SOUND PROCESSING WITH THE SndObj LIBRARY: AN OVERVIEW

Victor Lazzarini

Music Technology Laboratory
Music Department
National University of Ireland, Maynooth
Maynooth Co. Kildare
Ireland
e-mail: Victor.Lazzarini@may.ie

ABSTRACT

This paper presents an overview of the sound processing applications of the Sound Object (SndObj) Library version 2.0. The SndObj library is an object-oriented sound synthesis and processing programming library. It is designed for the development of music applications, as well as research and implementation of DSP algorithms. The paper includes a brief but comprehensive description of the library class trees. It also presents the new features and changes introduced in the latest version of the library. A discussion of sound processing and synthesis programs, including some code examples, completes this article.

1. THE LIBRARY

The SndObj Library is a C++ object-oriented library. It was developed to assist the development of audio synthesis and processing software on general-purpose computers [3][5]. It comprises a series of signal processing and control classes which can be used in a number of signal processing applications. The library can be employed directly in music software applications, as a toolkit [7] or serve as a framework for the development and implementation of new sound processing algorithms. It has been developed with a great number of applications in mind, from the implementation of text-based or sound processing systems to DSP research and development.

The SndObj library has three important characteristics. The first one is encapsulation: the library classes encapsulate all the processes involved with production, control, manipulation, storage and performance of audio data. The second is modularity: processing objects can be freely associated, as if they were modules in an analogue synthesizer, or unit generators in a computer music system, as described in [2] and [6], each one performing specific functions. The final characteristic is portability: the core of the code is portable to any platform, requiring simply a (POSIX compliant) C++ compiler. The library also allows for machine-dependent specialization when necessary; for example, the realtime IO classes, dealing with specific DAC/ADC, are platform-specific.

Parts of this project have been developed on a number of different platforms, Sun Sparc under Solaris, IBM RISC 2000 under AIX, SGI O2 under IRIX, as well as on Intel PC under Linux and Windows (under Cygwin/Gnu g++). At the moment, the latest (beta) versions of the library have been released for Windows, Linux and IRIX operating system. The latest version of the library, discussed in this article, is licensed under the GNU Public License and it is available for download from http://www.may.ie/music/musictec.

2. CLASS HIERARCHY

The SndObj library v.2.0 is based on four base classes: SndObj, SndIO, Table and SndThread. The first three are basic models for types of objects involved directly in sound processing tasks, respectively: signal processing, signal input/output and mathematical function-tables. The fourth base class is a new addition to the library, introduced in its latest version. This class tree, for which only one class has been developed so far, is designed to supply thread (and in the future, process) management to the library.

2.1. SndObj classes

Objects of the SndObj class share some basic properties, such as the sampling rate, output vector size, an output buffer (for the processed signal), a SndObj input object (which provides the signal to be processed) and an on/off switch, as shown in **fig. 1**:

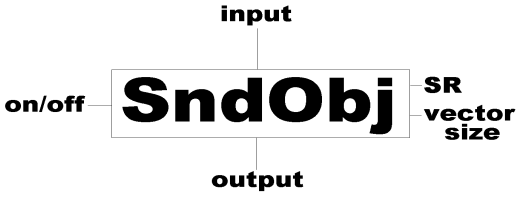

Figure 1. *The model for a SndObj object*

These objects will also share methods for the basic operations, such as addition, subtraction and multiplication (new on version 2.0), as well methods for setting and retrieving their basic attributes. These include a method for retrieving samples from the output buffer, which plays a very important part in the connectivity of the objects.

The SndObj classes also include a main processing method, `DoProcess()`, which is overridable. This is coded differently on each derived class, according to the processing algorithm implemented by the class. Typically, this method will access the

output signal from an input object and perform its processing, which will eventually fill the output buffer. For instance, the SndObj base class copies the signal from an input object to the output buffer. This is not in itself a very interesting operation and objects of the SndObj class will seldom be used (although there are some situations where they can be useful). Nevertheless, the importance of the SndObj class is that it provides a working mechanism for its derived classes.

