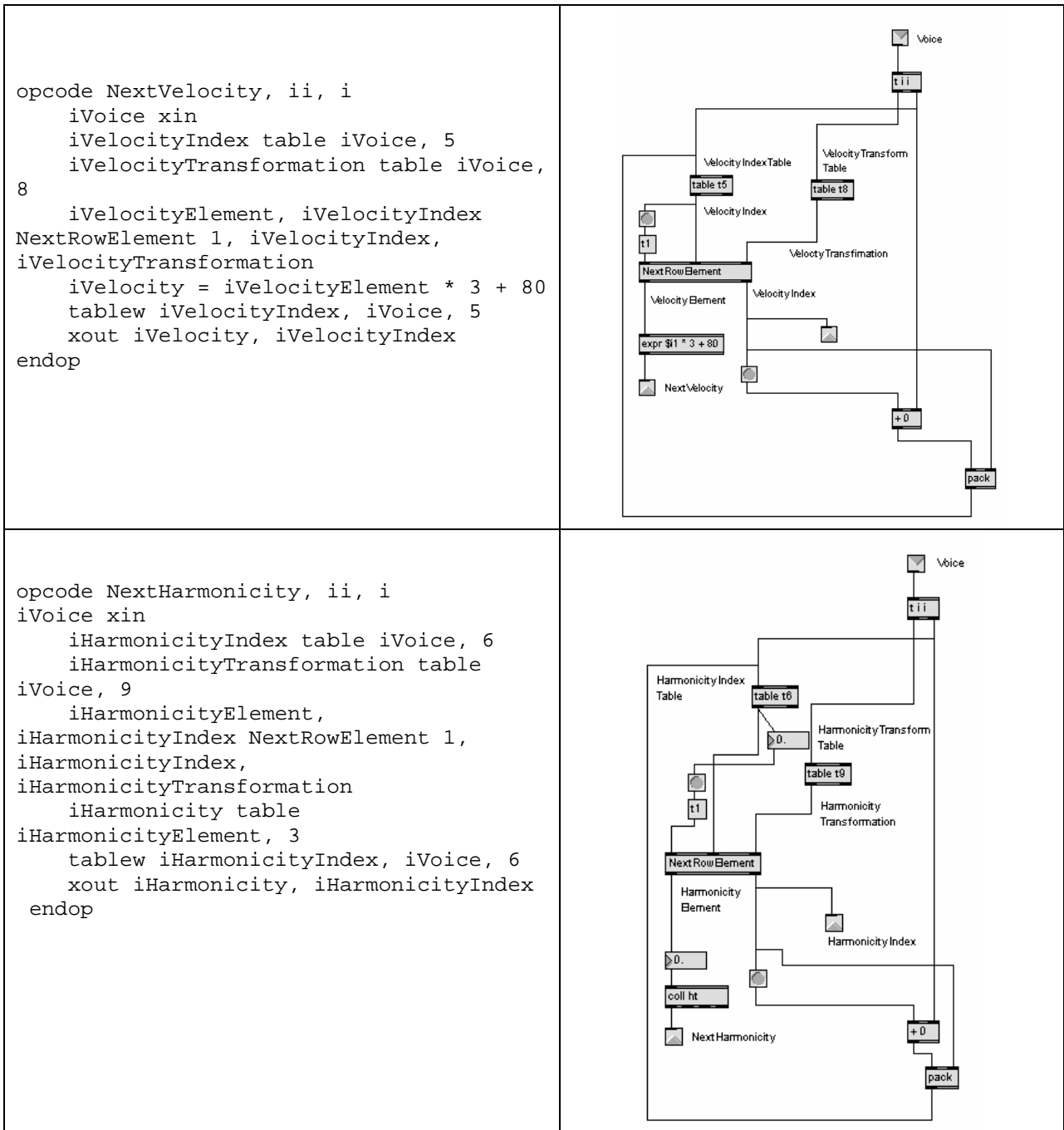


Figure 8 – NextPitch, NextVelocity and NextHarmonicity opcodes / patches

RestartRow Opcode

RestartRow opcode initiates the row traversal mechanism for a voice and a musical parameter (for example its sets R4 as the traversal method for 2nd voice). The transformation encoding takes place in this opcode, and it then stores this transformation integer number in the related table, using iTransformTable parameter. iIndexTable also specifies the table number for the related musical parameter (for instance 4 is considered as PitchIndexTable).

```
opcode RestartRow ,0, iiii
iVoice, iTransformation, iTransformTable, iIndexTable xin

tablew iTransformation, iVoice, iTransformTable
iTransformationType = int(iTransformation / 100)

if iTransformationType == 0 || iTransformationType == 1 then;P or I
  tablew 0, iVoice, iIndexTable
elseif iTransformationType == 2 || iTransformationType == 3 then; R
or IR
  tablew 11, iVoice, iIndexTable
endif
endop
```

