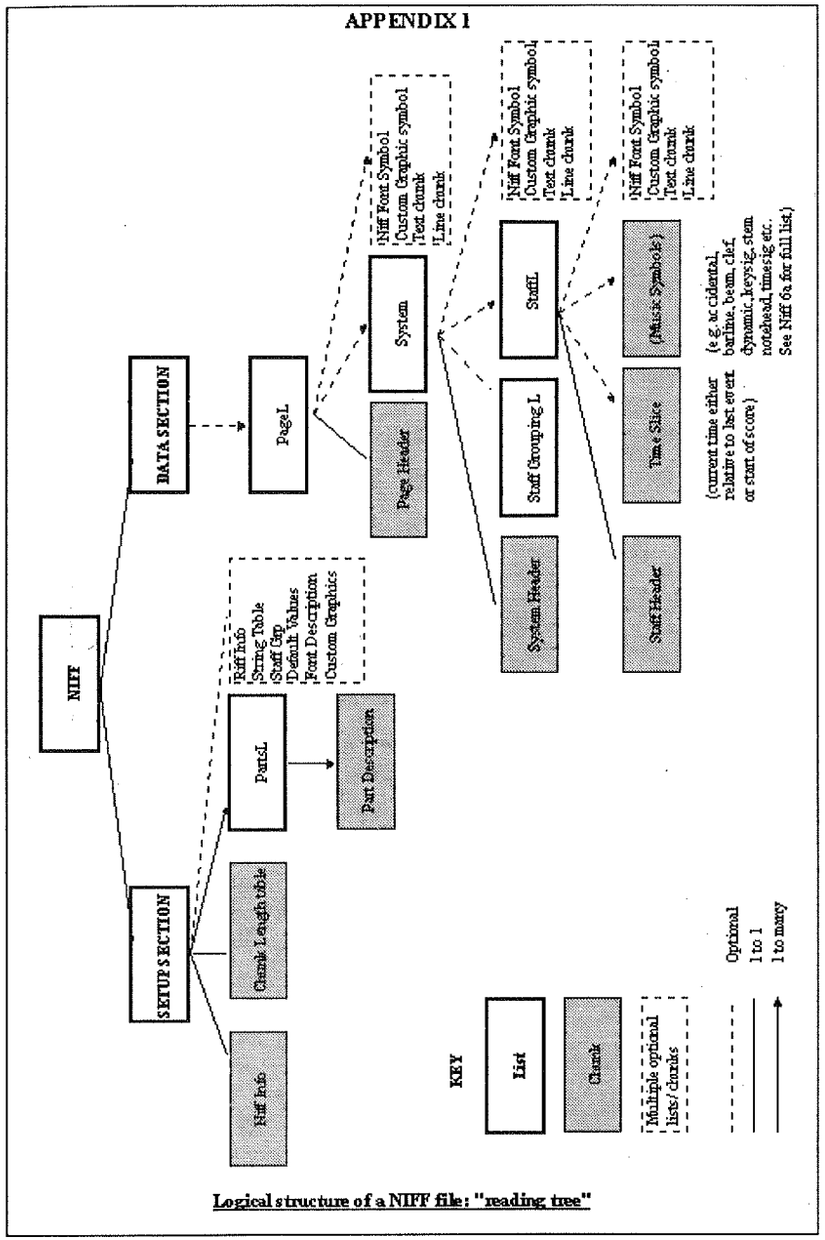


Logical structure of a NIFF file: "reading tree"

Designing a Sound Object Library

Victor Lazzarini e Fernando Accorsi

Núcleo de Música Contemporânea
Departamento de Arte / Departamento de Ciência da Computação
Universidade Estadual de Londrina

Abstract

The Sound Object Library is being designed to be employed by programmers and composers to develop sound manipulation applications. Its classes encapsulate all the processes involved in sound creation, manipulation and storage. The library can be used as a framework or, alternatively, as a set of objects to be patched together in a program. It is being developed to be portable across the UNIX and Windows platforms. The library class hierarchy is founded on three base classes, corresponding to sound processing objects, maths function-tables and sound input/output objects. A number of classes have already been implemented. A programming example shows one of the possible applications of the library objects. A GUI interface for the objects is proposed, using the V C++ framework.

