

AudioDrome

Alessandro Petrolati

www.alessandro-petrolati.it

info@alessandro-petrolati.it

SOMMARIO

AudioDrome è una raccolta di codice scritto in linguaggio C++, il namespace fornisce le API (*Application Programming Interface*) per lo sviluppo di DSP (*Digital Signal Processing*) *open-source*. Progettato per una facile “portabilità”, il *framework* è formato da un nucleo di classi primitive e da classi *wrapper* per le librerie di terze parti. Gli “oggetti” interagiscono attraverso un’interfaccia flessibile e semplice. Questo articolo presenta le caratteristiche principali del lavoro e descrive in generale, le problematiche legate alla programmazione di *software* per l’elaborazione del segnale digitale. Il *framework AudioDrome* si impiega come un sintetizzatore *code-based*, simile a *Csound* o *Super Collider*.

DIGRESSIONE SU CSOUND

Un dispositivo digitale per l’elaborazione e per il trattamento del suono, deve necessariamente comprendere i seguenti punti:

I/O

Process

Control

I/O è l’ingresso e l’uscita del DSP (*Process*), lo *stream* audio può essere gestito dall’OS (*Operating System*) o da specifici *drivers* software: *CoreAudio*, *Jack*, *ALSA*, *OSS*, *MME*, *DirectX*, *ASIO* etc..., l’I/O *hardware* della scheda audio (*sound board*), è chiamato ADC (*Analogic Digital Converter*) e DAC (*Digital Analogic Converter*). Anche la lettura/scrittura di uno specifico *file* audio è I/O.

Process è il nucleo software per l’elaborazione e la generazione. I *Csound opcodes* e le classi *AudioDrome* sono oggetti per il *Processing*.

Control è un I/O specializzato, lavora sui protocolli di controllo come per esempio il MIDI (*Music Instrument Digital Interface*) e OSC (*Open Sound Control*), permette il controllo dei parametri di sintesi dei DSP.

Un esempio è la seguente *patch* di *Csound*:

```
instr 1
  iAmp = ampmidi 7000
  iBaseCps = cpsmidi

  aSig oscili iAmp, iBaseCps, 1
  out aSig
endin
```

Lo *strumento* implementa un oscillatore *table-lookup* interpolato, controllato in ampiezza e in frequenza da messaggi MIDI. L’esempio indica chiaramente la semplicità di programmazione e la straordinaria potenza e flessibilità di *Csound*. I codici operativi (*opcodes*) sono “funzioni-oggetti”, *ammidi* e *cpsmidi* leggono dal *buffer* del dispositivo MIDI (in questo caso solo in *input*), *out* scrive i campioni su un *file* audio e/o in *real-time* sul DAC della scheda audio. L’*opcode oscili* (*Interpolate Oscillator*) è il modulo d’elaborazione del segnale (*Processing*).

L’alto livello d’astrazione, non ci permette di comprendere quello che avviene tra l’*opcode out* e il DAC. *Csound* e *AudioDrome* gestiscono l’I/O audio tramite la libreria *cross platform Portaudio*. Generalmente il *file orchestra* di *Csound*, contiene l’*header* per il *setup software/hardware* del sistema:

```
sr = 44100
kr = 4410
ksmps = 10
```

S’impone come *sr* (*Sampling Rate*), una velocità di campionamento e come *kr* un sotto-campionamento; *ksmps* esprime il rapporto *sr/kr* che dev’essere intero. L’*header* è costante (*const*), pertanto non può essere cambiato a tempo d’esecuzione (*runtime*), la frequenza di campionamento (SR) è il dato di riferimento assoluto. Ogni campione con velocità *kr* (o *sr*), è separato da un intervallo di tempo di $1/kr$ secondi, in questo margine sono chiamate tutte le funzioni di calcolo degli *opcodes*. Nel caso di quelli che ritornano identificatori audio (che cominciano con la lettera ‘a’), sono chiamate *ksmps* volte le funzioni di processamento restituendo un *buffer* per ogni *kr* campione. L’esempio è preso da un sorgente di *Csound*:

```
int nn;
nn = csound->ksmps;

while (nn-->0) {
  ...
  xd = //opcode result
  *(ar++) = (MYFLT) xd;
}
```

la variabile d’appoggio locale *xd* è aggiornata col risultato di calcolo della funzione dell’*opcode*, viene incrementalmente copiata nel *buffer-array*, il modello è la base del sottocampionamento in *AudioDrome*.

SPECIFICHE

Le classi *AudioDrome* sono ADT (*Abstract Data Type*) che modellano specifici DSP, l’istanza delle classi appare molto simile agli *opcode* di *Csound*. Le funzioni d’interfaccia di classe *setQualcheCosa()* impostano i parametri di sintesi del DSP, la frequenza di campionamento è un dato assoluto. Questa è gestita staticamente e accessibile in *read/write* solo dalle funzioni *getAbsoluteSR()* e *setAbsoluteSR()*, ma a differenza di *Csound* tutti i parametri di configurazione dei DSP, la SR e il fattore di sotto-campionamento possono variare a tempo d’esecuzione infatti, quando varia a *runtime* tutti gli oggetti vengono notificati e si ri-configurano automaticamente. La specifica più importante di *AudioDrome*, permette agli oggetti di avere un proprio **sotto-campionamento** espresso in campioni, questo determina una **SR locale per ciascun oggetto**. Un sotto-campionamento di 4 campioni, significa una risoluzione di $SR/4 = 44.100/4 = 11025$ campioni e quindi un segnale con profilo a scalini (aggiornato ogni 4 campioni). Sotto-campionare gli oggetti di

controllo e/o secondari, agevola il lavoro della CPU e incrementa la *performance* (figura 1).

