RECENT DEVELOPMENTS IN PWSYNTH

Mikael Laurson and Vesa Norilo

Centre for Music and Technology, Sibelius Academy, P.O.Box 86, 00251 Helsinki, Finland
laurson@siba.fi,vnorilo@siba.fi

ABSTRACT

PWSynth was originally a visual synthesis language situated in PatchWork. Recently our research team has started a complete rewrite of the system so that it can be adapted to our new programming environment called PWGL. In this paper we present the main differences of the old and new systems. These include switching from C to C++, efficiency issues, interface between PWGL and the synthesis engine, and a novel copy-synth-patch scheme.

1. INTRODUCTION

Real-time sound synthesis has recently become more feasible due to advances in low-cost hardware. This evolution has opened new possibilities of how to combine high-level computer assisted composition environments with real-time sound synthesis. Computer assisted composition environments have been used mostly as non-real-time tools that deal with compositional problems resulting in material or scores for acoustical instruments. Sound synthesis, in turn, has been used often within improvisational real-time systems or to enhance musical instruments with the help of live electronics. PWSynth [1] can be seen as an attempt to make a bridge between these worlds that were previously seen as separate entities. Here the user can use the system as a general purpose synthesis engine or to use music notation as a starting point to generate control information for sound synthesis. The latter approach has been extensively used to control physical models of musical instruments ([2] and [3]).

PWSynth was originally written as a PatchWork (PW, [4]) user library. It consisted of a collection of C-subroutines and a collection of visual boxes that could be used within the PW environment to define sound synthesis patches. Lately PW has been rewritten and has resulted in a new visual language called PWGL [5]. The main difference between PW and PWGL is that the graphics part of the latter one is written in OpenGL. During this rewrite process we decided also to renew the PWSynth system. The most important change was to use C++ instead of C. C++ allows to use an object-oriented approach that results in a system where new DSP-units can be designed in a more flexible manner than before. (Benefits of using object-oriented languages in DSP-applications are discussed for instance in [6].) Also we removed several bottlenecks of the previous system so that the current version is significantly faster than the old one.

PWSynth consists of two main parts: the C-component and the Lisp-component. The C-component is written in C++ and it contains a library of DSP-units, real-time scheduling, sequencer, audio hardware support and some other general purpose tools. Our system uses the PortAudio library [7] for cross-platform audio input and output. The Lisp-component, in turn, is written in Common Lisp and CLOS (Common Lisp Object System) and is used to access the DSP-units database provided by the C-component. In PWGL this information is used to build popup-menus, boxes and sliders. These visual components can be used to build patches which in turn are translated into a sequence of C function calls. The C-component builds an internal representation of the patch and starts a scheduling process which results in an audio stream. While the scheduler is running it can receive events from the PWGL system. These events can either be user events (such as changing the state of a slider during playback) or pre-calculated sequencer events (typically coming from a music notation package). The synthesis engine updates the outputs and the inputs of the DSP-modules at each sample, which is needed for applications dealing with physical modelling.

The rest of the paper is organised as follows. First we give an overview of the C-component of the system. This section describes a general DSP-unit class. Next we go over to efficiency issues. We describe how a PWGL patch forming a graph structure is translated into a linear queue structure. We also give some ideas of how the system responds to events. After this we show how the user can define DSP-units. In Section 3 we discuss the Lisp-component. We describe among other things the database provided by the C-component and show how it is translated into visual entities that can be used to build patches in PWGL. We end with an implementation example that demonstrates some of the key features in the system.

2. C-COMPONENT

This section discusses the main features of a generic C++ base class called `pwsBox` and how it is used to define all DSP-units within PWSynth.

2.1. pwsBox

The most important members of `pwsBox` are `inputs`, `idName`, `idCategory` and `docString`. The members define the necessary data that is required so that C++ DSP-units are functional and that they can be converted to visual boxes in PWGL. `pwsBox` has one or several input-boxes that are represented by `pwsInputCaps` structures. The `inputs` member is a pointer to an array of input-boxes. `pwsInputCaps`, in turn, has members for `idName`, `dataType`, `default-value`, `min-value`, `max-value` and some other members for internal use.