1 Introduction

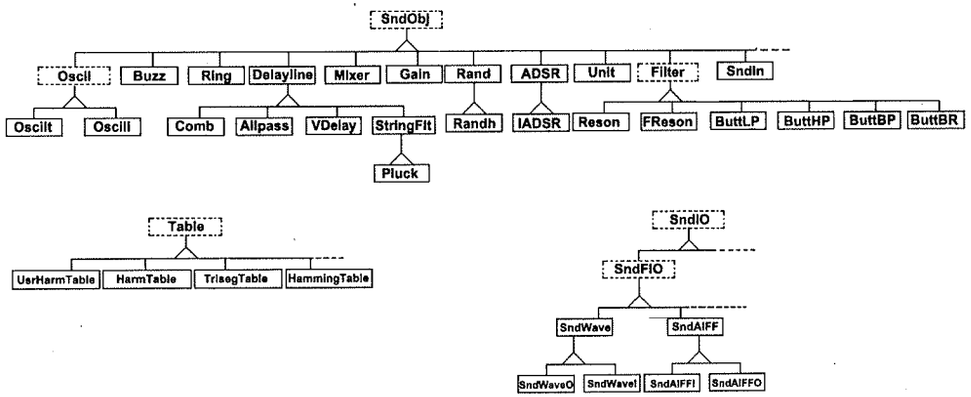
The research described here derived from some ideas that evolved during the work on the software *Audio Workshop* (Lazzarini, 1998). In that process, it was noted that the development of a set of objects for audio signal processing could be very useful in the design of new programs. It would help provide higher level tools for audio programming and software with a more intuitive user interface. This set would be used in two ways: as finished objects, to be employed directly in a program (or in a visual patching application), and as a framework, to which developers could add their own specialized code. Consequently, programs to carry out specific tasks would be easily developed by connecting the available objects and providing standard control-of-flow. Also, a text-based sound compiler could be designed to make use of the library. Using derivation and inheritance, new objects could be created from the existing ones, thus leaving the development possibilities open. The creation of a Sound Object Library seemed a worthwhile, though complex, task to pursue.

The project took shape as some important premises were being defined. Firstly, the library objects should encapsulate all the processes involved with production, manipulation and storage

of audio data. Secondly, the user would have to be able to freely associate the objects, as if they were modules in an analog synthesizer, or opcodes in systems such as, Csound (Vercoe, 1992) and Cmusic (Moore, 1990). Finally, the core of the code should be portable, allowing for machine-dependent specialisations when necessary. This, together with other practical reasons, lead to the choice of a portable high-level language such as, C++ (Stroustroup, 1991), with compilers available for a great number of machines. The project has been implemented, initially, on two UNIX platforms, Sun SparcServer and ULTRASparc under Solaris, and IBM RISC 2000 under AIX, as well as on Pentium PCs under Windows 95 and Windows NT. Because of its availability across all the target platforms, including the Windows operating systems, the chosen development environment was that of the Gnu Project, including g++, gdb and the Emacs editor. In the Windows platforms, MS Visual C++ was also used, since it compiles to an object code different to that of the Cygnus Windows Gnu Project (cygwin) c++ compiler. The programs created with the cygwin tools also require a special DLL to run, although their performance seems to be acceptable. An alternative port of the gnu compiler for the win32 operating system was also used. It is provided by the Minimalist Gnu Win32 (Mingwin32) tools, which do not require a runtime DLL and therefore have a better performance.

2 The Sound Object Library hierarchy

The proposed hierarchy for the library is based on three abstract base classes: SndObj, Table and SndIO. These form the base for three types of objects that integrate the set, respectively: sound processing, maths function-table and sound input/output objects. A diagram showing the inheritance relationships (Rumbaugh *et al*, 1994) between the classes in the library is shown below:



The classes derived from SndObj are involved in the production and manipulation of sound samples and they can make use of the Table classes. The SndIO tree is dedicated to all the processes involving sound input or output: disk, ADC/DAC, screen printing, etc.. These classes use a SndObj as their input and a SndIn class (derived from SndObj) can receive a SndIO-derived input, completing the link with the real world. In order to provide sound buffering services, a circular buffer class, SndBuff, was implemented and is used by the SndIO-derived classes.