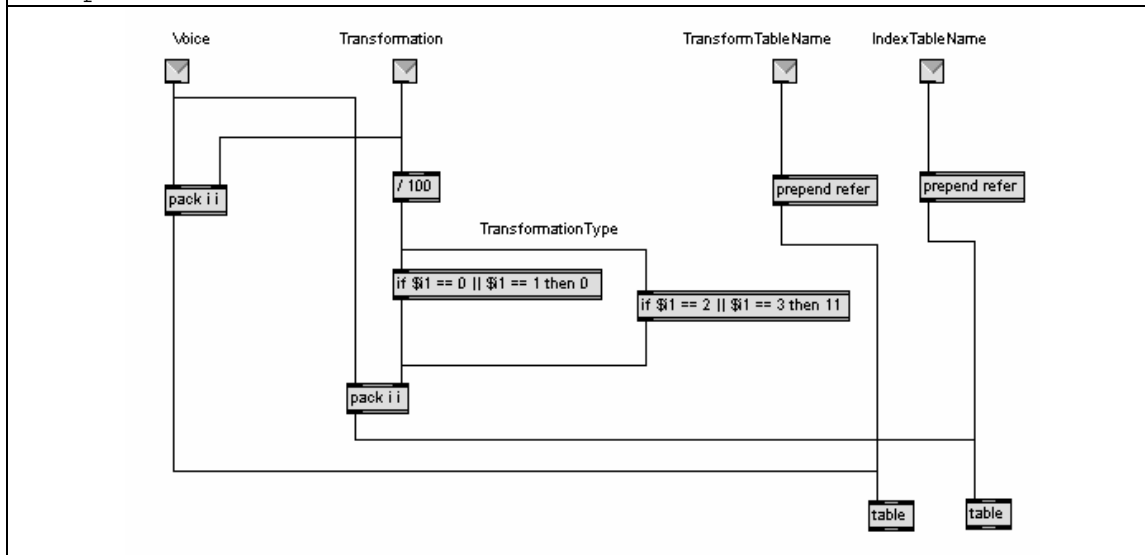


Figure 9 – RestartRow opcode / patch

NextRowElement Opcode

NextRowElement opcode manipulates the index tables, using the last transformation command called for each musical parameter and voice. It also rotates the index, whenever it reaches to the end.

```

opcode NextRowElement, ii, iii
iRowNumber, iIndex, iTransformation xin
; iTransformType : P=0 I=1 R=2 IR(X) =3
; R4 : iTransformType=2, iTransposition=4
iTransformType = iTransformation / 100
iTransposition = iTransformation % 100

; Set nextRowElementvalue, -1 if out of range
if iIndex >= 0 && iIndex <= 11 then
    iNextRowElement table iIndex, iRowNumber
else
    iNextRowElement = -1
endif

; Advance index, rotate if it has reached to the end range
if iTransformType == 0 || iTransformType == 1 then;P or I
    if iIndex >= 11 then
        iIndex = 0 ; Rotate
    else
        iIndex = iIndex + 1
    endif
endif

if iTransformType == 2 || iTransformType == 3 then;R or RI
    if iIndex <= 0 then
        iIndex = 11 ; Rotate
    else
        iIndex = iIndex - 1
    endif
endif

xout (iNextRowElement + iTransposition) % 12, iIndex
endop

