Csound as a Real-time Application

Typical Problems and Solutions for the Use of Csound in a Live Context

Joachim HEINTZ
Incontri - Institute for new music
HMTM Hannover
Emmichplatz 1, 30175 Hannover, Germany
joachim.heintz@hmtm-hannover.de

Abstract

This article discusses the usage of Csound for live electronics. Embedding Csound as an audio engine in Pd is shown as well as working in CsoundQt for live performance. Typical problems and possible solutions are demonstrated as examples. The pros and contras of both environments are discussed in comparison.

Keywords

Csound, Pd, Live Electronics.

1 Introduction

Csound [1] is well known as one of the most powerful and approved audio libraries, but its main programming model stems from a non-real-time approach: 'instruments' are called for specified durations by a 'score'. This is perfectly suited for tape compositions, and there is still a predominant opinion that Csound is good for fixed media compositions but cannot be used in live situations. Or at least, that it is a pain to do so.

Indeed, if the user does not want to hit a dead end when using Csound in a real-time situation, the architecture and event structure of Csound needs to be carefully considered. It is the aim of this contribution to exemplify how Csound can be used successfully for live performance.

As Csound has no native user interface, it can be embedded in different hosts. Each host application offers different advantages and its own manner of interfacing with Csound. In the field of free software, the use of Csound in Pd [2], and the use of Csound in CsoundQt [3] are probably the most interesting choices.[1] The examples given here utilize these two hosts. They will describe the real-time transposition of a live input, which would represent a typical real-time transformation.

2 Csound in Pd: Building a polyphonic real-time transposer

Combining Csound's DSP power with Pd's flexibility and interfacing should be a really nice idea, both for Csound and Pd users. Victor Lazzarini's *csoundapi~* external for Pd[2] offers this connection. Strange enough, it is much fewer used than it would be expected to – perhaps due to a certain lack of documentation. The first example uses Csound as a polyphonic transposition machine inside Pd and discusses some typical issues.

2.1 Running the *csoundapi~* object

It is beyond the scope of this article to describe the installation of the *csoundapi~* external in Linux, MS Windows and Apple OSX. Descriptions can be found, for instance, in the Csound Floss Manual.[3]

Once the *csoundapi~* object has been installed, it is available in Pd and refers to a Csound file (*.csd).

---

[1] For commercial software, the use of Csound in MaxMsp is well known and perfectly documentated by Davis Pyon [4]. As any Max object itself can be embedded in Ableton Live via "Max4Live", it is possible to call Csound in Ableton Live, too. This has been popularized by Richard Boulanger and partners as "Csound4Live" [5].

[2] The sources can be found in the "frontends" directory of the csound code [1] or in the git repository at http://csound.git.sourceforge.net/git/gitweb-index.cgi

[3] www.flossmanuals.net/csound/index

Now audio and control data can be sent from Pd to Csound and back.

2.2 Building a four voice transposition instrument

In this example, audio from a microphone is received by Pd and passed to Csound which then creates the four voice transposition which is then sent back to Pd as stereo mix. Csound implements these transpositions by first converting the received audio into a frequencydomain signal and then creating the four pitch shifted signals. These four signals are converted back into the timedomain and panned and mixed to create a stereo output.[4] This stereo signal is finally passed back into Pd which in turn sends it to the speakers.