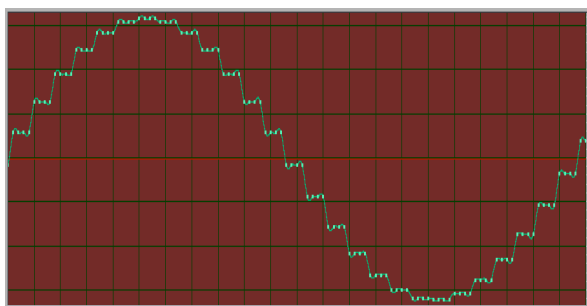


Figura 1. Sinusoide sotto-campionata di 4.

Ogni classe contiene la variabile **_localSR**, qualificata *private/protected*, l'aggiornamento del valore *_localSR* di classe, è gestito automaticamente dal meccanismo di notifica *Subject/Observer*, ogni volta che cambia nel sistema *_absoluteSR* o il fattore di *downsampling* per l'oggetto.

Le specifiche principali sono qui riassunte:

- Notifica "Sampling Rate is Changed" a runtime
- Down Sampling (sotto-campionamento) degli oggetti
- Notifica "Down Sampling is Changed" a runtime
- Facilità di connessione degli oggetti.

TASSONOMIA E INTERFACCIA

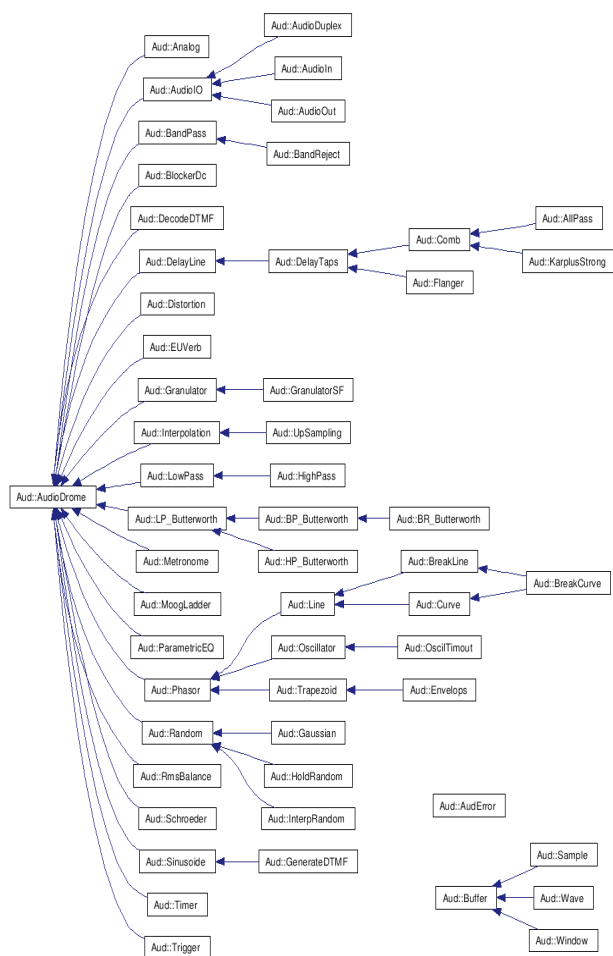


Figura 2. Le classi *AudioDrome*.

Nella gerarchia ereditaria (figura 2), ci sono modelli primitivi per la generazione e per l'elaborazione del segnale, tutte le classi DSP discendono da *AudioDrome* che contiene i metodi d'interfaccia. La "superclasse" inoltre, implementa le procedure di notifica "The Sampling Rate is Changed" e "Down Sampling". Ogni ramo corrisponde a uno specifico ambito operativo, ci sono classi per composizione e/o per aggregazione, l'argomento è approfondito di seguito.

Le classi derivate implementano le funzioni *set/get*, che forniscono l'interfaccia per l'oggetto mentre le funzioni "imperative" sono ereditate da *AudioDrome*, queste controllano il **PROCESSING** e l'I/O dei campioni dei DSP. I motori sono implementati localmente nelle sottoclassi, attraverso la procedura *void Processing()* qualificata *virtual protected* gestita da *public: PROCESSING()* di *AudioDrome*.

Un esempio delle funzioni *get/set*:

```
Oscillator obj; //creo un oggetto
obj.setFrequency(440); //imposto una frequenza
obj.getAbsoluteSR(); //imposto una Sampling rate
//oppure:
AudioDrome::setAbsoluteSR(48000);
```

l'interfaccia principale ereditata dalla "superclasse", queste funzioni **devono essere chiamate al campione**:

```
obj.PROCESSING();
obj.Input();
obj.Output();
```

necessariamente chiamate nel *loop* principale (ogni istante di campionamento) o nella *callback* (spiegata di seguito).

