COMPOSITIONAL SKETCHES IN PWGLSYNTH

Mikael Laurson, Vesa Norilo, and Mika Kuuskankare

CMT,
Sibelius Academy
Helsinki, Finland
laurson@siba.fi
vnorilo@siba.fi
mkuuskan@siba.fi

ABSTRACT

PWGLSynth has already a long history in controlling physics-based instruments. The control system has been score-based, i.e. the user prepares a score in advance, and by interactive listening process the result can be be refined either by adjusting score information, performance rules and/or the visual instrument definition. This scheme allows detailed control on how the instrument model reacts to control information generated from the score. This paper presents a complementary approach to sound synthesis where the idea is to generate algorithmically typically relatively short musical textures. The user can improvise with various compositional ideas, adjust parameters, and listen to the results in real-time either individually or interleaved. This is achieved by utilizing a special code-box scheme that allows any textual Lisp expression to be interfaced to the visual part of the PWGL system.

1. INTRODUCTION

This paper presents the latest developments in our visual synthesis environment PWGLSynth [1] which is situated in PWGL [2]. PWGLSynth has been controlled either by using our score-based scheme, or directly at patch level by using real-time sliders and/or MIDI controllers. Especially the first approach has been used extensively to produce simulations of several physics-based instrument models.

Thus our environment has not been particularly suitable in more experimental contexts, where the user could quickly sketch musical ideas algorithmically. For this end we present in the following a new system that allows to control visual synthesis instrument definitions with the help of short textual code fragments. This is achieved with two additions to our system. First, the code is interfaced to the visual part of PWGL using a special box type, called code-box. The visual code-box can then be triggered at patch level by the user. Typically this results in a musical texture that is played by the system in real-time. Several code-boxes can be triggered simultaneously or interleaved resulting in overlapping realizations of the active code fragments. Second, we present the new syntax addition that allows the user to control low-level synthesis parameters from Lisp. The syntax is used to send triggers, write floats or float vectors to memory locations, and to send MIDI events.

PWGL is one of the three major Lisp-based composition environments along with IRCAM's OpenMusic (OM; [3]) and Common Music (CM; [4]). CM differs from the other environments as it relies on text-based programming and various third party applications. CM can be used for sound synthesis with outside systems such as Csound. OM, like CM, does not contain a dedicated software synthesizer and thus control information has to be exported to external synthesis environments. SuperCollider (SC) [5], although not Lisp-based, is closer in this context to our environment than CM or OM, as it allows to combine sound synthesis with event generation facilities within one environment (events are generated in SC with the help a library of pattern and stream classes). However, the difference between SC and PWGLSynth is that our scheme is not strictly a real-time system as the user can call any Lisp code to calculate the control information. Thus, depending on the complexity of the application, there can be a noticeable delay before a sequence starts playing. If necessary, this delay can be minimized by caching results at the patch level before the actual playback. By relaxing the strict real-time constraint we are able to solve more complex (and hopefully more interesting) musical problems: a typical case is for instance a search problem that may require backtracking (recently real-time constraint systems have been investigated in [6], but these are clearly special cases which can handle only a limited set of problems). Thus our system allows to combine sound synthesis besides with Lisp also with several aspects of the underlying PWGL environment, such as visual instrument definitions, compositional tools, and high-level music representation schemes.

The rest of the paper is organized as follows. We start by presenting the two new tools in our system: code-box (Section 2) and event syntax (Section 3). After this we give several case studies with increasing complexity that demonstrate how the system can be used to realize various compositional situations when preparing musical raw material for sound synthesis.

2. CODE-BOX INTERFACE

This section presents a special PWGL box type, called code-box, which can be seen as an extension to our already elaborate collection of visual boxes. The code-box allows the user to harness the full power of the underlying Lisp language yet preserving the visual nature and functionality of a normal PWGL box. It allows the user to effectively express complex control structures, such as loops, thereby hiding those details from the patch. While the user writes the code in a text editor, the code is simultaneously analysed. This analysis consists of basic syntax checks and extraction of free variables and function names that result in a parameter list of the final Lisp expression. This scheme provides the main inter-

face to PWGL and allows the user to access information from the visual part of the system. The appearance of the box is calculated automatically based on this analysis. By default the code-box is called 'code-box'. This name can be changed by the user. In order to distinguish this box from the ordinary ones, there is a label 'C' at the low-right corner of the box.

