

The pureCMusic (pCM++) framework as open-source music language

Leonello Tarabella

computerART project of ISTI/CNR
Research Area of CNR, Pisa
via Moruzzi 1 – 56124 Pisa, Italy
leonello.tarabella@isti.cnr.it
<http://tarabella.isti.cnr.it>

Abstract. The pureCMusic (pCM++) framework gives the possibility to write a piece of music in terms of an algorithmic-composition-based program -also controlled by data streaming from external devices for giving expressiveness in electro-acoustic music performances- and of synthesis algorithms. Everything is written following the C language syntax and compiled into machine code that runs at CPU speed. The framework provides a number of predefined functions for sound processing, for generating complex events and for managing external data coming from standard Midi controllers and/or other special gesture interfaces. I'm going to propose pCM++ as open-source code.

1 Introduction

In order to put at work the mapping paradigm and the expressive facilities offered by the gesture interfaces we realized at *computerART project of CNR-Pisa* for composing and for performing with expressiveness interactive computer music [3], [5], [6], [7], [8], [9], [13] I started to write a very basic library of functions for sound processing. In the long run the library became a very efficient, stable and powerful framework based on pure C programming, that is *pure-C-Music* or *pCM*.

This programming framework gives the possibility to write a piece of music in terms of synthesis algorithms, score and management of data streaming from external interfaces. The pCM framework falls into the category of the *embedded music languages* and has been implemented using the Xcode C language compiler i.e. the native, free, development environment available on the MacOS X operating system of Macintosh computers.

As a result a pCM composition consists of a XCode project assembled with all the necessary libraries able to implement in realtime the typical synthesis and processing elements such as oscillators, envelope shapers, filters, delays, reverbs, etc. The composition itself is a C program that mainly consists of the Score and Orchestra parts.

Everything here is compiled into machine code and runs at CPU speed.

The Object Oriented paradigm is used for defining instruments in terms of class declaration then instanced as many times as wanted. It's a good practice to collect in

the Orchestra section all the necessary instruments and to add together the result of the computation of every instrument is use.

For that the name of the framework is pCM++. At the end, pCM++ is a realtime programming music language that in respect to the other textual languages such as CSound [1] and SuperCollider [4], has two main advantages: there is no need to learn a *new* language (just because *this is* the C language itself) and what's more, it is much faster (more than 10 times) in respect to those quoted because pCM++ entails compilation rather than interpretation.

1.1 Past and Future

I have already presented previous implementations of pCM at other conferences [10], [11], [12]. The new version here introduced runs as host of Xcode compiler (MacOSX). It is easily portable to other platforms and also provides a general purpose console interface the composer can use for setting up synthesis algorithms, for monitoring in/out audio signals and MIDI messages. Printout alerts and others basic interaction features (not oriented to live performance) are also provided.

The pCM++ framework has been written in Standard C++ [2] and makes use of the PortAudio and PortMIDI libraries commonly used as open-source and multi platform libraries able to manage audio signals and MIDI messages in real time. As a consequence it can be easily ported to other platforms such as Windows and Linux.

My aim is towards releasing pCM++ as open source code to be used and enriched by other composers and researchers.

2 Composition

A pCM++ composition consists of an Xcode-C-compiler project properly packed using all the necessary standard C libraries and the new original library that puts the framework at work. This library consists of a number of functions called *elements* (at the moment more than 50) able to implement in realtime the typical synthesis and processing building blocks such as oscillators, envelope shapers, filters, delays, reverbs, etc..

A composition is organized as an usual C program where the main program opens and closes all the necessary audio and MIDI message channels and calls in sequence the movements that make part of the composition. The default name of the main program is `Score` and is intended as an algorithm (from simple to complex) which also may include sequences of note-events as requested by scored music.

The `Orchestra` that generates the audio signal is defined as a separate void and consists of two main parts: the declaration of the elements (such as oscillators and filters) used for defining the synthesis algorithm, and the synthesis algorithm itself.

The orchestra, as expected, usually consists of more than one instrument.

The `Score` is the program part, which triggers and *feeds* the instruments by assigning proper values to common variables.