Un esempio del ciclo principale, nella funzione *main*:

```
long P3 = AudioDrome::getAbsoluteSR() * 5;
for (long i = 0; i < P3; i++)
{
    obj.PROCESSING();
    dac.PROCESSING();
    dac.Input( obj.Output() );
}
```

Il ciclo *for* di 5 secondi, invoca $SR * 5 = 44.100 * 5 = 220.500$ volte la procedura *PROCESSING()*. Il risultato è accessibile "al campione" (*sample*), tramite la funzione *Output()* che ritorna il valore (*typedef double AudFloat*). La funzione *Input()* dell'oggetto *dac* prende in ingresso l'uscita dell'oggetto *obj*. La specifica di sotto-campionamento è implementata in *PROCESSING()*, la procedura controlla il numero di chiamate delle "polimorfiche" *Processing()* di classe. Il fattore di sotto-campionamento s'impone con:

```
obj.setDownSampling(10);
```

come detto, le funzioni *Input()* e *Output()* accedono alle relative variabili di classe alimentate da *PROCESSING()*. Alcuni oggetti sono multi-canale, quindi dovranno implementare le necessarie funzioni per l'I/O e conterranno le relative variabili d'appoggio:

```
AudFloat Output_3() { return _output_3; };
AudFloat Output_4() { return _output_4; };
```

etc..

```
Input_3(const AudFloat & val) { _input_3=val; };
Input_4(const AudFloat & val) { _input_4=val; };
```

etc..

Anche se la logica “orientata al campione” è più semplice e intuitiva, si preferisce gestire il segnale a *bufferFrame* di campioni, oppure con funzioni *callback* multi-thread. Si rimanda ad una trattazione approfondita dell’argomento.

THE SAMPLING RATE IS CHANGED!

AudioDrome::Phasor è un modello primitivo importante, implementa un movimento di “*phase*” (rampa) normalizzato (da 0 a 1). Il numero di rampe per secondo è espresso dalla funzione *setFrequency()*, ciascuna rampa ha durata 1/freq. Un campionamento a intervalli di tempo regolari, tiene conto degli istanti di tempo che separano l’informazione, per calcolare valori d’incremento bilanciati (in relazione) sul numero di chiamate della funzione per secondo e sul valore di frequenza desiderato. *AudioDrome class* contiene la variabile *static _absoluteSR*, qualificata *private*. Le sotto-classes accedono al dato con *setAbsoluteSR()* e *getAbsoluteSR()*, secondo i canoni della OOP (*Object Oriented Programming*). In genere il valore di SR è impiegato nel calcolo delle variabili stazionarie di classe: incrementi, coefficienti, valori temporali etc... usate da *Processing()*. Questo compito è assolto dalle funzioni *sets*, che sollevano *Processing()* da operazioni inutili, non necessarie a tempo di campionamento. Le funzioni *sets*, aggiornano le variabili di classe soltanto quando invocate.

Com’è calcolato l’incremento:

```
void Phasor::setFrequency(const AudFloat &freq)
{
    _freq = freq;
    _incr = _freq / _localSR;
}
```

_localSR dipende dal fattore di sottocampionamento e da *getAbsoluteSR()*, *_incr* è utilizzata da *Processing()*:

```
void Phasor::Processing()
{
    _output_1 = _phi += _incr;
    if (_phi >= 1.0)
    {
        _phi -= 1.0;
    }
    else if (_phi < 0.0)
    {
        _phi += 1.0;
    }
    _output_1 -= _incr;
}
```

La procedura è richiesta da *PROCESSING()* nel ciclo principale SR volte per secondo, mentre *setFrequency()* solo per modificare la frequenza d’esecuzione delle rampe.

Cosa accade se, cambia la frequenza di campionamento (SR)?

La risposta è “relativamente” semplice: *_incr* non è aggiornato sulla base della nuova SR, la frequenza audio risultante dell’*object Phasor* sarà inversamente proporzionalmente al cambiamento della SR.

Serve pertanto, un meccanismo capace di aggiornare tutte le variabili di classe che dipendono da *getAbsoluteSR()* e/o dal numero di sotto-campionamento. L’implementazione della specifica si appoggia sul *Pattern Design* “*Subject/Observer*”, la tecnica di programmazione proposta dal **GoF** (*Gang of Four*) è semplificata grazie alla gerarchia ereditaria unitaria (tutte le sottoclassi discendono da *AudioDrome*). Il costruttore della classe *AudioDrome*, registra ogni istanza delle sottoclassi, il meccanismo di notifica chiama le procedure *srUpdate()* di tutti gli oggetti in memoria, localmente ciascuna *srUpdate()* inoltra una serie di chiamate alle funzioni membro che utilizzano il valore SR (generalmente le funzioni *sets*) e

alla *srUpdate()* della classe genitrice. Per contro, vanno considerate alcune regole quando si programmano nuove classi e quando si creano classi che contengono istanze di altri classi *AudioDrome*.

COMPILAZIONE E DIPENDENZE

Il *package* è un **Xcode 3.0 Project** GCC/GNU sotto **Macintosh**, l’unica dipendenza facoltativa è la **Portaudio library** che serve per il trattamento dell’I/O audio. La libreria permette una “portabilità” (da cui il nome) del codice in quasi tutti i sistemi operativi, con la *pre-processor* `__WITH_PORTAUDIO__` macro, si abilita la compilazione dei *files* necessari al supporto audio, le classi *AudioIO*, *AudioIn*, *AudioOut* e *AudioDuplex* sono *wrappers* di *Portaudio*, interfacciano le API normalizzandola secondo la specifica *AudioDrome*. Omettendo la macro si rimuove la dipendenza, per esempio includendo i sorgenti in progetti VST (*Virtual Studio Technology*), in cui l’I/O audio è trattato direttamente dall’*host*. La cartella contiene le librerie statiche **STK** (*Synthesis ToolKit*) di Gary P. Scavone/Perry R. Cook e **Portaudio** di Ross Bencina, compilate. Xcode IDE (*Integrated Development Environment*) è configurato per i seguenti *Targets*: *Static-Library* (*libaudiodrome.a*), *Shell* (compila un *exec* da terminale) e *Application* (compila e crea un *bundle* per Mac OSX). La sottocartella **Dependencies**, contiene le librerie e gli *header files* delle dipendenze. La *pre-processor* macro `__DOUBLE_PRECISION__`, definisce il tipo di dato *AudFloat*, questo è un *alias* per i tipi primitivi *double* (virgola mobile a doppia precisione 64 bit) o *float* (virgola mobile a singola precisione a 32 bit) secondo la codifica IEEE-754. La doppia precisione esprime il range $-10^{-324} \div 10^{308}$ contro $-10^{-45} \div 10^{38}$ dei *float*. *AudFloat* rappresenta la precisione di calcolo dei DSP (variabili di ritorno, locali, temporanee, etc...). Se *AudioDrome Library* (*libaudiodrome.a*) è compilata in modalità `__DOUBLE_PRECISION__`, i progetti che la includono dovranno dichiarare la macro, per avere una corretta corrispondenza (*match*) tra gli *headers* e *libaudiodrome.a*