For instance, after entering the following Lisp expression the system will detect the free variables 'a' and 'b', and this analysis will automatically create a box with two inputs (one input for each free variable):

```
(iter (for i from 0)
      (for x in a)
      (for y in b)
      (when (oddp i) (collect (list x y))))
```

The code-box has many interesting applications and it is used extensively in our system. The code-box provides an important complement to our visual system and it allows the user to change programming style from visual to textual and vice versa whenever appropriate.

3. INSTRUMENT DEFINITIONS AND EVENT SYNTAX

Next the code-box scheme is used in a sound synthesis context to calculate control information using Lisp code. All code examples are wrapped inside the 'with-synth' macro that will send synth events to the running synthesis patch. 'with-synth' has before the main code entry two arguments: (1) 'instrument' (gives the current instrument definition, a pointer to an abstraction containing the instrument), and (2) 'max-poly' defining the maximum number of voices required by the texture realization.

Typically, the user starts with a visual instrument definition that is situated in a PWGL abstraction. Figure 1 gives a simple resonator example. Here the boxes that have a 'S' label at the down-right corner are ordinary synth modules producing audio. The figure contains also five 'synth-plug' boxes that define control entry points for our instrument. Four of them contain the label 'D' (stands for 'discrete') and they allow to update float values at the leaves of the patch. The synth-plug box with the 'T' designation, in turn, is used to send triggers. All synth-plug contain in their first input a name or label (e.g. ':amp', ':trig', etc.). These symbolic references allow the user to refer to specific inputs while sending control events to the instrument. Note that the names within an abstraction should be unique. Identical names can, however, be used in different instrument abstractions as each abstraction has its own name space.

The synth events are created calling either the 'synth-trigger' or the 'synth-event' method.

'synth-trigger' has two arguments: (1) 'time' (delay in seconds, 0 means immediate), and (2) 'name' (a keyword, must match one of the 'synth-plug' labels of the current instrument).

'synth-event', in turn, has three required arguments: (1) 'time' (delay in seconds, 0 means immediate), (2) 'name' (a keyword, must match one of the 'synth-plug' labels), and (3) 'value' (a float, or a list of floats). Two optional keyword arguments can be given to add an ID number (this is used to distinguish different box instances in a polyphonic situation; this case will be dealt with later in this paper), or to give a type specifier if the event is not a normal one (this can be used for instance to send MIDI events instead of ordinary synth events).

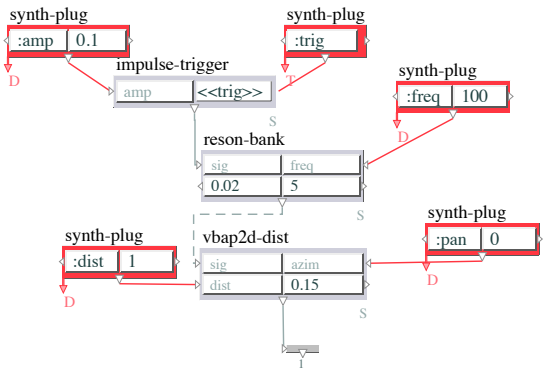

Figure 1: *A visual instrument definition containing an impulse trigger, a resonator, and a spatialization module.*

For example, we could trigger immediately the resonator in the patch in Figure 1 using the following code (we assume that 'ins' is a pointer to the instrument abstraction given in Figure 1):

```
(with-synth ins 1 (synth-trigger 0 :trig))
```

Or, we could trigger the resonator after 2s:

```
(with-synth ins 1 (synth-trigger 2 :trig))
```

We could also change the 'freq' input of the resonator:

```
(with-synth ins 1 (synth-event 0 :freq 200))
```