2.1 The SndObj classes

A brief description of the interface for the SndObj base class is shown below:

```
class SndObj {
        protected:
      float* output; // output samples buffer
      float  m_sr;      // sampling rate
      int    m_error;   // run-time error code
      short  m_enable;  // object status (default ENABLED)

        public:
      void   Enable();  // enable object
      void   Disable(); // disable object
      float  GetSr();   // get the sampling rate
      void   SetSr(float sr); // set the sampling rate
      float  Output();  // get the current output sample
      virtual char* ErrorMessage(); // error messages
      virtual short DoProcess();    // processing
            };
```

The sound objects have one output, a sampling rate, an on/off switch and an error code. They also have a `DoProcess()` function that carries out all the due processing of a particular object and other methods to access its member variables. The derived classes have an undefined number of inputs (objects and/or set values) and some of them also rely on maths function-table objects to do their processing having one or more members of the Table type. Some examples of SndObj classes already implemented are: Oscilt and Oscili, two types of oscillators; Mixer, a sound object mixer; Gain, a gain modifier; Rand and Randh, two types of noise generators. Many other classes, such as envelope shapers, filters and delay-line objects were also implemented and some spectral-domain processing objects are being developed. The sound processing objects are designed to be easily interconnected, as it will be shown in the programming example. Most of them can receive one or more SndObj inputs. For instance, Oscillators can receive any sound object as their

frequency and amplitude inputs. Filters receive SndObjs as their audio input, as well as their frequency and bandwidth inputs. Mixers can receive any number of input objects and mix them together. The ability to make patches of processing boxes gives the flexibility necessary for users to create a great number of applications. As an example of a SndObj-derived class, an abbreviated definition of the Mixer class is shown below:

```
class Mixer : public SndObj{
        protected:
      SndObjList*  m_InObj;  // list of input SndObj
      int          m_ObjNo;  // number of input objects

        public:
      Mixer();   // constructors and destructor
      Mixer(int ObjNo, SndObj** InObjs);
      ~Mixer();

      int GetObjNo(); // return number of inputs
          short AddObj(SndObj* InObj); // add an SndObj to the
                                       // input list
      virtual short DoProcess();   // mixing operation
      virtual char* ErrorMessage(); // error messages
                        };
```

This object can receive any number of SndObjs as its inputs, adding them to an input list. The mixing is done by summing the output samples of the input objects in the `DoProcess()` method. An instance of the Mixer class can be created either by passing the number of initial SndObj inputs and an array of SndObj pointers or as an empty object whose inputs are going to be set later by the `AddObj(...)` method.

2.2 Tables

The table classes were developed to supply certain sound objects with tabulated maths functions. A common use for them is to provide one cycle of a certain waveshape to be continuously sampled by an oscillator. The HarmTable is an example of such an object that creates any of the following four harmonic waveforms: sine, saw, square or pulse (buzz). It can be used by an oscillator to create a pitched sound of a particular timbre. Similar to that, is the UsrHarmTable, which allows the user to define the relative amplitude of the individual harmonics. Another common type of function table is to store a shape to be used as an amplitude or frequency envelope. The TrisegTable class creates a three-segment line, with the option of logarithmic or linear lines, that can be used by an oscillator to control a parameter of some other sound object. Also, a

generalized Hamming window function table is supplied, as the HammingTable object. Table objects are very useful and can be employed in a variety of ways and will be constantly added to the library implementation.

2.3 Input and Output

The SndIO classes are designed to deal with all the input and output services need by the sound objects. They provide the `Read()` and `Write()` methods that will be used to perform the IO functions, regardless of whether the target is a soundfile, the ADC/DAC, the computer screen or some other device. The derived classes are designed to fit into these main categories: soundfile IO, which will have derived classes for each format, DAC/ADC IO and screen output, both having derived classes that will be platform-dependent. The resumed description of the interface for the SndIO class is shown below:

```
class SndIO {
        protected:
      float* output;   // output sample buffer
      SndBuffer* m_SndBuff; // internal buffer
      float m_sr;          // sampling rate
      int m_error;         // run-time error code

        public:
      float GetSr();      // get the sampling rate
      float Output(int channel);// get the current output sample
                                // for the specified audio
                                // channel
      virtual short Read();     // read from device
      virtual short Write();      // write to device
      virtual char* ErrorMessage(); // error message
        };
```