Since objects of these classes will typically process a vector_size group of samples each time they are called, their `DoProcess()` methods are usually placed on a loop of some sort. Earlier versions of the library implemented the output on a single-sample basis. This permitted some useful tricks, such as multiple sampling rates for control purposes, etc., as well as a more simpler implementation of the processing methods. Nevertheless, as the use of a vectorial processing is usually more efficient, this was modified in the latest version of the library. As it turns out, the processing method implemented permits the use of dual sampling rates (i.e. audio and control), which is usually sufficient if extra performance is needed.

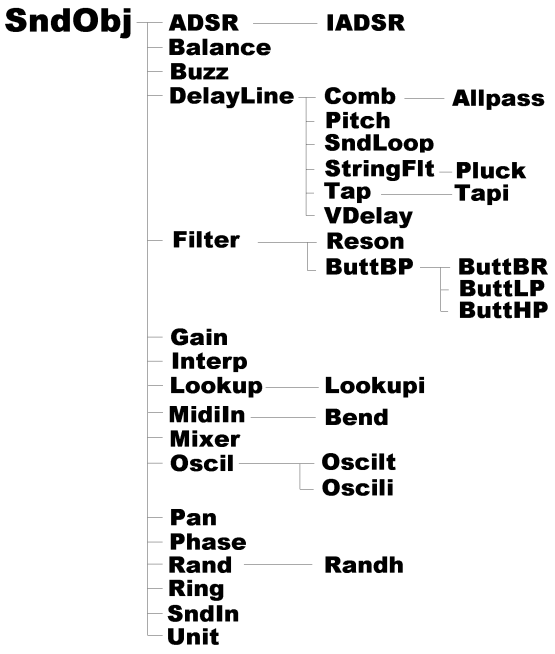

Figure 2. *The SndObj class tree.*

Input signals to SndObj objects, as hinted before, are obtained from input objects. Details of implementations are beyond the scope of this paper, but can be found in the documentation (available online at the download site) and in [3]. The base class contains one such input, which is the signal to be modified. Some of the derived classes will not use signal from an input object, as in the case of certain signal generators. Others will implement more than one signal input (used for controlling parameters such as a variable frequency or delaytime input). There is freedom within the framework for any number of inputs. For a very large, or an unspecified, number of inputs, classes will usually implement these as linked lists of objects (the Mixer class is a typical example of this). Processing parameters can also be controlled by constant values (as opposed to signal vectors), using the `Set..()` methods provided by the different classes. **Fig. 2** shows the SndObj class tree.

2.2. SndIO classes

Objects of the SndIO class tree (**fig. 3**) are designed to deal with input and output of audio. They implement five basic tasks: standard IO (stdin/stdout/stderr), soundfile IO, digital-to-analog / analog-to-digital converter IO, buffer memory (RAM) IO and Musical Instrument Digital Interface (MIDI) input. Other types of IO can also be implemented under this tree (such as network streams, screen graphics output, gesture control input, etc.).

The SndIO base class implements a very simple standard IO. As with SndObj, the importance of this class is that it sets the basic mechanism for input/output. The most important methods of this class are the overridable `Read()` and `Write()` functions. These, as their names suggest, perform the reading and writing tasks associated with IO operations. As in the classes of the SndObj tree, the derived classes of SndIO will implement these methods according to what operations they are supposed to perform. These methods also operate on the same vectorial basis as the SndObj processing functions, so they should also be placed accordingly on a loop. Because of the nature of their operation, certain classes, like the realtime audio ones, will also implement other secondary software and hardware buffers (as well as the ones used for the output vectors), depending on the platform.

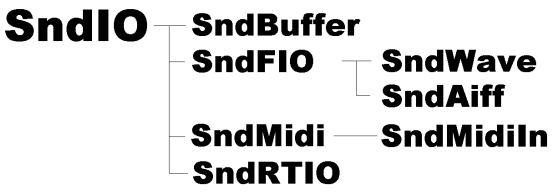

Figure 3 *The SndIO class tree*

The SndIO classes connect with SndObj classes in two ways: they can receive input from SndObj objects (which will be sent to whatever output they implement) and they can send their signal to special SndObj classes (such as SndIn, MidiIn and Bend) which can then be used in the processing chain. There are also two operators (<< and >>) defined within SndObj which can be used to receive and send audio signals from/to SndIO objects.