TUTORIAL

1 - Hello grano!

```
/* Main.cpp */
#include "AudioDromeAll.h"
using namespace Aud;

int main(int argc, char ** argv)
{
    AudioDrome::setAbsoluteSR(44100);
    AudioOut dac;
    Granulator grn;

    ul P3 = AudioDrome::getAbsoluteSR() * 7.f;
    for (ul i = 0; i < P3 ; i++)
    {
        grn.PROCESSING();
        dac.PROCESSING();

        dac.Input( grn.Output() );
        dac.Input_2(grn.Output_2() );
        //or simply:
        //dac << grn;
    }
    dac.abortStream();
    return 0;
}
```

L’inclusione di *AudioDromeAll.h* comporta l’inclusione di tutti gli *headers files* delle classi, queste appartengono al *namespace Aud*. La direttiva *using* impone al compilatore di scavalcare il controllo, tuttavia in progetti di grandi dimensioni è sconsigliata a favore di una *scope resolution* esplicita.

La prima istruzione della funzione *main*, imposta il valore SR generale e poi s'istanzano gli oggetti per le classi *AudioOut* e *Granulator*. Dentro il ciclo *for*, si "attivano" i motori di *grn* e *dac*, quindi interconnessi mediante le relative funzioni I/O.

```
dac.Input( gr.Output() );
```

l'*overloading* degli operatori del C++, permette connessioni facili e intuitive:

```
dac << gr;
```

2 - Modulo riverbero in tempo reale:

```
/* Main.cpp */
#include "AudioDromeAll.h"
using namespace Aud;

int main(int argc, char ** argv)
{
    AudioDrome::setAbsoluteSR(44100);
    AudioIn adc;
    AudioOut dac;
    EUVerb euv;
    euv.setT60 (1.8);
    euv.setInperpolation(LINEAR);
    euv.setReverberatedLevel(0.3);
    euv.setDirectLevel(0.7);
    euv.setDirectDelay(0.9);

    while (true)
    {
        adc.PROCESSING();
        dac.PROCESSING();
        euv.PROCESSING();

        /* RealTime IO */
        euv.Input( adc.Output() );
        dac.Input( euv.Output() );
        dac.Input_2( euv.Output_2() );
        //or simply:
        //dac << euv << adc;
    }
    adc.abortStream();
    dac.abortStream();
    return true;
}
```

L'esempio presenta un riverbero basato sul modello proposto da Eugenio Giordani (LEMS), la classe è realizzata per composizione di oggetti *AudioDrome*, in questo caso il DSP prevede un solo *input* e restituisce un segnale pseudo-stereofonico. Ecco le connessioni alternative::

```
dac << euv << adc;
```

l'*output* di *adc*, ossia l'ingresso microfonico della scheda audio, è indirizzato sull'*input* del riverbero *euv*. L'uscita del riverbero sull'*input* del *dac*. L'operatore di direzione del flusso audio (*left shift*), collega gli oggetti in modalità STEREO per *default*, in questo caso l'*output* del riverbero è stereofonico.

Gli operazionali sovraccaricati semplificano ulteriormente la gestione del flusso del segnale:

```
dac << euv/2 << adc+0.5;
```

L'esempio di sopra divide l'uscita del riverbero e somma un *off-set* all'ingresso microfonico. Sono possibili le operazioni direttamente sugli oggetti:

```
dac << distort << (sine*0.5 + noise/10) - 1.0;
```

dove *sine*, *noise* e *distort* sono *AudioDrome* DSP.

3 - Full duplex audio stream mediante funzione call-back:

```
/* Main.cpp */
#include "AudioDromeAll.h"
using namespace Aud;

int MyCallback( const void *input, void *output,
                unsigned long frameCount,
                const PaStreamCallbackTimeInfo* timeInfo,
                PaStreamCallbackFlags statusFlags,
                void *userData )
{
    (void)(userData);
    // Prevent unused variable warning.

    PaFloat *out = (PaFloat*) output;
    PaFloat *in = (PaFloat*) input;

    for( unsigned int i = 0; i < frameCount; i++ )
    {
        *out++ = *in++;
        *out++ = *in++;
    }
    return 0;
}

int main(int argc, char ** argv)
{
    AudioDrome::setAbsoluteSR(44100);
    AudioDuplex io;
    io.setStreamCallBack(&MyCallback, NULL);

    io.sleep(60000);
    io.abortStream();
    return 0;
}
```

I parametri formali della funzione *MyCallback*, sono specificati da *Portaudio*, *AudioDuplex* è una classe *wrapper*, si noti l'assenza del ciclo *for* nel *main*, sostituito da *io.sleep(60000)*; *sleep* "congela" il *thread* corrente per un minuto (60000 Ms) mentre un'altro *thread* gestisce *MyCallback*. Nell'esempio la funzione collega semplicemente l'*input* con l'*output*.