Some classes of the input and output tree were initially implemented. SndFIO is a SndIO-derived abstract class that controls file I/O, from which other classes, dealing with specific soundfile formats, are derived, like SndWave and SndAiff. Two classes are derived from SndWave, SndWaveI and SndWaveO. They perform RIFF-Wave file input and output, respectively. SndAiffI and SndAiffO are SndAiff-derived classes that deal with AIFF soundfile format. A special sound object, SndIn, can receive one-channel sound input of SndIO-derived object. Its output can be used by any SndObj-derived object in the processing chain. For multichannel input, multiple SndIn objects can be used.

3 Programming example

The following application uses some of the Sound Object classes to simulate the ingenious Jean-Claude Risset design (Dodge & Jerse, 1985). The output sound is a cascading harmonics drone, created by the minute differences in frequency of the nine oscillators employed. This programme, named *Risset*, can be called with the following command line:

```
Risset filename.wav duration(s) fr(Hz) amp(dB) no_of_harmonics
```

This application consists basically of a `main()` function that uses the Sound Object Library classes. It uses four types of sound objects: `Oscilt`, a truncating oscillator; `Oscili`, an interpolating oscillator; `Gain`, a gain control; and `Mixer`, a sound object mixer. The code can be divided in two parts: the creation and patching of the objects and the synthesis loop. In the first, all the boxes are set and connected: the mixer receives all the nine interpolating oscillators as inputs, which in turn receive the truncating oscillator as their amplitude input. The `SndWaveO` sound output receives the gain object, which attenuates the mixer signal. The synthesis loop is simply based on the ordered calls to the `DoProcess()` methods of the respective objects, followed by a call to the soundfile output `Write()` member function. The complete code (with the exception of the `usage()` function) for the Risset program is shown below.

```
//*******************************************************//
//      RISSET.CPP                                       //
//      Sample application using Sound Object Classes    //
//      synthesizes a cascading harmonics drone, as      //
//      designed by J C Risset                           //
//                  Victor Lazzarini, 1997               //
//*******************************************************//
#include <iostream.h>
#include <stdlib.h>
#include "AudioDefs.h"
void usage();

int
main(int argc, char* argv[]){
    if(argc != 6){
    usage();  // usage message
    return 0;
             }

// command line arguments
float fr = (float)atof(argv[3]);          // frequency
```



```
float amp = (float)atof(argv[4]);         // amplitude
float duration = (float)atof(argv[2]);    // duration.

// Envelope breakpoints & function table object
float TSPoints[7] = {.0f, .05f, 1.f, .85f , .8f, .1f, .5f};
TrisegTable envtable(512, TSPoints, LINEAR);

// Wavetable object
HarmTable table1(1024, atoi(argv[5]), SQUARE);

// truncating oscillator object (envelope)
Oscilt envoscil(&envtable, 1/duration, 32767);

// 9 interpolating oscillator objects
Oscili oscil1(&table1, fr, 0.f, 0, &envoscil);
Oscili oscil2(&table1, fr-(fr*.03f/110), 0.f, 0, &envoscil);
Oscili oscil3(&table1, fr-(fr*.06f/110), 0.f, 0, &envoscil);
Oscili oscil4(&table1, fr-(fr*.09f/110), 0.f, 0, &envoscil);
Oscili oscil5(&table1, fr-(fr*.12f/110), 0.f, 0, &envoscil);
Oscili oscil6(&table1, fr+(fr*.03f/110), 0.f, 0, &envoscil);
Oscili oscil7(&table1, fr+(fr*.06f/110), 0.f, 0, &envoscil);
Oscili oscil8(&table1, fr+(fr*.09f/110), 0.f, 0, &envoscil);
Oscili oscil9(&table1, fr+(fr*.12f/110), 0.f, 0, &envoscil);

// Mixer & gain objects
Mixer  mix;
Gain   gain((amp-15.f), &mix);

//  Add the oscili objects to the mixer input
mix.AddObj(&oscil1);
mix.AddObj(&oscil2);
mix.AddObj(&oscil3);
mix.AddObj(&oscil4);
mix.AddObj(&oscil5);
mix.AddObj(&oscil6);
mix.AddObj(&oscil7);
mix.AddObj(&oscil8);
mix.AddObj(&oscil9);

// output to a RIFF-Wave file
SndWaveO  output(argv[1], 1, 16, OVERWRITE);
output.SetOutput(1, &gain);

// synthesis loop
unsigned long dur =(unsigned long)(duration*envoscil.GetSr());
```

```
for(unsigned long n=0; n < dur; n++){

    envoscil.DoProcess(); // envelope
    oscil1.DoProcess();   // oscillators
    oscil2.DoProcess();
    oscil3.DoProcess();
    oscil4.DoProcess();
    oscil5.DoProcess();
    oscil6.DoProcess();
    oscil7.DoProcess();
    oscil8.DoProcess();
    oscil9.DoProcess();
    mix.DoProcess();      // mix
    gain.DoProcess();     // gain attenuation
    output.Write();         // file output
                              }
  return 1;
}
```