Here and in the following, the `courier` font is used for example programs and **bold names** are used to indicate reserved keywords of pCM. This is an example of a very simple composition:

```
float  vol,freq; //common variables for communication
                //between Score and Orchestra

DefOrchestra Ther() { //name of the orchestra
    sinosc sinT=newOsc(); //declaration of an oscillator
    BeginOrch
        float alfa=vol*Sinosc(sinT,freq);
        outLR(alfa,alfa);
    EndOrch
}
void Score() {
    float hor,vert;
    InitpCM
    Orchestra=Ther; // The Ther orchestra is activated
    Movement
        GetMouse(&hor,&vert);
        freq=100.+hor*1000.;
        vol=vert;
    EndMovement
    ClosepCM
}
}
```

An Orchestra is identified by a name and referenced when necessary. All the necessary variables are defined following the C language syntax. Values are assigned to variables by instructions that make part of the program defined in the Score section, that is the composition, or by data coming from the external. The Orchestra uses common variables for getting parametric values computed by the Score.

The synthesis algorithm consists of the code placed in the block **BeginOrch-EndOrch** that also includes the `outLR(.,.)` element. It is possible to define as many as wanted orchestras but only one at a time can be active. The Score activates an orchestra with the instruction `Orchestra=orchname`.

The **Score** is the composition: it initializes pCM, activates the current orchestra, open (if necessary) MIDI channel, etc.; usually a composition is made up of a number of movements each defined by the block **Movement-EndMovement**. The block is intended as loop that exits under certain conditions (here not examined).

The two functions run as two concurrent processes at different rates. The Orchestra function is automatically (*by a callback mechanism*) called at each audio buffer switch and since an audio buffer of 256 samples long and a 44100kHz sampling rate are used, the Orchestra function is called every 5.8 msec. The rate of the **Movement** depends on the computer performance and on the complexity of the composition. Then it is not predictable but, as experienced, it is fast enough to complete all the realtime functionalities as required.

3 Toolkits

A composition makes use of the functions belonging to the original library which puts the pCM framework at work at both Score and Orchestra levels. This library consists of three different groups of functions named *toolkits*, each one devoted to specific tasks: DSP, Events and Command toolkits.

- 1 DSP toolkit deals with the synthesis and processing of sound and groups elements such as oscillators, envelope shapers, filters, delay lines, reverbs, etc.. (Orchestra level).
- 2 Events toolkit deals with the generation and the scheduling of events including timing and management of external events (Score level).
- 3 Commands toolkit manages messages coming from and sent to the Midi interface, controls the activation of the computer built-in CD player, allows to directly record onto memory the audio signal and to store it onto disk as .aiff or .wave file, provides miscellaneous mathematical functions (Score level).

Each toolkit gathers together functions of the same kind and shares with the others groups the same approach for defining and using the basic pCM elements. An *element* is a building block such as an oscillator, a delay line, a filter, an envelope, etc., used for assembling synthesis algorithms; pCM elements are declared in the same way ordinary C variables (e.g. `int`, `long`, `float`, `bool`) are declared. This is the list of the types provided, so far, by pCM:

```
oscillator, pluck, noise, pulse, envelope, slider, delay-  
line, lpfilter, hpfilter, bpfiler, reverb, delay, sam-  
ple, scheduler...
```

All types are `low-case` keywords and the elements are usually defined at the beginning of an Orchestra definition and of the Score. What follows is a quick overview of the main elements belonging to the pCM-toolkits as provided so far.

3.1 DSP Toolkit

This is the main toolkit that consists of those functions, which generate and process the audio signal therefore used inside an Orchestra.

Remarks. Computation of audio signals is performed in floating point mathematics so that variables and parameters must be defined as float. Values are normalized at 1.0 so that oscillators range between -1.0 and 1.0, envelopes range between 0.0 and 1.0, the overall signal sent DACs ranges between -1.0 and 1.0. Time and durations are given in seconds.

§ An oscillator is defined with the type

```
oscillator myosc=newOsc(phase);
```

and used in **Orchestra** as follows:

```
val=Osc(myosc, freq);
```

The **Osc**(myosc, freq) function returns the proper value for oscillator myosc at each sampling tick and stores it into the val variable (previously defined). The first parameter of **Osc** identifies the oscillator by the name given when defined; the second parameter specifies the current frequency of the oscillator, that can be given as a numeric value, a variable or a complex expression. The phase of the oscillator can be initialized when defined.