2.2. Efficiency issues

The section will show how the synthesis engine has been optimised. A visual PWGL patch which is a graph structure

(i.e. a tree that can contain cycles) is collapsed into a linear array of DSP-boxes. This approach improves significantly the speed of sample calculation without sacrificing the generality and the flexibility of the system.

Before the real-time run, the patch is sent to the C-component as a series of commands in order to create and connect the modules. Internally, the C-component treats the patch as a linear queue of subroutines. Starting from synth-box, the compiler collects the boxes its input depends on. These are then inserted at the head of the queue, making sure each module precedes all the modules that depend on its output and that every module is added exactly once.

These two precautions allow to build arbitrary patches with recursions, loops and splits. In the case of recursion the patch cannot be rendered perfectly in the digital domain. Recursive connections exhibit a unit delay as a by-product of the scheduling scheme

2.3. Event handling

While running, the synthesis engine can respond to events in several ways. In the simplest case events can be sent from PWGL so that values are written directly into the data addresses of the input-boxes. Often it is, however, beneficial to use a scheme where events call a refresh method that allows to convert the incoming data in some meaningful way. Filters, for instance, get often user updates as high-level data (such as frequencies, amplitudes and bandwidths) which are in turn converted to low-level coefficients. Another example where the refresh method is useful is a spatialization module, such as the VBAP (Vector Base Amplitude Panning [8]) system, where typically the high-level input data (azimuth, elevation and distance) changes slowly.

Our system is optimal in the sense that data is converted only by the refresh method when receiving events. Otherwise the calculation runs with full speed using low-level parameters. Another benefit provided by this scheme is the fact that our system typically does not need separate box definitions for audio and control rate cases.

2.4. Vectored inputs and outputs

PWSynth DSP-boxes support vectored inputs and outputs (mono signals are only a special case where the vector length is equal to 1). This scheme is useful as it allows to construct compound entities which are used often in sound synthesis such as banks, parallel structures, serial structures, etc. As any box can return a vectored output, each item in the vector can be processed separately, if necessary.

PWSynth provides a rich set of tools that allow to manipulate vectors. For instance vectors can be summed into one signal, vectors can be split into sub-vectors, etc.

2.5. Box creation in C++

Next we give two concrete examples of how new box instances can be defined using C++. A box definition is done in three steps. First, a header file gives a class declaration typically inheriting from the general base class `pwsBox` discussed in Section 2.1. The second step consists of an initialisation routine - implemented in a C file - giving number of inputs, box name, box category name and input names. Finally, we specialise a method called `process` which executes the actual calculation. Some boxes also require the specialisation of methods that initialize and/or refresh the box.

The first example implements a resonating filter. The code is shown in the Appendix. Due to space limitations we can give only a rough overview of the code. The resonator has four inputs: signal, amplitude, frequency and bandwidth. We first declare a new class inheriting from the base class `pwsBox` (1). In the implementation we provide the database information for the box-name, inputs, popup-menu, etc. (2). In (3) we initialize the low-level state variables. Next we define a `refresh` method that is called each time there is a change in the high-level input values (4). Finally, in (5) we define the `process` method that performs the actual sample calculation using the low-level state variables.

The second example - also given in the Appendix - is a bank of resonators which takes one signal input, vectored inputs for amplitudes, frequencies and bandwidths and returns a vectored output. In (1) the new class `pwsResonBank` inherits from a generic `pwsBankBaseBox` class. In order to define a new bank module, basically the only thing that we have to do - besides some initialization routines in (2) - is to define the `createNewUnit` method (3). This method defines the basic DSP-unit - in this case the resonating filter - used by the bank module to calculate the final output. This scheme is very powerful as it allows to reuse the existing modules in the system to build more complex entities.