```

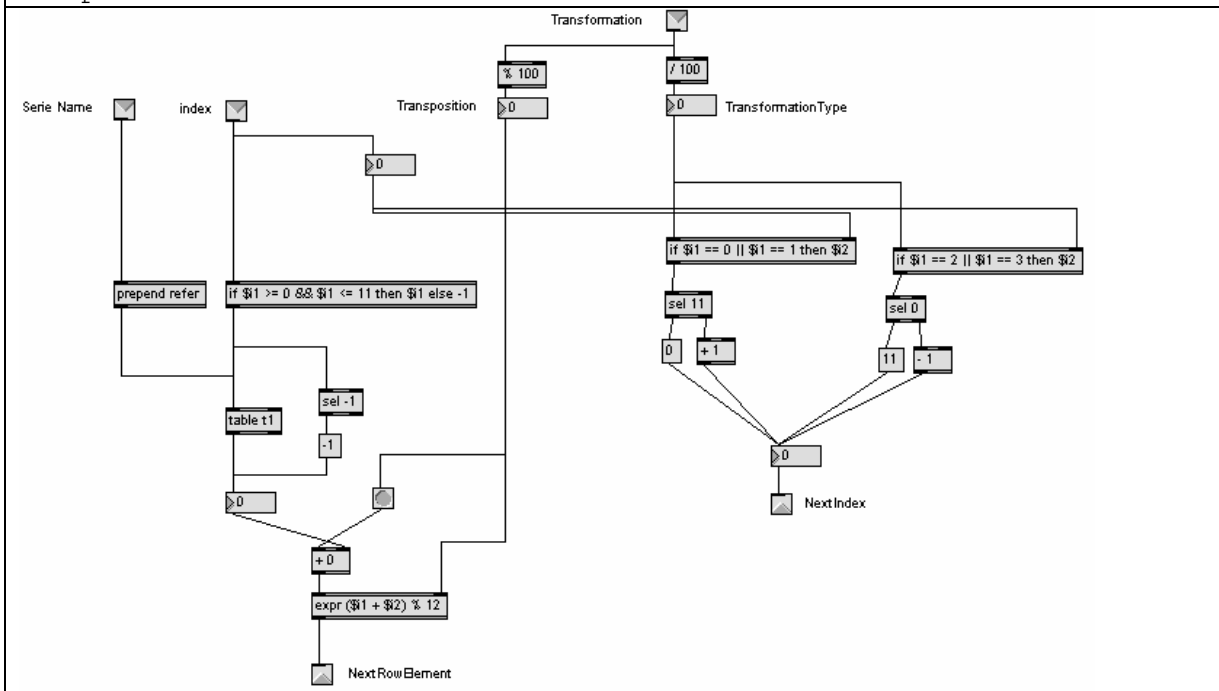


Figure 10 – NextRowElement opcode / patch

Conclusion

Comparing Csound and Max/MSP is not an easy issue while both of them are powerful environments in computer music domain. Having proper experience, many complicated problems can be implemented with each of them easily. But there are some factors which can be pointed out regarding these two alternatives.

Max/MSP provides a graphical user interface based on block diagrams as building blocks to design and implement some patches. This may be the easiest method for modeling situations dealt with connections and wirings. For example connecting some audio units and components for audio manipulation or live electronics is easily modeled using Max/Msp. Moreover, as Max/MSP patches are shown in diagrams, they can reflect the design behind of a problem more easily than some lines of code. In addition, providing a graphical user interface is easy and straightforward, as Max/MSP components are user interface components themselves and do not need another presentation layer (like FLTK or Widgets in Csound). But sometimes combining logic and user interface may cause some drawbacks and may be misleading. It is a better idea to separate logic and user interface in different layers. The “Presentation View” in Max/MSP 5.0 is a method for separating user interface and the underlying implementation logic.

On the other hand, Csound is a textual environment to facilitate a computer music programmer to develop a specific idea. Using Csound, the developer does not face the problems of drawing, resizing etc, and is able to implement the logic by means of some lines of code, like what is offered by the programming languages. A value can be simply stored in a variable and used later many times in the code (while in Max/MSP different wires should be used to pass the value of a component to other components). But, for a musician who is unfamiliar with the programming concepts, it takes more time to learn the Csound approach, comparing to a more user-friendly graphical environment (such as Max/MSP). In Csound, the presentation and logic layers can be separated completely (while in Max/MSP both aspects are tied together), as the presentation components can be designed independently from logic code (Using e.g. FLTK or Widgets).

Apart from the differences between Max/MSP and Csound, a so-called divide-and-conquer approach would simplify implementing an idea. Dividing a problem into different components, either Csound opcodes or Max/MSP patches, may result in having a proper solution, regardless of the used tools. But the problem is how many divisions should be used to have a proper balance in the size and number of patches or opcodes, and the degree of abstraction one would achieve [6]. All in all, the ability to design properly and being familiar with design patterns and implementation techniques such as divide-and-conquer, is more important than the used environment.

Acknowledgements

References and links

- 1- Csound FLOSS Manual <http://www.flossmanuals.net/csound/>
- 2- Serialism definition <http://en.wikipedia.org/wiki/Serialism>
- 3- Max/MSP Tutorials <http://www.cycling74.com/docs/max5/tutorials/msp-tut/mspindex.html>
- 4- Frequency Modulation in Csound http://booki.flossmanuals.net/csound/_v/1.0/d-frequency-modulation/
- 5- Frequency Modulation in Max/MSP <http://www.cycling74.com/docs/max5/tutorials/msp-tut/mspchapter11.html>
- 6- Csound FLOSS Manual , User Defined Opcodes - “Is There An Optimal Design For A User Defined Opcode?” http://booki.flossmanuals.net/csound/_v/1.0/f-user-defined-ops/