As expected, some SndIO classes are platform dependent, which is the case of SndRTIO and SndMidi/SndMidiIn. These classes are implemented on three platforms only (Linux/OSS, SGI (Irix) and Windows/MME) and are the exception on what is otherwise a fully cross-platform library. The SndRTIO class is now the sole realtime audio class in the latest version of the library, taking the place of SndOssRTI/O, SndSgiRTI/O and SndWinRTI/O. These classes were all consolidated in one, resulting in a more easily portable code. There are still some minor differences in the syntax of the declarations for these classes on the three platforms, but in general programs can be written which can largely ignore the differences.

Other changes introduced in version 2.0 include the consolidation of separate input and output classes into a single class and a rationalization of the class tree. All base classes were transformed

into instantiable ones, which eliminated a number of intermediate abstract classes. There was also a (well overdue) revision of the code, especially of the soundfile classes, which increased the efficiency and performance of objects.

2.3. Table classes

The classes found in the Table class tree are basically designed to generate utility objects for use in the sound processing algorithms. Obvious examples of mathematical function tables are found on table-lookup oscillators and other similar algorithms. These classes implement a series of useful tabulated functions, from harmonic ones (HarmTable) to generalised windows (HammingTable) and conversion tables (NoteTable). There are also tables which can hold audio signals (SndTable) for sampling and other similar applications.

2.4. The SndThread class

The SndThread class (new in version 2.0) has been developed to implement some experimental thread-management tasks. This class encapsulates the main processing loop as a separate POSIX pthread [1]. Typically, applications implemented with this class will instantiate objects of SndObj, SndIO and Table classes, and one (or more) object(s) of the SndThread class. The processing objects will then be passed to the SndThread class which will then control the synthesis/processing operations.

The SndThread class has three basic attributes: two lists of SndIO objects (for input and output, respectively) and a list of SndObj objects. Methods are provided for adding (to the top of the list), inserting (to any position in the list) and deleting objects (from the list). At the moment, this class does not implement any sorting algorithm to position objects in the right order for processing, as it is expected that the application using SndThread objects will take care of this task. A means of sorting objects will probably feature in later versions.

The processing control implemented by SndThread is very straightforward. The methods `ProcOn()` and `ProcOff` are used to switch the processing thread on/off. Since it runs on a secondary thread, control is always returned to the calling main program. The example below shows how a typical SndThread-based application works:

```
// SndObj (or derived) objects instantiated
SndObj obj1(…);
SndObj obj2(…);

// ibid. SndIO
SndIO  input(…);
SndIO  output(…);

// ibid. SndThread
SndThread  process;

// Add the objects to the thread
process.AddObj(&obj1);
process.AddObj(&obj2);
process.AddObj(&input, SNDIO_INPUT);
process.AddObj(&output, SNDIO_OUTPUT);

// Start the processing
process.ProcOn();
(…)

// Finish the processing
process.ProcOff();
```

Parallel processing can be implemented with the use of more than one thread class and synchronization can be achieved by the use of SndBuffer objects. As this is an experimental class, it is expected that more complex types of synchronization will be implemented, as well as a more powerful parallel processing support.

3. APPLICATIONS

The SndObj library is distributed with a number of examples of sound processing applications, as well as templates for the development of new classes. These programs are designed to present in a simple way the use of SndObj objects in signal processing and they do not include any GUI code which could obscure the examples given. This section will discuss this applications in more detail. A number of useful graphical applications have also been developed with the library, in conjunction with a commercial application framework for the Windows platform. Some of these applications are also available for download at the site mentioned in the first section. An example of this type of programs is presented later in this paper.

3.1. A signal processing example: cvoc

The **cvoc** program implements a simple filter-based vocoder, based on Butterworth filters. It analyses a signal with any number of filters, extracting its spectral envelope, which is then applied to a second signal, using the same number of parallel bands. The signal flowchart for one of the analysis-synhesis pair that makes up a band is shown on fig.4. The program allocates any number of these objects, according to the user's specifications.