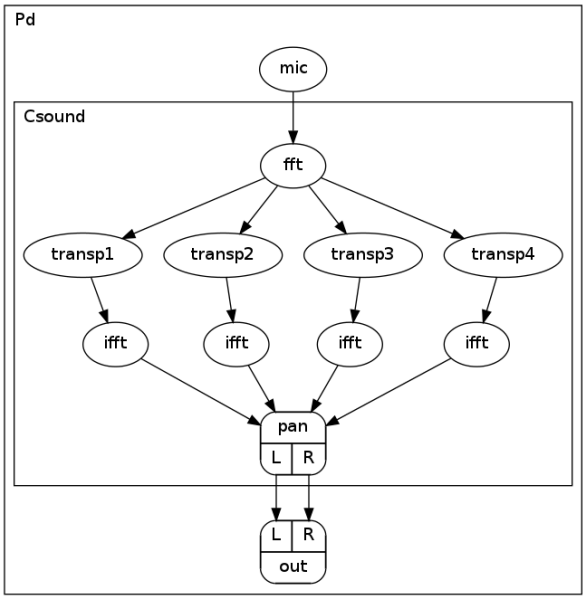

Figure 1: Scheme of embedding Csound in Pd for a four-voice live transposition

The related Csound file is very simple:

```
  <CsoundSynthesizer>
  <CsInstruments>
  sr = 44100
  ksmps = 32
  nchnls = 2
  0dbfs = 1

    instr transp
;input from pd
  aIn       inch      1
;transform to frequency domain
  fIn       pvsanal   aIn, 1024, 256, 1024, 1
;transposition
  fTransp1  pvscale   fIn, cent(-600)
  fTransp2  pvscale   fIn, cent(-100)
  fTransp3  pvscale   fIn, cent(200)
  fTransp4  pvscale   fIn, cent(500)
;back to time domain
  aTransp1  pvsynth   fTransp1
  aTransp2  pvsynth   fTransp2
  aTransp3  pvsynth   fTransp3
  aTransp4  pvsynth   fTransp4
;panning
  aL1, aR1  pan2      aTransp1, .1
  aL2, aR2  pan2      aTransp2, .33
  aL3, aR3  pan2      aTransp3, .67
  aL4, aR4  pan2      aTransp4, .9
  aL        =         (aL1+aL2+aL3+aL4)/3
  aR        =         (aR1+aR2+aR3+aR4)/3
;output to pd
            outs      aL, aR
    endin

  </CsInstruments>
  <CsScore>
  i "transp" 0 99999
  </CsScore>
  </CsoundSynthesizer>
```

The whole Pd patch looks like this:

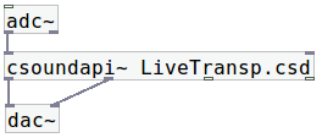

Figure 2: Pd patch for four-voice live transposition embedding Csound via csoundapi~

2.3 Sending and receiving control data

For a fuller interaction between Pd and the embedded Csound it is normally desirable to also pass control signals between Pd and Csound. The transposition values in the given example might have to be controlled from Pd instead of being fixed:

[4] A quadrophonic output redirection into Pd would also be possible.

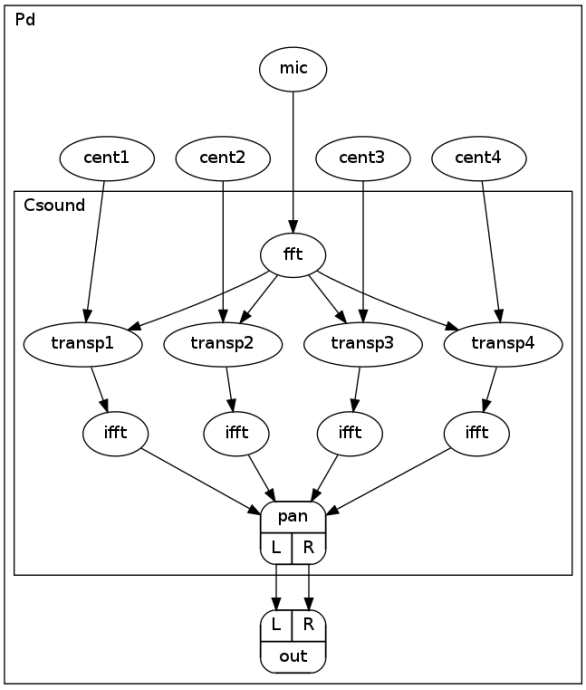

Figure 3: Sending cent values from Pd to Csound

From Pd's point of view, the cent values are a "message" to be sent to Csound. From Csound's point of view, the cent values are "control signals" which are received on certain "control channels".

The *csoundapi~* object understands the message "chnset ...". A Pd message box containing "chnset cent1 -600" means: send the value -600 on the software channel named "cent1".

In Csound, the related message needs to be received by a *chnget* opcode.[5] The instrument code has to be changed slightly now, as follows:

```
  ...
    instr transp
;audio input from pd
  aIn       inch      1
;control input from pd
  kCent1    chnget    "cent1"
  kCent2    chnget    "cent2"
  kCent3    chnget    "cent3"
  kCent4    chnget    "cent4"
;transform to frequency domain
  fIn       pvsanal   aIn, 1024, 256, 1024, 1
;transposition
  fTranspl  pvscale   fIn, cent(kCent1)
  fTransp2  pvscale   fIn, cent(kCent2)
  fTransp3  pvscale   fIn, cent(kCent3)
  fTransp4  pvscale   fIn, cent(kCent4)
  ...
```

And this is the Pd patch, with some choices to send values:

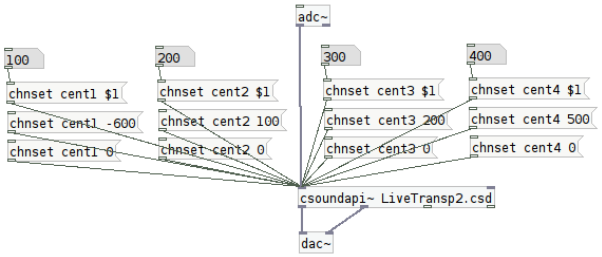

Figure 4: Pd sending cent values to Csound

2.4 The "just once" problem

It is worth to have a closer look at what is actually happening in the example above. It looks easy, and it works, but actually two languages are communicating with each other in a strange way. In this case, they understand each other, but this is pure chance if the differences have not been understood by the user.

Pd differentiates between messages and audio streams.[6] Messages are sent just once, while audio streams are sending data continuously, in the sample rate. Csound has no type for sending something in realtime "just once". If something is sent as a control value to Csound, like the cent values in the example above, Csound considers this as a control signal, i.e. a continuous stream of control data.[7]

In this case, it works. Pd sends a new cent message just once, and Csound interpretes this as a repeated control message. It works, because the *pvscale* opcode needs a control-rate input. But if the user wants to trigger something "just once", they

[5] The alternative choice is to send messages as "control ..." from Pd to Csound, and receive them with the *invalue* opcode in Csound.

[6] This is the same in MaxMsp, and is the well known difference between an object with a tilde and an object without a tilde. For instance, the object '*' multiplies two numbers: just once, when the left inlet has received a number as message. The object '*~' multiplies two audio streams (or an audio stream and a number) continuously.

[7] This stream of control data has a lower rate than the audio rate, depending on the *ksmps* value. If ksmps=32, one control sample is used for 32 audio samples.

have to ensure that Csound does not repeat the message from Pd all the time.

As a simple example, the user may want to trigger a short beep each time a message box with a '1' has been pressed:

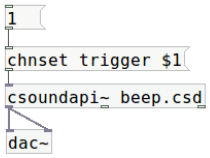

Figure 5: Sending a '1' message from Pd to Csound

It looks straightforward to code the beep.csd in this way:

```
<CsoundSynthesizer>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 1
0dbfs = 1

  instr master ;wrong!
kTrigger  chnget    "trigger"
 if kTrigger == 1 then
          event     "i", "beep", 0, 1
 endif
  endin

  instr beep
aBeep     oscils    .2, 400, 0
aEnv      transeg   1, p3, -6, 0
          out       aBeep * aEnv
  endin

</CsInstruments>
<CsScore>
i "master" 0 99999
</CsScore>
</CsoundSynthesizer>
```

A master instrument would receive the data on the channel called 'trigger' from Pd, and call a subinstrument "beep" each time a '1' has been received. That's the wish of the user. But what really happens: the *kTrigger* variable will always be set to '1', because there is no other message coming from Pd, and so Csound will trigger a beep not "just once", but once at each control cycle, in this case 44100/32 times per second ...

Two things have to be done here: Pd must send a '0', if the message box has not been pressed. The following patch sends, when hitting the 'bang' first a '1' and then after 10 milliseconds a '0':[8]

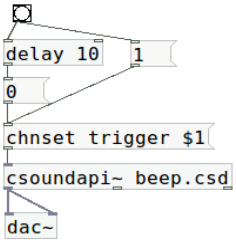

Figure 6: Pd sending '1' followed by '0' to Csound

On the Csound site, just the first '1' received after any zeros should trigger a beep. So Csound has to ignore the repetitions. This is done by the following code in the master instrument:

```
  instr master ;now correct
kTrigger  chnget    "trigger"
kNewVal   changed   kTrigger
 if kTrigger == 1 && kNewVal == 1 then
          event     "i", "beep", 0, 1
 endif
  endin
```

This opcode returns '1' exactly in the control cycle when a new value of *kTrigger* has been received. So the if-clause has to ask whether both, *kNewVal* and *kTrigger* equal 1. Only in this case the subinstrument is to be triggered.

2.5 Results

The use of Csound for realtime applications in Pd can be considered as easy and straightforward in general. The major difficulties come from the the different signal paradigms in both languages. Special care has to be taken by the user for "just once" cases, which are common in Pd but need cautious coding in Csound.

[8] Using a trigger in Pd which sends a '1' from the right output, followed by a '0' from the left output will not work, because Csound will just receive the '0'.