§ In order to create a delay line, the **delayline** type is used. The instruction

```
delayline mydelay=newDelay(12.5);
```

defines a delay line identified with the name mydelay and allocates a memory space corresponding to a duration of 12.5 seconds. There exist two main functions for using a delay line with the expected way of working:

```
PutDelay(delayname, value); and  
val=GetDelay(delayname);
```

§ An envelope requires more information, necessary for describing its shape in terms of breakpoints: the *enum* type facility of C is used for this task by defining a list of numbers arranged as the number-of-break-points followed by the sequence of break points given as couples of values (time,value). Then the envelope is created in this way:

```
float ev={3, .0,.0, .5,1.0, 1.2,.0};  
envelope myenvel=newlinEnv(ev);
```

The **newlinEnv**(ev) function allocates the necessary memory, computes all the values for the envelope by linearly interpolating the break points and returns the pointer to the envelope then stored in myenvel. It's also possible to use exponential interpolation using **newexpEnv**(.); in this case break-points must given in triplets, third value referring to the mode how exponential curves must be computed.

The main functions which manage envelopes are **trigEnv**(envname) used at Score level, which starts the scanning of the envelope and **Env**(envname) used in Orchestra that returns the current value of the envelope.

§ Sound samples are defined and loaded with

```
sample mysample=LoadSample(filename.wav);
```

As seen for envelopes, there exist two main functions that manage samples: **trigSample**(sampname); which starts the scanning of the sample and used at Score level, and **Sample**(sampname); which returns the current value of the sample (used in Orchestra).

§ Filters are defined with no particular specification but their behaviors: lowpass, highpass, bandpass, resonator, etc.

```
lpfilter mylpfilt=newlpFilter();  
hpfilter myhpfilt=newhpFilter();...
```

In Orchestra they are used as follows

```
va=LPFilter(mylpfilt,signal,stopbandl);  
vb=HPFilter(myhpfilt,signal,stopbandh);
```

§ A reverb is defined with **reverb** myrev=newReverb();

and used in Orchestra as expected:

```
currval=Reverb(myrev,signal);
```

§ A further element makes part of the DSP toolkit which is not precisely involved in sound synthesis or sound processing but it is a rather useful tool for overall controls: this is **Fader** (fname). A fader is created and initialized as follows:

```
fader myfader=newFader(initval);  
Then it is setup with wanted values by  
setFader(myfader, when, dur, start, end);  
and used with currVal=Fader(myfader);
```

The value of *when* is the precise moment (see 3.2 paragraph) when the fader is planned to start moving; *dur* tells how much time the fader takes for going from *start* to *end*. A fader is a general purpose facility that can be used, for instance, -for controlling the general volume of an instrument or of the whole orchestra, -for gradually introducing the amount of a sound effect such as chorus, flanging, reverb, etc.. A fader can be used at both Score and at Orchestra levels.

Final comment. What here reported is only a small but significant excerpt out of the DSP Toolkits developed so far: -there exist other generators such **Pluck** (based on the Karplus-Strong algorithm), **Pulse**, **Noise**, **Addosc** (based on a predefined harmonic-based look-up-table): -envelopes can be defined also as Attack-Sustain-Release envelopes and the **Envasr**(envnm) and **ReleaseEnvasr**(envnm) must be used; -audio samples can be played with different speeds using **SampleSpeed**(samplname,speed); -delay lines may be read inside the line with **TapDelay**(gap); -two other fundamental functions make part of this Toolkit which perform signal transmission to DACs and signal input from ADCs: **outLR**(left,right) and **inLR**(&left,&right).

3.2 Events Toolkit

The pCM framework has been implemented mainly bearing in mind the algorithmic approach to composition and the interactive gesture controlled live performance of electro-acoustic music paradigm. That entails the program that describes the

composition issues events in terms of timed sets of values, which affect the instruments defined in the Orchestra. Since everything happens in real time under the control of the running program/composition, events are treated with reference to the global variable `Time` that holds the updated realtime clock value. The `Resettime` directive forces `Time` to 0 so that it reports (in seconds) the time elapsed since the last `Resettime`. Testing `Time` can generate events such as a note-triggering. However there exists a much formal and efficient approach for managing events consisting of the so called *Scheduler mechanism*.

A Scheduler, in the pCM framework, is an element which gives the possibility of enqueueing timed events in order to be taken into consideration later at the *right* time. As usual, it's necessary first to define a Scheduler element:

```
scheduler mysched=newScheduler(evnum);
```

The `Event(schedname,dur,value)` function enqueues the event defined as duration-value couple, into the specified scheduler. This function is usually invoked at Score level and can also be affected by data coming from the external. Once the events are placed in the Scheduler queue, the instruction

```
if(nextEvent(schedname,&retval)) do_something(retval);
```

is used for checking whether or not the time duration of the current event is finished. If yes, `nextEvent` returns *true* and `retval` has a valid value of the next event that will be used in the instruction `do_something` that usually trigs an instrument.