3. LISP-COMPONENT

When PWGL is launched it creates all necessary components so that the synthesis engine can be used within a visual programming environment. These include a hierarchical pop-menu which is built according to the category-name and box-name information provided by the C-component. Furthermore, the system builds a database from available DSP-boxes according to box-names and input-box information. This process is automatic and reflects the current state of the C-component. Thus any changes in the C-component will correctly be updated in the visual part of the system.

3.1. PWGL interface

A synthesis patch is realised in three steps. First, the user builds a visual PWGL patch consisting of DSP-boxes and connections. Special abstraction boxes can be used to define sub-patches. After this the patch is translated into C-function calls to add boxes, connections and input terminators. These are in turn used by the C-component to build the internal representation that will perform the final sample calculation. Figure 1 shows a typical PWGL window that contains a synthesis patch implementing a guitar model:

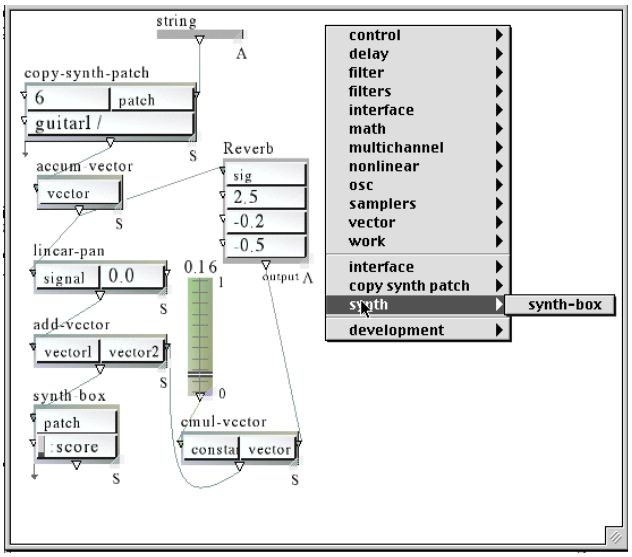

Figure 1: *A PWSynth example patch with boxes, connections, two abstractions, a real-time slider and a popup-menu for creating boxes.*

3.2. Copy-synth-patch scheme

Visual synthesis languages have been criticised of being static. Furthermore, more complex synthesis problems tend to result in patches that are crowded and confusing. One way to avoid these problems is to introduce special control structures that allow to mimic similar constructs like loops found in some textual synthesis languages, like SuperCollider [9]. In order to meet this challenge PWSynth introduces a new special box called copy-synth-patch having two inputs count and patch. copy-synth-patch duplicates a patch connected to the patch-input count times. This scheme has many applications such as a guitar-model that contains for instance six string-model instances. Our system allows to define a string-model only once and let the copy-synth-patch box duplicate the required amount of strings automatically. Figure 1 shows one copy-synth-patch example where the patch input is connected to an abstraction box called 'string'. The contents of the abstraction is copied here 6 times in order to create a guitar model.

In order to distinguish between different duplicated patch instances the copy-synth-patch scheme generates automatically symbolic references to specific user defined entry points. These entry points are specified by connecting a PWSynth-plug box at the leafs of a synthesis patch. The entry-points are used afterwards to control the synthesis process. The symbolic references are path-names, similar to the one found in OSC [10] and in the previous version of PWSynth, such as 'guitar1/1/freq' or 'guitar2/6/lfgain'.

3.3. Other applications

The clear and compact interface between the Lisp-component and the C-component allows to create other synthesis related applications in PWGL which are not directly related to a visual patch. Such applications are of great interest in our system as it allows to use sophisticated OpenGL based 2D- or 3D-graphics in conjunction with sound synthesis. Possible applications are for instance sample editors, interactive instrument models and tools for computer aided instrumentation.