3 Presets, software channels and more: An extended live instrument in CsoundQt

CsoundQt has been written and developed by Andrés Cabrera since 2008.[9] It is now the most widely used frontend for Csound. It uses the Qt toolkit[10] for building a graphical user interface.

CsoundQt offers all necessary tools to work with live electronics in Csound, without using another host application. But how can Csound be trimmed to support presets which are necessary for each extended live electronic context? An example is given which uses a midi keyboard as instrument for controlling both some real-time processings of a microphone input, and electronic sounds generated in real time.

3.1 Presets

A preset is a general configuration in a live electronic application which defines the meaning of certain input signals in the context of this preset environment. For instance, if the midi key number 60 has been pressed in the context of preset 1, it may mean "open the live input with a fade in". If the same key has been pressed in preset 2, it may mean something totally different, for instance "play a percussive sound".

As CsoundQt has native widgets, these widgets can be used to represent a preset.[11] Csound will then look at the value of this widget, and will decide what to do at a certain midi input. This is the general scheme:

As "triggering an event" in Csound usually means "calling an instrument", the Csound code looks like this:

```
   instr midi_receive
;getting the midi note number
  iNotNum    notnum
;getting the actual preset number
```

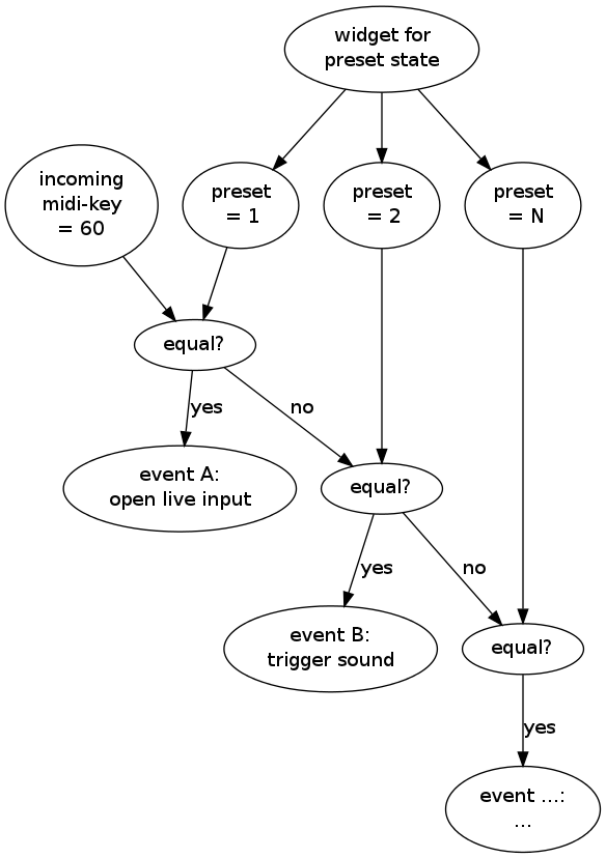

Figure 7: General preset scheme

```
  iPreset    =          i(gkPreset)
;bindings for preset 1
   if iPreset == 1 then
    if iNotNum == 60 then
             event_i "i", "live_fade_in", 0, 1
    elseif iNotNum == ... then
             event_i    ...
    endif
;bindings for preset 2
   elseif iPreset == 2 then
    if iNotNum == 60 then
            event_i "i", "trigger_sound", 0, 1
    elseif iNotNum == ... then
             event_i    ...
    endif
;bindings for preset ...
   elseif ...
   ...
   endif
    endin
```

The *gkPreset* variable which holds the status of the preset given by the preset widget, is created in an "always on" instrument. This instrument should be the first of all the instruments,[12] and its definition should contain a line like this ("preset" is a string which defines the name of the widget's channel):

[9] The name has been changed from QuteCsound to CsoundQt after discussions at the Csound Conference in Hannover in october 2011.

[10] http://qt.nokia.com

[11] CsoundQt also offers a possibility to store widget states as presets. This is very useful in many cases, but not sufficient here, because here the presets do not affect just widgets, but also determine the effect of a Midi key pressed, a parameter being set, and more.