4 - Riverbero in tempo reale in full duplex audio stream con funzione call-back:

```
/* Main.cpp */
#include "AudioDromeAll.h"
using namespace Aud;

int MyCallback( const void *input, void *output,
                unsigned long frameCount,
                const PaStreamCallbackTimeInfo* timeInfo,
                PaStreamCallbackFlags statusFlags,
                void *userData )
{
    EUVerb* euv = static_cast< EUVerb*>(userData);
    PaFloat *out = (PaFloat*) output;
    PaFloat *in = (PaFloat*) input;

    for( unsigned int i = 0; i < frameCount; i++ )
    {
        euv->PROCESSING();
        euv->Input(*in++);

        *in++;
        //not righth channel input for the EUVerb

        *out++ = euv->Output();
        *out++ = euv->Output_2();
    }
    return 0;
}
```

```

int main(int argc, char ** argv)
{
    AudioDrome::setAbsoluteSR(44100);
    EUVerb euv;
    AudioDuplex io;
    io.setStreamCallback(&MyCallback, &euv);

    io.sleep(60000);
    io.abortStream();
    return 0;
}

```

Simile all'esempio precedente ma in questo caso viene passato a *MyCallback* l'oggetto *AudioDrome::EUVerb*, trasferito come puntatore a *void*, occorre un un *cast* esplicito per ottenere l'oggetto *EUVerb*:

```
EUVerb* euv = static_cast< EUVerb*>(userData);
```

Il *loop for* incrementa i puntatori del *buffer* sul dispositivo *Portaudio* I/O, il segnale d'ingresso del riverbero è il canale sinistro dello *stream audio In*:

```
euv->Input(*in++);
```

l'uscita del riverbero è assegnata allo *stream audio Out*:

```
*out++ = euv->Output();
*out++ = euv->Output_2();
```

5 - Variazione a tempo d'esecuzione della SR:

```

/* Main.cpp */
#include "AudioDromeAll.h"
using namespace Aud;

int main(int argc, char ** argv)
{
    AudioDrome::setAbsoluteSR(44100);
    AudioOut dac;
    Wave* saw = new Wave(4096);
    saw->genSaw();
    Oscillator osc(saw);
    osc.setFrequency(440);

    Metronome mt(false);
    mt.setFrequency(1);

    ul P3 = AudioDrome::getAbsoluteSR() * 5;

    for (ul i = 0; i < P3; i++)
    {
        dac.PROCESSING();
        mt.PROCESSING();
        osc.PROCESSING();

        if(mt.Output())
            AudioDrome::setAbsoluteSR (11025);

        dac << osc;
    }
    delete saw;
    dac.abortStream ();
    return true;
}

```

Absolute Sampling Rate è inizializzata a 48 Khz, nel *loop for* l'istruzione *if* verifica se *mt* (*Metronome*) ritorna un *trigger true*, quindi dopo il primo secondo (poiché *mt* ha una frequenza di 1 Hz) il tasso di campionamento si aggiorna a 11 Khz. Quando il *trigger* invoca *setAbsoluteSR()*, tutti gli oggetti in memoria vengono notificati, ri-configurandosi a *runtime*. L'istanza *dac* di *AudioOut* deve necessariamente chiudere lo *stream audio* per poi riaprirlo sulla nuova SR.

Si noti il costruttore dell'oscillatore che accetta il puntatore *Wave*, ossia ad una tabella *look-up* di 4096, contenente il profilo a "dente di sega" usato come forma d'onda:

```

Wave* saw = new Wave(4096);
saw->genSaw();
Oscillator osc(saw);

```

6 - DownSampling (sotto-campionamento):

```

/* Main.cpp */
#include "AudioDromeAll.h"
using namespace Aud;

int main(int argc, char ** argv)
{
    AudioDrome::setAbsoluteSR(44100);
    AudioOut dac;
    Oscillator lfo;
    lfo.setAmplitude(0.5);
    lfo.setFrequency(3);
    lfo.setDownSampling(50);

    Oscillator osc;
    osc.setFrequency(440);

    ul P3 = AudioDrome::getAbsoluteSR() * 5;
    for (ul i = 0; i < P3; i++)
    {
        dac.PROCESSING();
        lfo.PROCESSING();
        osc.PROCESSING();
        osc.setAmplitude( lfo[LEFT] + 0.5 );
        dac << osc;
    }

    dac.abortStream();
    return true;
}

```

L'esempio sopra tratta una semplice AM (*Amplitude Modulation*). L'*lfo* (*Low Frequency Oscillator*) è sottocampionato 50 campioni, i vantaggi in termini di velocità di calcolo sono evidenti. Un procedimento più sofisticato interpola il segnale sotto-campionato, attraverso l'oggetto *UpSampling* che viene inserito nella catena. Questo riceve in ingresso l'uscita dell'*lfo* e restituisce il segnale interpolato progressivamente sul numero di sotto-campionamento:

```
UpSampling up(lfo);
```

Il costruttore di *UpSampling*, configura *up* sulle basi di *lfo* (oggetto da interpolare), il costruttore accetta "per riferimento" un *AudioDrome object*, o direttamente il valore di sotto-campionamento.

```

for (ul i = 0; i < P3; i++)
{
    dac.PROCESSING();
    lfo.PROCESSING();
    osc.PROCESSING();
    up.PROCESSING();

    up << lfo; //lfo è l'input di up
    osc.setAmplitude( up.Output() + 0.5 );
    dac << osc;
}

```

Il modulo d'interpolazione *UpSampling*, permette di spingere il sotto-campionamento a 500 senza degradazione del segnale, il sottocampionamento non si sottrae alla legge di Nyquist, se 44.1 Khz permettono fino 22050 Hz di banda, allora $44100/500 = 88.2/2 =$ permettono soltanto 44,1 Hz di banda.