Similarly, a user with some knowledge of C/C++ can easily build signal processing applications, just by writing his/her own `main()` functions and patching together the library objects. In addition, an experienced programmer would be able to extend the library by adding his/her own processing objects.

4 Sample applications

Together with the development of the SndObj library, a number of sample applications were developed with a twofold purpose: as a set of programming examples and as a group of useful programs to be used in computer music composition. They include command-line cutting, splicing and mixing programs, multi-purpose filters, reverbs, synthesizers and other utilities. It is expected that they will form a comprehensive set of tools for composers of electroacoustic and computer music. The technical documentation of the SndObj library will include the complete code for all the sample applications.

5 Graphical user interface

Studies are being carried out to couple the development of the library objects with a graphical user interface (GUI) using the V C++ framework (Wampler, 1996). V is a portable C++ GUI library developed for X-Windows and MS-Windows environments. Because of its portability across the platforms used in this research project, it has been considered for the implementation of visual counterparts to the objects of the Sound Object Library. These would be processing boxes that would provide an intuitive interface to the C++ code. The initial idea is to develop a patching

application that will let the user play with the sound objects to create his/her own signal processing module, by interconnecting the available boxes. At the present moment, some sample visual applications were developed using V and the SndObj library, running under MS-Windows and X-Windows (AIX and Solaris). These are intended as test applications for the development of a portable GUI. This would provide an intuitive use of the library objects and a powerful tool for composers of electroacoustic and computer music.

6 Conclusion and further research

This paper described the development of a set of objects for sound manipulation in the computer. The Sound Object Library is being designed to be a comprehensive multi-platform set of C++ classes to be used by programmers and composers. It has a modular design, akin to analog synthesizers and computer music systems, and simple connectivity. All the processes involved in the sound production, manipulation and storage are encapsulated by the library classes. Research is continuously being carried out in the development of new objects, including spectral analysis and resynthesis. The development of a GUI for the library, using the multi-platform V framework, is also under study. A beta-version of the first release of the SndObj library, together with command-line and graphic sample applications, will soon be available for download at the internet site http://www.dcop.uel.br.

Acknowledgments

The authors would like to thank CNPq for financial support of the present research and the Department of Computer Science (UEL) for granting the use of the IBM-RISC lab and the ULTRAsparc workstation.

Bibliography

Dodge, C & Jerse, T (1985). Computer Music: Synthesis, Composition and Performance. Schirmer Books, New York.

Lazzarini, VEP (1998). "A Proposed Design for an Audio Processing System ". *Organised Sound* 3 (1). Cambridge Univ. Press, Cambridge.

Moore, FR (1990). *Elements of Computer Music*. Prentice-Hall, Englewood Cliffs.

Stroustrup, B (1991). *The C++ Programming Language.* Addison-Wesley Publishing Co., Reading, Mass..

Rumbaugh, J ; Blaha, M; Premerlani, W; Eddy, F; Lorensen, W (1994). *Modelagem e Projetos Baseados em Objetos*, Editora Campus, Rio de Janeiro.

Vercoe, B (1992). *Csound, A Manual for the Audio Processing System.* MIT, Cambridge, Mass..

Wampler, B (1996). *V Reference Manual.* HTML document. http://www.cs.unm.edu/~wampler/vwebref.html