[12] Technically speaking: the instrument with the smallest number.

```
gkPreset  invalue   "preset"
```

By this, the current preset state is received from the spin box widget, and sent as the global variable *gkPreset* to each instrument. The widget itself can be changed by the user either via the GUI or via any input event, for example a reserved midi note for increasing, and another note number for decreasing the preset number. Assuming the reserved midi note for increasing is 72, and for decreasing 48, the code looks like this:[13]

```
if iNotNum == 72 then
          outvalue   "preset", iPreset + 1
elseif iNotNum == 48 then
          outvalue   "preset", iPreset - 1
endif
```

The method proposed here for handling presets offers the user all flexibility. All events are embedded in instruments which are triggered if a midi key is received in the context of a particular preset. General conditions for a new preset can be set in a similar way as shown in chapter 2.4 with the *changed* method. For instance, if the live microphone is to be open at the beginning of preset 1, but closed at the beginning of preset 2, the code in the master instrument could be:

```
gkPreset  invalue   "preset"
kNewPrest changed   gKPreset
 if kNewPrest == 1
  if gkPreset == 1 then
kLiveVol  =         1 ;mic open
  elseif gkPreset == 2 then
kLiveVol  =         0 ;mic closed
  endif
 endif
```

3.2 Software busses for control and audio data

Many situations need a flexible interchange of control data. Suppose the user wants to transpose the live input first in four voices with the cent values -449, -315, 71 and 688, and then in the next bar with the cent values -568, -386, 428 and 498. Both events – activating the four voice transposition and changing the values – are to be triggered by two different midi keys, for instance 60 and 62.

This is a typical case for the use of internal software busses. Four audio signals are set. For each of them a software control bus is created, holding the transposition value. The first instrument sets the initial values. When the second instrument is triggered, the control busses are set to the new values (figure 8).

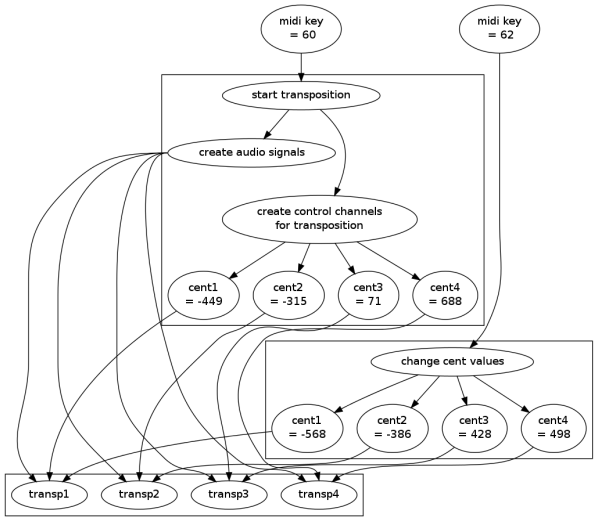

Figure 8: Working with software channels to change transposition values

This is the related Csound code:

```
    instr set_transp
;create software busses
             chn_k      "transp1", 3
             chn_k      "transp2", 3
             chn_k      "transp3", 3
             chn_k      "transp4", 3
;set initial values
             chnset     -449, "transp1"
             chnset     -315 "transp2"
             chnset     71, "transp3"
             chnset     688, "transp4"
;receive the values from the software busses
  kCent1     chnget     "transp1"
  kCent2     chnget     "transp2"
  kCent3     chnget     "transp3"
  kCent4     chnget     "transp4"
  ;receive the live audio input
  aLvIn      chnget     "live_in"
;perform fourier transform
  fLvIn      pvsanal    aLvIn,1024,256,1024,1
;perform four voice transposition
  fTp1       pvscale    fLvIn, cent(kCent1)
  fTp2       pvscale    fLvIn, cent(kCent2)
  fTp3       pvscale    fLvIn, cent(kCent3)
  fTp4       pvscale    fLvIn, cent(kCent4)
;resynthesize
  aTp1       pvsynth    fTp1
```

[13] The instrument is the same as the one above called "midi_receive". This instrument is triggered directly by a midi note on message. It works during Csound's initialization pass, which is in some cases another option for the "just once" problem.

```
  aTp2        pvsynth     fTp2
  aTp3        pvsynth     fTp3
  aTp4        pvsynth     fTp4
;add and apply envelope
  aTp         =           aTp1+aTp2+aTp3+aTp4
  kHul        linsegr     0,.3,1,p3-0.3,1,.5,0
  aOut        =           aTp * kHul
;mix to global audio bus for live out
              chnmix      aOut, "live_out"
    endin

    instr change_transp
              chnset      -568, "transp1"
              chnset      -386, "transp2"
              chnset      428, "transp3"
              chnset      498, "transp4"
    endin
```

As can be seen from this example, the software busses are not just used for control but also for audio signals. The live audio input is sent to a channel called "live_in". The instrument set_transp gets the live input via the line

```
  aLvIn       chnget      "live_in"
```

After processing, the live transposed signals are sent to an audio bus called "live_out". As other instruments may also send audio to this bus, *chnmix* is used instead of *chnset*:

```
              chnmix      aOut, "live_out"
```

Software channels are the solution to many different situations in programming live electronics: mixing, routing, interchanging of values. The *chn* opcodes offer a flexible and reliable system to work with them in Csound.