PROGRAMMARE UNA NUOVA CLASSE

Il codice di *myClassTemplate* è il modello per programmare nuove classi nella gerarchia, può ereditare da *AudioDrome* o da una sottoclasse dipendentemente dal livello d'astrazione del nuovo DSP. Il modello include la funzione *myMemberFunction()* che sarà rinominata come funzione membro specifica per l'astrazione, a titolo d'esempio è stata contraddistinta come */*! Sampling Rate Change SLOT */* per indicare che utilizza, relativamente al DSP, la variabile *_localSR* e/o *getAbsoluteSR()*. La funzione è quindi chiamata da *srUpdate()* notificata dal processo "Sampling Rate is Changed". La classe appartiene al namespace *Aud*.

```
/*! \class myClassTemplate */
#include "AudioDrome.h"

//begin namespace
namespace Aud
{
    //inherits from AudioDrome or a sub-class
    class myClassTemplate : public AudioDrome
    {
    public:
        /*! Public constructor
        myClassTemplate();

        /*! Public destructor
        virtual ~myClassTemplate();

        /*! Sets or gets ...
        /*!
        * if you need to upgrade
        * the class variables,
        * dependent on _localSR
        */
        /*! Sampling Rate Change SLOT */
        void myMemberFunction(const AudFloat &val){}

        * You can read or write the sample
        * in the DSP with the virtual Output()
        * or Input() functions.
        * Also you can redefined.
        */
        virtual AudFloat Output();
        virtual AudFloat Output_2();

        virtual void Input(const AudFloat& in);
        virtual void Input_2(const AudFloat& in);

protected:
        /*! Processing the audio sample
        /*!
        * ...
        */
        virtual void Processing();

        virtual void srUpdate()
        {
            /*! Sampling Rate Changing SLOT */
            this->myMemberFunction();
        }
        //my class variables
    };
} //end of namespace Aud
```

Il modello di classe, deve considerare tre punti principali

- 1) La classe implementa la procedura *Processing()*
- 2) La classe implementa la procedura *srUpdate()*
- 3) Se contiene oggetti *AudioDrome*, ridefinisce la procedura *PROCESSING()*

Processing() è l'ingranaggio di calcolo locale del DSP, in realtà potrebbe non essere ri-definita, ereditando la *Processing()* della sotto-classe. La procedura *srUpdate()* dev'essere ridefinita solo se la nuova classe utilizza la variabile *_localSR* o *getAbsoluteSR()*, la *_localSR* è la frequenza di campionamento dell'oggetto, calcolata come *_absoluteSR / _downSampling*. In particolare è necessario ridefinire *PROCESSING()* nelle classi contenitrici, infatti la gestione del *downSampling* è responsabilità della classe contenitrice, per esempio la classe *Granulator* contiene istanze di *OscilTimeout* e *Envelops*, il sotto-campionamento della sola istanza di *Granulator* porta a conseguenze drammatiche. Nelle classi contenitrici di oggetti, il sotto-campionamento dev'essere unitario, *Granulator::Processing()* chiama le *PROCESSING()* di calcolo degli oggetti contenuti, questi non sono configurati correttamente è necessario quindi inoltrargli la richiesta di sottocampionamento. In questo caso però si determina un **sotto-campionamento doppio**, quello attuato dalla classe contenitrice e quello dagli oggetti contenuti. La soluzione è semplice: **si inoltra il sotto-campionamento a tutti gli oggetti contenuti, ma non si sotto-campiona la classe contenitrice**. Un caso più complesso è quando la classe contenitrice chiama le *PROCESSING()* delle istanze degli oggetti contenuti e la *Processing()* della classe da cui eredita. In questo caso, il sottocampionamento dovrà essere gestito inoltrando la richiesta anche alla sola classe genitrice.

```
/*! Sampling Rate Change SIGNAL */
virtual void setDownSampling(const ul& samples)
{
    AudioDrome::setDownSampling(samples);
    _trig->setDownSampling(samples);

    for (unsigned int i = 0; i < _overlap; i++)
    {
        _osc[i]->setDownSampling(samples);
        _env[i]->setDownSampling(samples);
    }
}
```

L'esempio estrapolato da *Granulator*, ridefinisce la funzione *setDownSampling()*, la procedura inoltra la richiesta di sotto-campionamento a tutti gli oggetti contenuti.

```
virtual void PROCESSING()
{
    this->Processing();
}
```

La classe ridefinisce anche *PROCESSING()*, al fine di scavalcare il controllo *downSampling* della *AudioDrome::PROCESSING()*, questa chiama semplicemente *this->Processing()*. Per eliminare l'*overhead* di chiamata di funzione, *this->Processing()* potrebbe direttamente essere definita come *this->PROCESSING()*.