4. IMPLEMENTATION EXAMPLE

We end with an implementation example that demonstrates some of the capabilities offered by PWSynth. The example patch in Figure 2 is a straightforward implementation of a feedback delay network (FDN) reverberation [11]. Since FDN is basically a vectored comb filter, the vectorization capabilities of PWSynth offer an advantage. The building blocks for FDN are delay, loss filter and feedback. By vectorizing them and turning the feedback into a matrix, a prototype FDN is created. Input signal is injected into a vectored delay line that is in turn fed into a vectored loss filter. A highly efficient Hadamard matrix is used to distribute the feedback between delay lines. The vectored modules automatically determine the number of elements by examining their inputs. In the example case we are using eight elements in each vectored module. The benefits of PWSynth-based effect design are the quickness of the design/implementation/evaluation cycle and the possibility of using custom routines written in Lisp to calculate various low level parameters such as loss filter coefficients and delay line lengths. Additional twists such as inserting allpass filters in the feedback paths or changing the way the delay line lengths are chosen can be done quickly and easily. PWSynth also provides an infrastructure to flexibly control sound processing via real-time or sequenced events.

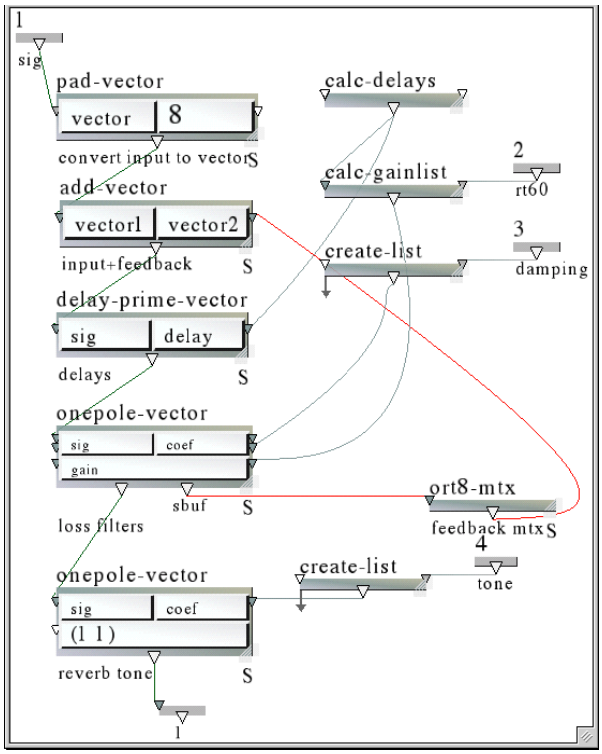

Figure 2: *An abstraction patch which defines a FDN reverberator.*

5. CONCLUSIONS

This paper has presented a new visual software synthesis language written in C++ and Lisp. The key features in the new system are an improved C++ programming environment, speed optimizations, event handling, vectored input and output boxes and a novel copy-patch scheme. The aim is to combine the speed of low-level DSP-routines calculation with the flexibility of a high-level visual programming environment.

The current system runs on Mac OS 9 and our work is now concentrated on a OS X port. Future plans include a port to Linux and Windows.

6. REFERENCES

[1] M. Laurson and M. Kuuskankare, "PWSynth: A Lisp-based Bridge between Computer Assisted Composition and Sound Synthesis", in *Proc. ICMC'01*, pp. 127-130, Havana, Cuba, 2001.

[2] M. Laurson, C. Erkut, V. Välimäki, and M. Kuuskankare, "Methods for Modeling Realistic Playing in Acoustic Guitar Synthesis", Computer Music Journal, vol. 25, no. 3, pp. 38-49, Fall 2001.

[3] V. Välimäki, M. Laurson, and C. Erkut, "Commuted Waveguide Synthesis of the Clavichord", Computer Music Journal, vol. 27, no. 1, pp. 71-82, Spring 2003.

[4] M. Laurson, PATCHWORK: *A Visual Programming Language and Some Musical Applications*. Doctoral dissertation, Sibelius Academy, Helsinki, Finland, 1996.