`GetMidi(&cmd,&chn,&val1,&val2);` is a boolean function which returns *false* if no MIDI message has been received; otherwise it returns *true* and the `cmd`, `chn`, `val1`, `val2` variables report the proper values.

3.3 Commands Toolkits

This toolkit groups those functions that work as commands and directives for the composition. Their names specify what they do. The following four commands are usually placed in the main program: **AudioOpen**, **MidiOpen**, **MidiClose**, **AudioClose**. **ClickToStart** and **ResetTimer** are usually placed, where requested, in the movements. The following commands allow to play and to control audio tracks from the CD driver:

```
CDtrackSearch(tracknum);  
CDplay and CDstop  
CDvolume(vol);
```

`Record("filename");` starts to record the global audio signal (as sent to the DACs) onto memory with no loss of quality and save it as a file onto the Harddisk with the specified name and `.aiff` or `.wav` extension format.

3.4 Expandability

Since toolkits are defined as collections of ordinary C functions and are clearly visible by the user, they can be upgraded as wanted and/or requested. It's a good practice, however, to follow the same approach I used in developing the current state of the three toolkits in order to be consistent with all that already existent.

4 Instruments as Objects

The *Orchestra* computes the audio signal by processing the instructions which implements the instruments as defined by the composer using the DSP toolkit functions.

More than one instrument can be defined inside an orchestra, the number depending on their complexity and the power of the computer in use. In any case, when many instruments are defined in an Orchestra, conflict problems coming from the names of variables, delay lines, filters and envelopes in use may arise, specially when putting together previously programmed instruments.

In order to avoid these problems the object-programming paradigm has been introduced for defining and using instruments. Besides, as a result, a cleaner layout for the program-composition is reached.

An instrument is then defined as a class object and declared, that is, instanced in the movement as many times as required. This is done using the very formal criteria of Object Oriented programming. The following is an example of a simple instrument based on the *pluck* element with some other additional elements that enrich its functionality:

```
class Stringks {
private:
    pluck      string;
    envelope   envks;
    envelope   envlpf;
    delay      rit;
    lpfilter   filtk;
    bool       created;
    float      vala,vax,vaxrit;
    float      valf,pitch;
public:
    void       setup();
    void       trig(float freq);
    float      tick();
    ~Stringks();
};
```

The *String* class is defined as public object that includes both the private section where the elements and the variables are defined, and the public section where the methods are declared. Usually, in the class instrument declaration, three are the methods declared plus one for destroying the instanced objects. These methods do the following jobs: -sets up everything necessary in order to the object work properly

such as to create delay lines and envelopes, to load samples, etc.; -activates (`trig`) the synthesis algorithm and, finally, -performs (`tick`) the synthesis algorithm which actually computes the signal. Then the methods are given.

```

Void Stringks::setup() {
    string =newPluck();
    rit    =newDelay(0.05);
    filtks =newLPFilter();
    float  st[]={3,.0,.0,1.2,.01,1.0,1.5, 2.,.0,.0};
    float  fl[]={3,.0,.0,1.2,.01,1.0,1.5, 2.0,0.0,0.0};
    envks  =newexpEnv(st);
    envlpf =newexpEnv(fl);
}
void Stringks::trig(float frq) {
    pitch=frq;
    trigPluck(string,pitch);
    trigEnv(envks);
}
float Stringks::tick(){
    vax=Env(envks)*Pluck(string,pitch)+getDelay(rit);
    putDelay(rit,vax);
    vala=LPFilter(filtks,vax,(Env(envlpf)*1000));
    return vala;
}

```

Remarks. Basically, sound is here generated using the well known Karplus-Strong algorithm; however it is also controlled by an envelope shaper for better control on the sound *corda*; the signal is also added to a delayed copy of itself and then filtered with a variable-cut-frequency lowpass filter. It's now possible to define an instrument for an orchestra the same way that the primitive pCM elements are declared. The only difference consists in putting a * pointer before the name which identifies the instantiated object as required by Object Oriented programming:

```
Stringks *mystring; mystring->setup;
```

It's worthwhile to invoke soon the `setup` method in order to initialize the just created instrument. From now on, the `trig` method (with the wanted frequency given as parameter instead of through common variables) will be used at `Score` level and the `tick` method will be used at `Orchestra` level for getting the computed signal.