In fase di sviluppo, ho scelto il *pattern design*, come meccanismo di notifica della "Sampling Rate is Changed", al posto della tecnica *Signal/Slots*, implementata dalle *Boost C++ Library*. Nel caso del *pattern design*, le procedure *srUpdate()* devono richiamare se stesse lungo il ramo ereditario fino alla "superclasse" infatti, ogni classe potrebbe aver bisogno di aggiornare le proprie variabili. Nel caso dei *Signal/Slots* i costruttori di tutte le classi devono registrare la propria procedura *srUpdate()*, inserendo in un *vector* il puntatore a funzione. Entrambe le tecniche funzionano correttamente, ma non sollevano l'utente programmatore che sviluppa una nuova classe, dalla responsabilità di **ricordarsi la corretta implementazione di un particolare**. L'attuale soluzione appare concettualmente corretta e non troppo complicata:

```
virtual void srUpdate()
{
    Oscillator::srUpdate();
    this->setTime(_dur);
}
```

l'esempio è implementato nella classe *OscilTimeout*, la procedura richiama la funzione membro *setTime()* e nello stesso tempo chiamata la *Oscillator::srUpdate()* della classe genitrice, la stessa cosa fanno *Oscillator* e *Phasor* risalendo il ramo ereditario.

UN PROGETTO VST

Per sviluppare un plugin VST è necessario procurarsi, dal sito ufficiale della **Steinberg**, il VST SDK (*Software Development Kit*). La compilazione del plugin dipende dalla piattaforma: in Windows è una DLL (*Dynamic Link Library*), in Mac OSX è una *bundle* con estensione *.vst*, ossia una cartella che contiene il file *plist.info* con la chiave *CFBundleName* per la definizione del nome. Su BeOS e SGI (MOTIF, UNIX) un plugin VST è una *shared Library*. Si rimanda alla documentazione in rete per studiare e approfondire l'argomento, in particolare segnalo il sito **www.parravicini.org** con la documentazione necessaria in italiano. Nel progetto si devono includere l'SDK VST e la *audiiodrome library* (*libaudiiodrome.a*) o direttamente i sorgenti, senza il supporto per l'audio in quanto gestito dall'*host*. Se la libreria è compilata in `__DOUBLE_PRECISION__`, è necessario ripetere la definizione al pre-processore nel progetto VST.

Dal sito www.alessandro-petrolati.it si può scaricare un *Xcode project* del seguente codice, si tratta di un *delay* stereofonico a interpolazione cubica:

```
// myPlugin.h
#include "AudioEffectX.h"
#include "AudioDromeAll.h"

using namespace Aud;

enum{
    // Global
    kNumPrograms = 1,
    // Parameters Tags
    kMaster=0, kDelayL, kDelayR, kNumParams
};

//! myPlugin estende la classe AudioEffectX
class myPlugin : public AudioEffectX
{
public:
    //! Costruttore pubblico
    myPlugin (audioMasterCallback audioMaster);

    //! Distruttore pubblico
    ~myPlugin(){};

    //! Implementazione della logica di Processing
    virtual void processReplacing (float **inputs,
                                  float **outputs,
                                  VstInt32 sampleFrames);

    //! Inoltra la Sampling Rate dell'Host
    //! agli oggetti AudioDrome
    /*!
     * La funzione è chiamata quando il tasso
     * di campionamento dell' Host cambia,
     * lavora in modo analogo alla srUpdate()
     * nell'architettura AudioDrome.
     */
    virtual void setSampleRate(float sampleRate);

    //! Chiamata quando cambia un parametro
    void setParameter(VstInt32 index, float value);
```

```
    //! Ritorna il valore del parametro
    virtual float getParameter(VstInt32 index);

    //! Imposta una stringa di rappresentazione
    //! del valore ("0.5", "-3", "PLATE", etc...)
    virtual void getParameterDisplay(VstInt32 indx,
                                      char* text);

    //! Imposta una stringa di rappresentazione del
    //! valore ("Time", "Gain", "RoomType", etc...)
    virtual void getParameterName(VstInt32 indx,
                                   char* text);

    //! Etichetta per il parametro
    //! ("sec", "dB", "type", etc...)
    virtual void getParameterLabel(VstInt32 indx,
                                    char* label);

protected:
    float fMaster;
    float fDelayL;
    float fDelayR;
private:
    DelayTaps delayL;
    DelayTaps delayR;
};
```

L'enumeratore (*enum*) è importante, imposta il numero di programmi in questo caso uno. I programmi sono i *presets* disponibili per il *plugin* anche se l'esempio non implementa i *presets*, il parametro è richiesto dal costruttore della classe genitrice *AudioEffectX* che stiamo estendendo. Le funzioni *sets* (virtuali) sono ridefinite perchè chiamate dall'*host*, l'enumeratore contiene la lista di tutti i parametri contemplati dal DSP, per ognuno è creato un controllo (*widget o slider*) nella GUI (*Graphic User Interface*) del *plugin*. *kNumParams* è passato quindi al costruttore di *AudioEffectX*, la classe deve contenere le variabili associate a ciascun controllo, in questo caso sono tre (*fMaster*, *fDelayL*, *fDelayR*) e gli oggetti DSP di *AudioDrome*. Di seguito l'implementazione delle funzioni della classe nel file *.cpp*:

```
// myPlugin.cpp
#include "myPlugin.h"

myPlugin::myPlugin(audioMasterCallback audioMaster) :
AudioEffectX(audioMaster, kNumPrograms, kNumParams)
{
    //default settings
    setParameter(kMaster, 0.5);
    setParameter(kDelayL, 0.3);
    setParameter(kDelayR, 0.6);
    setNumInputs(2); // stereo input
    setNumOutputs(2); // stereo output
    delayL.setInterpolation(CUBIC);
    delayR.setInterpolation(CUBIC);
}

la lista d'inizializzazione costruisce AudioEffectX, il costruttore di myPlugin configura lo stato iniziale dei controlli del plugin tramite le funzioni setParameter(), imposta inoltre il numero di canali I/O dell'host e il tipo d'interpolazione per gli AudioDrome objects.

void myPlugin::setSampleRate(float sampleRate)
{
    AudioEffectX::setSampleRate(sampleRate);
    AudioDrome::setAbsoluteSR( (AudFloat) sampleRate
)
}
```

In particolare si osservi la funzione *setSampleRate()*, chiamata dall'*host* quando cambia la SR, è ridefinita per inoltrare la richiesta a *AudioDrome::setAbsoluteSR()*, facendo scattare la

notifica “*Sampling Rate is Changed*”.