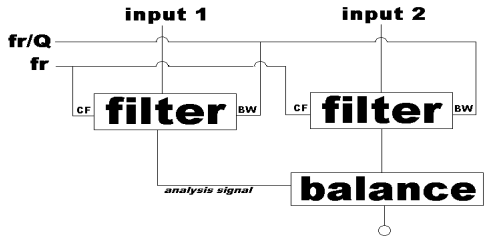

Figure 4 *An analysis band of cvoc*

This application presents a working example of soundfile input/output, filter, balance and mixer objects. It also shows how connections between objects are effected. For instance, one of the soundfile inputs is performed by a SndWave object called `*input1;` a SndIn object receives this input, making it available to SndObj objects. It passes this signal to all of the analysis filters (which are declared in an array). This, as well as the setting of the filter parameters, is shown in the C++ code fragment below (`nfltrs` is the number of bands used):

```
SndIn sound1(input1, 1); // audio input ButtBP*
Infilters = new ButtBP[nfltrs];
(…)
for(int i=0;i<nfltrs; i++){
Infilters[i].SetFreq(fr[i]);
```

```
Infilters[i].SetBW(fr[i]/Q);
Infilters[i].SetInput(&sound1);
(…)
}
```

From this code, the connection mechanism used in the library is clear. Pointers to input objects are used as a way of setting the signal chain. Similar code is used to set the other filter, balance, etc. objects. The processing part of the code is implemented by placing calls to the `DoProcess()` (and `Read()/Write()`) methods of the objects being used.

3.2. MIDI control

Some of the examples are provided to introduce the implementation of MIDI control by the library. This is the case of **midisynth** and **pluck** programs, monophonic MIDI-controlled synthesis applications. The former is a variation on the **risset** program, which is explored in detail on [3]; in this paper we will be discussing aspects of the latter. Pluck is a Karplus-Strong-based plucked-string synthesizer, implemented using an object of the Pluck class. The frequency of the plucked tone is controlled by the MIDI note number and pitchbend value, as shown in fig. 5. MIDI input is performed by a SndMidiIn object (*named* `midi`). It listens to all channel messages and makes them available at its output. A MidiIn (`note`) object is used to parse NOTE messages and output note numbers. Objects of this class can be set to parse any channel messages. Its output is used by a table-lookup (`conversion`) object which converts the note numbers into equal-tempered frequency values (using a NoteTable conversion table, `ptable`). A Bend object (`pitch`), also connected to the SndMidiIn object, is used to 'bend' these frequency values up or down in an specified range (2 semitones).

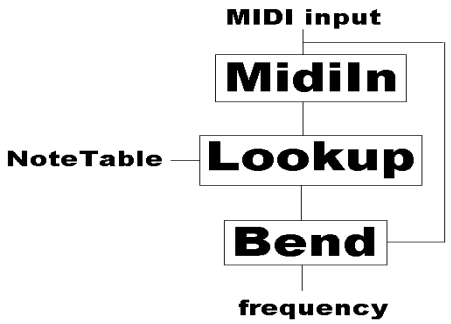

Figure 5. *MIDI control of frequency in pluck*

```
SndMidiIn* midi = new SndMidiIn(port, 10); MidiIn*
note = new MidiIn(midi);
Lookup* conversion = new Lookup(&ptable, 0,
note);
Bend* pitch = new Bend(conversion, midi, 2);
```

The signal out of the Bend object is then used to control the frequency of a Pluck. This outputs a full-amplitude signal which is then attenuated by a Gain object. This object will be controlled by the MIDI velocity value detected by `midi`.

```
Pluck* pluck = new Pluck(0.f, 32767.f, res,
pitch);
Gain* gain = new Gain(-90.f, pluck);
```

The processing loop is implemented as before, except that, because of the need to detect MIDI notes and velocities, some extra code elements will have to be included in it. The first is to detect whether a note has been switched off (`cur_note` is the current note number):

```
if(cur_note == midi->NoteOff())
             gain->SetGain(-90.f);
```