4. A SIMPLE RESONATOR EXAMPLE

The following Lisp code can be used in conjunction with the instrument patch given in Figure 1 to produce a dense cloud or cluster of notes. The code-box code analyzer will find the following free variables: 'ranges', 'dur', 'dtime', and 'instrument', where 'ranges' gives the pitch limits of the resulting texture, 'dur' gives the global duration of the result, 'dtime' gives the delta-time between individual events, and 'instrument' is the instrument definition. In the actual code we use here the 'iter' loop construct to generate the events. At each iteration step we send three events that control the distance, pan, and frequency parameters (i.e. ':dist', ':pan', ':freq') of our instrument definition. Finally, we also trigger the resonator:

```
(with-synth instrument 1
 (let ((rnd1 (first ranges)) (rnd2 (second ranges))
       (pan (mk-circ-list '(-5 0 5))))
  (iter (for time from 0 to dur by dtime)
    (synth-event time :dist (random2 1 6))
    (synth-event time :pan (pop-circ pan))
    (synth-event time :freq
         (random2 (m->f rnd1) (m->f rnd2)))
    (synth-trigger time :trig))))
```

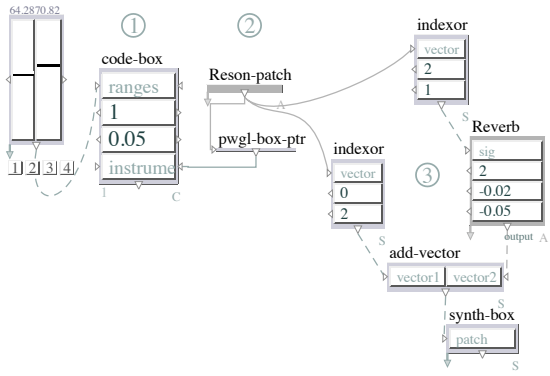

Figure 2: *The top-level patch with the code-box (1), the instrument abstraction (2), and synth boxes (3) defining a global reverb and the final synth output.*

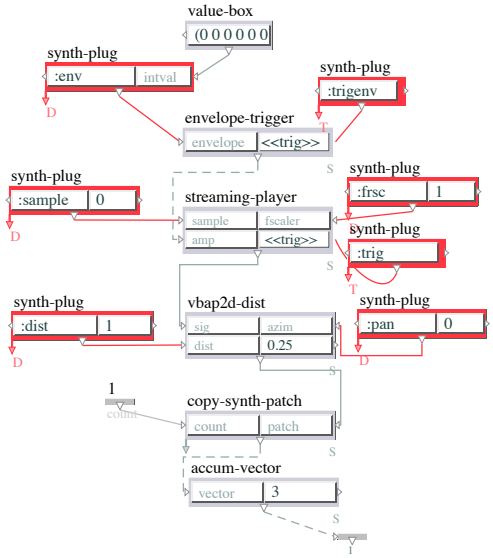

Figure 3: *A polyphonic sampler instrument definition.*

Figure 2 shows the complete top-level patch definition of our resonator example. The code is situated inside the code-box (1) (note that the free variables found in the code are represented visually by the four inputs), the instrument definition–see Figure 1–is inside the 'Reson-patch' abstraction (2). While the synth is running the user can trigger the code-box resulting in a sequence of note events. The user can adjust here in the visual part of our system various aspects of the result. For instance, the two sliders to the left control the 'ranges' parameter.

5. A POLYPHONIC SAMPLER EXAMPLE

Next we discuss a more complex case where a polyphonic streaming sample player is controlled by our event scheme. Figure 3 gives our visual instrument definition. The main difference between this one and the one found in Figure 1 is that we use here the 'copy-synth-patch' box to copy the patch connected to the 'patch' input 'count' times. Thus we will produce here a polyphonic version of an instrument definition. The symbolic references of the synth-plug boxes will be modified by this scheme by adding to the name an index at each iteration step during the copying process. Thus for instance the name ':env' will become ':1/env', :'2/env', and so on, until 'count' has been reached. This is done in order to keep the names of each copied plug box instance separate.