[5] M. Laurson and M. Kuuskankare, "PWGL: A Novel Visual Language based on Common Lisp, CLOS and OpenGL", in *Proc. ICMC'02*, pp. 142-145, Gothenburg, Sweden, 2002.

[6] V. Lazzarini, "Audio Signal Processing and Object-oriented Systems", in *Proc. of the 5th Int. Conference on Digital Audio Effects (DAFX-02)*, Hamburg, Germany, pp. 211-216, 2002.

[7] R. Bencina and P. Burk, "PortAudio – an Open Source Cross Platform Audio API", in *Proc. ICMC'01*, pp. 263-266, Havana, Cuba, 2001.

[8] V. Pulkki, "Virtual sound source positioning using vector base amplitude panning", in *Journal of the Audio Engineering Society*, 45(6) pp. 456-466, June 1997.

[9] J. McCartney, "Continued Evolution of the Super-Collider Real Time Environment", in *Proc. ICMC'98*, pp. 133-136, Ann Arbor, USA, 1998.

[10] M. Wright and A. Freed, "Open Sound Control: a new protocol for communicating with sound synthesizers", in *Proc. ICMC'97*, pp. 101–104, Thessaloniki, Greece, 1997.

[11] J-M. Jot, "Efficient Models for Reverberation and Distance Rendering in Computer Music and Virtual Audio Reality", in *Proc. ICMC'97*, pp. 236-243, Thessaloniki, Greece, 1997.

7. APPENDIX

```
        /* 1. Example: pwsResonBox */

class pwsResonBox:public pwsBox{            // (1)
public:  pwsResonBox();
         virtual void Process();
         virtual void Prepare();
         virtual int Refresh(int numInput);
         enum{ksig=0,kfreq,kamp,kbw,kNumInputs};
protected:float a0,b1,b2,p_out1,p_out2;};

pwsResonBox::pwsResonBox()                  // (2)
{setNumInputs(kNumInputs);
 setBoxName("reson");
 setBoxCategory("filters/IIR");
 setInputName(ksig, "sig"); setInputName(kfreq, "freq");
 setInputName(kamp, "amp"); setInputName(kbw, "bw");
 setInputFlag(kfreq,kNeedRefresh);
 setInputFlag(kamp,kNeedRefresh);
 setInputFlag(kbw,kNeedRefresh);
 boxCaps.docString="Two pole IIR resonator filter";
return;}

void pwsResonBox::Prepare()                 // (3)
{p_out1=p_out2=0;a0=b1=b2=0;}

int pwsResonBox::Refresh(int numInput)      // (4)
{ float fc,am,bw;
  float **ptr=(float **)inputBoxBuf;
  fc = *ptr[1]; am = *ptr[2]; bw = *ptr[3];
  b2 = pow(neperE,(NegPi2*(bw/_pws_sampleRate)));
  b1 =(((-4*b2)/(1+b2))*cos(NegPi2*(fc/_pws_sampleRate)));
  a0 = am;
  return pwsBox::Refresh(numInput);}

void pwsResonBox::Process()                 // (5)
{float sig, out;
 float **ptr=(float **)inputBoxBuf;
 sig = *ptr[0];
 out = ((a0*sig)-(b1*p_out1)-(b2*p_out2));
  p_out2 = p_out1;
  p_out1 = out;
  *((float *)outputBuf) = out;}

        /* 2. Example: pwsResonBank */

class pwsResonBank:public pwsBankBaseBox{   // (1)
public:  pwsResonBank();
         virtual pwsBox *createNewUnit();
         enum{kSig,kFreq,kAmp,kBW,kNumInputs};};

pwsResonBank::pwsResonBank()                // (2)
{copyUnitSettings();
 setBoxName("reson-bank");
 setBoxCategory("filters/vector");
 boxCaps.docString=
  "bank of resonators operating on the input signal";}

 pwsBox *pwsResonBank::createNewUnit()      // (3)
 {return new pwsResonBox;}
```