In the following `Score` example some of the features previously introduced are used: a `scheduler` for composing a semitone scale and two `faders`, the first one for rising the global volume and the second one for panning sound from left to right. The instrument `corda` is defined as an instance of the `Stringks` class just defined. Delay and panning effects are treated at orchestra level rather than in then object-instrument: it's up the composer to put here effects or to include them in the instrument definition. Notes are issued using `nextEvent(.,.,.)`.

```

Stringks *corda=new Stringks;
DefOrchestra KSorch() {
    BeginOrch
        val    = Fader(fadin)*corda->tick();
        signR  = val*Fader(panpot);
        signL  = val*(1.-Fader(panpot));
        outLR(signL,signR);
    EndOrch
}
void Score(); {
    float    note,semitone;
    scheduler scale(15);
    fader    fadin;
    fader    panpot;

    corda->setup;
    Orchesta=KSorch;
    for(int k=0,semitone=440.;k<=12;k++)
        {Event(scale,0.5,semitone);semitone=semitone*1.059;}

    InitpCM
    ResetTime
    Movement
        StartFader(fadin, 0.,2.,0.,1.);
        StartFader(panpot,0.,6.,0.,1.);
        if(nextEvent(scale,&note)) corda->trig(note);
    EndMovement
    ClosepCM
}

```

The composition is now ready and when Score is activated a scale from A3 to A4 with the string timbre lowpass-filtered and panning from left to right, is generated.

5 Conclusion

The pCM framework has been efficiently used for composing and performing many pieces of music under the control of the gesture tracking systems and devices realized at computerART project at ISTI/CNR in Pisa. The pCM framework has been implemented first for Macintosh computers. The current version here reported is the result of the experience I gained using the previous versions. As declared, I'm going to port the work for other platform and put it on the Net as a freeware open-source code music language.

6 Acknowledgements

Special thanks are due to Massimo Magrini who greatly contributed to set up the pCM main mechanism currently in use and the many facilities for data communication and audio processing. Also thanks to Roberto Neri who graduated in Electronic Engineering at Pisa University with a thesis regarding the upgrading of the pCM DSP Toolkit. pCM has been also included as special topic in the course of Computer Music I yearly teach at the Computer Science Faculty of Pisa University: special thanks to the many students who contributed to improve and/or optimize the pCM Toolkits.

References

1. Boulanger, R.: *The Csound Book*, The MIT Press, (1999)
2. Cargil, T.: *C++ Programming Style*, Addison-Wesley Professional Computing Series, Reading, MA, USA, (1992)
3. DePoli, G.: Expressiveness in music performance: Analysis and Modelling. In Proceedings of SMAC03. (Stockholm Music Acoustic Conference) Stockholm, (2003)17-20
4. McCartney, J.: <http://supercollider.sourceforge.net/>
5. Paradiso, J.A.: New way to play; *Electronic Music Interfaces*. IEEE Spectrum 34-12, (1997) 18-30
6. Rowe, R.: *Machine Musicianship*. Cambridge: MIT Press. March, ISBN 0-262-18296-8, (2001) (343-35)3
7. Tarabella L., Magrini M., Scapellato G.: Devices for interactive computer music and computer graphics performances, in Proceedings of the IEEE First Workshop on Multimedia Signal Processing, Princeton, NJ, IEEE cat.n.97TH8256, Computer Society Press, (1997)
8. Tarabella L., Bertini G.: Giving expression to multimedia performances. In Proceedings of ACM Multimedia 2000, Workshop "Bridging the Gap: Bringing Together New Media Artists and Multimedia Technologists" Los Angeles. (2000)
9. Tarabella, L., Bertini G.: The mapping paradigm in gesture controlled live computer music. In Proceedings of the 2nd International Conference "Understanding and Creating Music", Caserta -Seconda Università di Napoli, Facoltà di Scienze Matematiche, Fisiche e Naturali, (2002).
10. Tarabella, L., Bertini G.: The mapping paradigm in gesture controlled live computer music. In Proceedings of the 3rd International Conference "Understanding and Creating Music", Caserta -Seconda Università di Napoli, Facoltà di Scienze Matematiche, Fisiche e Naturali, (2003).
11. Tarabella, L.: The pCM framework for realtime sound and music generation. In Proceedings of the XIV Colloquium on Musical Informatic (CIM2003) Firenze, Italy, (2003).
12. Tarabella, L.: *Improvising Computer Music: an approach*. Sound and Music Computing, Ircam, Parigi. At <http://smc04.ircam.fr/> (2004)
13. O'Modhrain, S.: New Gestural Control of Computer-Based Musical, In Proceedings of NIME02 (New Interface for Musical Expression), Dublin, (2002)