Figure 3 contains also a plug input that accepts a list of floats as input instead of single float values. This is useful as here we can define and control envelopes in our system. The envelope generator–see the 'envelope-trigger' box–accepts a list of y and x values at the 'envelope' input.

Our event code example utilizing the instrument definition in Figure 3 assumes that the instrument will be copied 5 times ('max-poly' is equal to 5). In order to handle our polyphonic case we need to add to the 'name' arguments of the event methods correct index values so that they can refer to the individual plug instances of the instrument definition. This is done by using the optional :ID keyword argument. At each iteration step in the 'iter' loop we ask for a new ID value by calling the function 'get-next-synth-id'. This will return one by one numbers in a circular fashion starting from 1 up to 'max-poly' until the upper limit (here equal to 5) is reached. After this we start with ID number 1 again, and so on. Internally the name argument of the events will be modified in a similar fashion than was done during the copying scheme in the instrument definition.

Our code example is otherwise very similar to the one in Section 4. The code-box code analyzer will find this time the following free variables: 'low-high', 'dur', 'dtime', and 'instrument', where 'low-high' gives the sample ID limits that will be used to access samples from the current repertoire of sound samples known by the system (each sound sample has a unique ID number), 'dur' gives the global duration of the result, 'dtime' gives the delta-time between individual events, and 'instrument' is the instrument definition. In the actual code we use again the 'iter' loop construct to generate the events.

```
(with-synth instrument 5
  (let ((low (first low-high)) (high (second low-high))
        id sample)
    (iter (for time from 0 to dur by dtime)
        (setq id (get-next-synth-id))
        (setq sample (random2 low high))
        (synth-event time
           :env (mk-envelope
                      (list 0 0.01 dtime) (list 0 0.2 0))
           :id id)
        (synth-event  time :sample sample :id id)
        (synth-trigger time :trig :id id)
        (synth-trigger time :trigenv :id id))))
```

6. A CASE STUDY: MACRO-NOTE INTERFACE

In the final section we utilize our system in a more complex context. In the following example we sketch briefly how the user can interface existing compositional tools found in PWGL with our new event scheme. Our starting point is the 'macro-note' concept that has been used to simulate various idiomatic playing styles for our physics-based instrument models [7] (sound examples can be found at: www.siba.fi/pwgl/pwglsynth.html).

The instrument behind this example is close to the one in Figure 1, except we are using the 'copy-synth-patch' box to turn the resonator instrument into a polyphonic one (in this specific case 'max-poly' is equal to 10).

The code has four inputs ('midis', 'dur', 'indices', 'instrument') and it can be split into two major parts. First, it produces two lists of notes ('notes1' and 'notes2') by calling twice the 'macro-note' function. After appending these results, the final part of the code ('send events') loops through this note list, extracts from each note the relevant information needed for the synthesis, and finally sends the events to the instrument. The resulting two textures can be seen in piano roll notation in Figure 4.

```
(with-synth instrument 10
  (let* ((midis1(first midis)) (midis2 (second midis))
         (notes1 (macro-note     ;; notes1
             :dur dur
             :ornaments '(n g-1) :dtimes '(0.1 0.15)
             :midis (mapcar 'list midis1 midis2)
             :indices '(3 2 1 1 2)
             :amp (mk-bpf '(0 0.5 1.0) '(60 120 30))
             :time-modif
                (mk-bpf '(0 0.5 1.0) '(100 300 100))
             :len-function '(= 0  (mod (1- len) 3))
             :extra-params (list
                (list :pan (mk-circ-list '(-45 0 45)))
                (list :dist (mk-bpf '(0  1.0) '(1 9))))))
        (notes2 (macro-note      ;; notes2
             :dur dur    :ornaments '(n g-1)
             :offsets '(0 0.2 0.29)  :dtimes '(0.1 0.1)
             :midis (g+ 12 (mapcar 'list midis2 midis1))
             :indices indices
             :amp (mk-bpf '(0 0.5 1.0) '(60 120 30))
             :time-modif
                 (mk-bpf '(0 0.5 1.0) '(100 90 100))
             :len-function '(= 0  (mod (1- len) 3))
             :extra-params  (list
               (list :pan (mk-circ-list '(-45  45)))
               (list :dist (mk-bpf '(0  1.0) '(15 9)))))))
 ;; send events
    (iter (for note in (append notes1 notes2))
          (let* ((time (read-key note :enp-startt))
                 (id (get-next-synth-id)))
            (synth-event time :amp
                 (/ (vel note) 127 4) :id id)
            (synth-event time :freq
                (m->f (midi note)) :id id)
            (synth-event time :pan
               (read-extra-param note :pan time) :id id)
            (synth-event time :dist
               (read-extra-param note :dist time) :id id)
            (synth-trigger time :trig :id id)))))
```