3.3 Performance tweakings

Csound's performance for live applications depends mainly on its vector and buffer sizes.[14] As mentioned above, the *ksmps* constant defines the internal vector size. A value of ksmps=32 should be adequate for most live situations. The software and hardware buffer sizes must not be kept at Csound's defaults, but should be set to lower values to avoid an audible latency.[15] CsoundQt offers an easy way to adjust them in the configure panel. These are fair values:

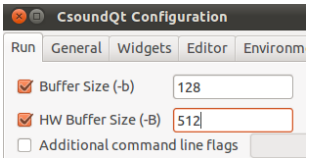

Figure 9: Adjusting the buffer sizes in CsoundQt's configuration panel

In future versions of Csound, the use of multiple cores and threads, will further improve the performance.[16] This would also be adjusted in CsoundQt's configure panel.

In addition to these general Csound adjustments, there are some performance tweaks particularly related to CsoundQt. The python callback, especially, must be disabled for the best live performance:

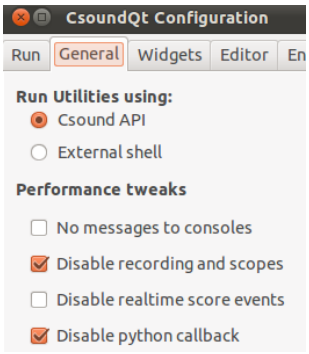

Figure 10: Performance tweaks in CsoundQt

3.4 Results

CsoundQt offers a nice graphical user interface for the use of "pure" Csound with standard widgets for live electronics. Instead of changing between different platforms, the users have an easier

[14] There are some opcodes which are not suited for live in Csound. Usually they have a fast and modern alternative. Especially the old pv (phase vocoder) opcodes should in pratical use be substituted by the excellent pvs (phase vocoder streaming) opcodes: www.csounds.com/manual/html/SpectralRealTime.html

[15] More information can be found at www.csounds.com/manual/html/UsingOptimizing.html

[16] Thanks to the work of John ffitch and others, the new parser for Csound is at the time of writing this article (end of the year 2011) in the process of becoming the default. Amongst other improvements, it will offer multithreading.

workflow than using Csound in Pd. (When making changes to a Csound file running within Pd it is necessary to save the Csound file and to reset the *csoundapi~* object that uses it for those changes to take effect.) Users can add their own functions ("user defined opcodes") or integrate any from the repository,[17] for instance for simplifying the use of the ascii keyboard or for easily recording and playing buffers.

On the user's wish list for using CsoundQt live is certainly an increased array of widgets, for example an audio meter, a midi keyboard widget and an interactive table editor. The possibility of creating more than one widget panel would certainly make CsoundQt much more flexible for use live.[18]

4 Conclusion

Csound can be used perfectly for any live application. The usage of Csound in Pd is very easy. It offers many new possibilities for Pd users, and an easy connection with any external devices (game pads, arduino boards and more) for Csound users.

The usage of CsoundQt for live applications offers a clean and pleasant internal graphical interface and a comfortable workflow to the user. Complex features like presets and software busses can be programmed easily. However, some additional widgets and GUI options might be desireable.

Ultimately, whether a user ought to use Csound live within Pd or purely with CsoundQt as the front-end, will depend on their personal preferences and the situation.

So it depends on the users, their preferences and situations, whether they prefer working with Csound for live in Pd or in CsoundQt.

5 Acknowledgements

Thanks to Anna, Alex, Andrés and Iain for reading the manuscript.

[17] www.csounds.com/udo

[18] Although not covered in this article, it should be
http://www.youtube.com/watch?v=O9WU7DzdUmE
http://www.youtube.com/watch?v=Hs3eO7o349k
http://www.youtube.com/watch?v=yUMzp6556Kw

References

[1] Csound: http://csound.sourceforge.net (with many links to related sites)

[2] Pd: http://puredata.info

[3] CsoundQt: http://qutecsound.sourceforge.net

[4] csound~ (a Max external to run Csound): www.davixology.com/csound~.html

[5] Csound4Live (Csound for use in Ableton Live via csound~ [4] and the Max4Live bridge): www.csoundforlive.com