Le due funzioni che seguono sono l’I/O tra la classe DSP e la GUI del *plugin*:

```
void myPlugin::setParameter(VstInt32 index,
                            float value)
{
    switch (index)
    {
        case kMaster:
            fMaster = value;
            break;
        case kDelayL:
            fDelayL = value;
            delayL.setDelayTime(value);
            break;
        case kDelayR:
            fDelayR = value;
            delayR.setDelayTime(value);
            break;
    }
}

float myPlugin::getParameter(VstInt32 index)
{
    switch (index)
    {
        case kMaster:
            return fMaster;
        case kDelayL:
            return fDelayL;
        case kDelayR:
            return fDelayR;
    }
}
```

Quando dalla GUI si agisce sui controlli del *plugin*, l’*host* chiama *setParameter()* sul relativo indice di parametro, il costruito *switch* canalizza il valore sulla relativa variabile di classe e/o la passa direttamente agli oggetti *AudioDrome*. In realtà il meccanismo è più complesso: le funzioni *set* e *get*, sono chiamate di conseguenza, infatti è la seconda che aggiorna la posizione dello *slider* sul nuovo valore. Il *plugin* invia i dati al DSP attraverso *setParameter()*, che implicitamente invoca *getParameter()*, in altri termini *queste* sono funzione *callback mutex* (a mutua-esclusione) chiamate quando cambiano i valori dei parametri nella GUI.

```
void myPlugin::processReplacing (float** inputs,
                                float** outputs,
                                VstInt32 sampleFrames)
{
    float* in1 = inputs[0];
    float* in2 = inputs[1];
    float* out1 = outputs[0];
    float* out2 = outputs[1];

    while (--sampleFrames >= 0)
    {
        delayL.PROCESSING();
        delayR.PROCESSING();
        delayL.Input(*in1++);
        delayR.Input(*in2++);

        (*out1++) = (float) delayL.Output() * fMaster;
        (*out2++) = (float) delayR.Output() * fMaster;
    }
}
```

La funzione *processReplacing()* è una classica *callBack*, si rimanda al **Tutorial** di questo articolo per chiarimenti. La funzione *main*, per i *plugins* VST è sostituita dalla seguente inclusa nel file *myPlugin_Main.cpp*:

```
// myPlugin_Main.cpp
#include "myPlugin.h"

AudioEffect* createEffectInstance(audioMasterCallback
                                  audioMaster)
{
    return new myPlugin(audioMaster);
}
```

ATTENZIONE, la documentazione VST assume che: *"All parameters - the user parameters, acting directly or indirectly on that data, as automated by the host, are 32 bit floating-point data. They must always range from 0.0 to 1.0 inclusive [0.0, +1.0], regardless of their internal or external representation."* Per mappare i parametri su valori diversi, è necessario implementare opportune funzioni, strutture dati o classi.

RIFERIMENTI

1. Curtis Roads. Computer Music Tutorial, The MIT Massachusetts Institute of Technology, 1996
2. B. Stroustrup. The C++ Programming Language. Addison-Wesley, Reading, Massachusetts, 2000
3. Bruce Eckel, Thinking in C++ Volume One (2nd Edition)
4. Carlo Pescio, C++, Manuale di Stile, Infomedia, 1995
5. Paolo Marotta , C++, una panoramica sul linguaggio Copyright 1996-1999, <http://www.tutorialpc.it/c++menu.asp>
6. Cipriani/Bianchini, Il suono Virtuale, ConTempo,1998
7. R. Bencina and P. Burk, PortAudio - an Open Source Cross Platform Audio API. 2007 <http://www.portaudio.com>
8. The Synthesis ToolKit in C++ (STK) Perry R. Cook & Gary P. Scavone Music Technology, Faculty of Music McGill University/Departments of Computer Science & Music <http://ccrma.stanford.edu/software/stk/>
9. The Sound Object Library (SndObj) Victor Lazzarini Music Technology Laboratory National University of Ireland, Maynooth <http://music.nuim.ie//musictec/SndObj/main.html>
10. Pattern Design, GoF [http://en.wikipedia.org/wiki/Design_pattern_\(computer_science\)](http://en.wikipedia.org/wiki/Design_pattern_(computer_science))
11. Steinberg 3rd party developer support http://www.steinberg.net/en/company/3rd_party_developer.html
12. Boost C++ Library http://www.boost.org/doc/libs/1_36_0/libs/libraries.htm
13. DoxyGen Source code documentation generator tool <http://www.doxygen.org>
14. Emanuele Parravicini, VST plugin - Getting Started <http://www.parravicini.org>
15. Eugenio Giordani LEMS (Laboratorio per la Musica Sperimentale), Conservatorio G. Rossini di Pesaro. <http://www.eugenio.giordani.it>