7. CONCLUSIONS

This paper presents an approach where the user can combine visual instrument definitions with textual code fragments situated in code-boxes in order to generate musical textures. The code-boxes can be triggered by the user resulting at the end in low-level events that control the associated instrument definitions. Furthermore, the code-box inputs can be interfaced with the visual part of our system. This, in turn, provides an interesting bridge between sound synthesis and our extensive set of tools related to computer-assisted composition, music notation, and constraint-based programming. Future plans include investigations of different caching strategies that would allow the system to run strictly in real-time. Furthermore, we will invest in the forthcoming PWGLSynth2 new strategies that would allow the the system to allocate new voices on the fly by the underlying synthesis engine. In the current system the user has to adjust the number of voices by trial and error, which may result in audible glitches if the given max-poly is exceeded by a process.

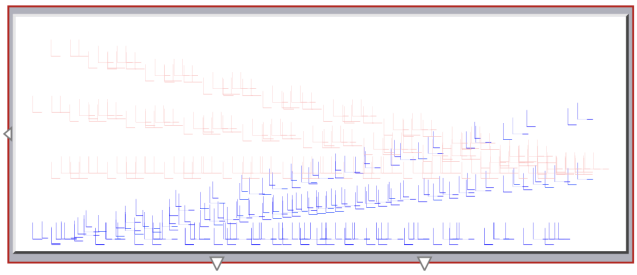

Figure 4: *One possible ten second realization of two overlapping macro-note textures. The pitch material of the lower texture progresses gradually from closed to open, whereas in the higher texture this process is reversed. The upper texture is more dense and contains very fast canon-like overlapping gestures.*

This work has been supported by the Academy of Finland (SA 105557 and SA 114116).

8. REFERENCES

[1] Mikael Laurson, Vesa Norilo, and Mika Kuuskankare, "PWGLSynth: A Visual Synthesis Language for Virtual Instrument Design and Control," *Computer Music Journal*, vol. 29, no. 3, pp. 29–41, Fall 2005.

[2] Mikael Laurson, Mika Kuuskankare, and Vesa Norilo, "An Overview of PWGL, a Visual Programming Environment for Music," *Computer Music Journal*, vol. 33, no. 1, 2009.

[3] Gerard Assayag, Camillo Rueda, Mikael Laurson, Carlos Agon, and Olivier Delerue, "Computer Assisted Composition at IRCAM: From PatchWork to OpenMusic," *Computer Music Journal*, vol. 23, no. 3, pp. 59–72, Fall 1999.

[4] Heinrich Taube, "Common Music: A music composition language in Common Lisp and CLOS," *Computer Music Journal*, vol. 15, no. 2, pp. 21–32, Summer 1991.

[5] James McCartney, "Continued Evolution of the SuperCollider Real Time Environment," in *Proceedings of ICMC'98 Conference*, 1998, pp. 133–136.

[6] Torsten Anders and Eduardo R. Miranda, "Constraint-based composition in realtime," in *Proceedings International Computer Music Conference*, 2008.

[7] Mikael Laurson and Mika Kuuskankare, "Towards Idiomatic and Flexible Score-based Gestural Control with a Scripting Language," in *Proceedings of NIME'08 Conference*, Genova, Italy, 2008, pp. 34–37.