It is also necessary to detect a new note, re-initialise the pluck algorithm (so that a new onset is generated) and set the amplitude to correspond (more or less) to the note velocity.

```
if(midi->NoteOn() != -1){
cur_note = midi->LastNote();
pluck->RePluck();
gain->SetGain(midi->Velocity(cur_note) -
127.f);
}
```

Both **midisynth** and **pluck** send output to a DAC device. This is, together with the MIDI functionality, implemented differently on the three realtime platforms (Linux/OSS, Irix, Windows). Nevertheless, The interface for these objects is very similar, except for a few minor parameters. For instance, the MIDI 'port' on Windows is an integer identifier, whereas on the SGI and Linux, it is a name string relative to the device to be open. Similar differences are found on the SndRTIO class declarations. These differences can be avoided by using the default values for these parameters which hide the different types. It is also important to note that these classes are not available on other operating systems (but the rest of the library is).

3.3. Graphical applications

The easy integration with GUI frameworks such as V and MFC, in the application development process is also an interesting feature of the SndObj library. Along with the development of library, several test programs were implemented using different graphical frameworks. It was discovered that these provide an easier means of implementing SndObj-based GUI programs than the use of scripting languages such as Tcl/Tk. One of the reasons is the uniformity of programming style that is obtained by keeping the code in C++. Nevertheless, certain aspects of Tcl/Tk might prove to be useful for the implementation of some applications, such as a sound processing system based on this library.

An example of the integration with C++ application frameworks is found in the implementation of GUI-based *computer instruments* (see definition in [2] and [6]). One of these was created for the piece 'The Trane Thing', composed by this author, for saxophone and computer. This software was implemented on the Windows platform, with the help of a commercial C++ application framework, MFC. Designed as a single-window 'dialog' application, it uses elements and concepts from the MFC application models. Nevertheless, its development process was much more based on a perversion of than compliance with these models.

The instrument involves two main types of process: string resonators (based on the SndObj StringFlt class) and sampling of live material (based on SndLoop). Graphic controls are provided for switching the whole, or parts, of the instrument on/off, as well

as for adjusting parameters and triggering actions during performance. The program also implements the main processing as a separate thread, so that the graphic interface can be operated interactively. The thread control mechanism is provided by the MFC CwinThread class. A detailed discussion of the implementation of this computer instrument can be found on [4].

The dependence on the MFC library makes programs such as this one non-portable. For this reason, steps are being taken towards the development of application, process and thread control facilities within the SndObj library. The introduction of the SndThread class (and the development of derived classes from it) should provide the means for developing cross-platform graphic applications, possibly using portable GUI frameworks (such as V).

4. FUTURE PROSPECTS

Future prospects include the support for different types of gesture control and new processing classes, including spectral data manipulation. Also, more composition applications, such as the one mentioned above, are being developed. Finally, the library is being considered to provide the main engine for the implementation of a distributed audio processing system. This system will be based on the Beowulf platform [8], which consists of the Linux operating system and one of the available communications libraries/systems, such as MPI (Message Passing Interface) and PVM (Parallel Virtual Machine). It will be possibly portable to other computer platforms (including SMP machines). A scalable design will be employed, so that the system can make the most of the computing resources available on a particular platform. This project will involve two main elements: the addition of inter-process communication capabilities to the SndObj library; and the development of a multi-processor audio engine. The main aim of this research is to provide an affordable supercomputing facility for sound processing and synthesis.

5. REFERENCES

[1] Bradford, N, Buttlar, D and Farrell, JP, *Pthreads Programming: a POSIX standard for better programming*, Sebastopol, CA: O'Reilly Publ., 1996.

[2] Dodge, C and Jerse, T, *Computer Music: Synthesis, Composition and Performance*, New York: Schirmer Books, 1985.

[3] Lazzarini, V,"The Sound Object Library", *Organised Sound 5 (1)*, Cambridge: Cambridge Univ. Press., 2000, pp 35-49.

[4] Lazzarini, V, "Some Applications of the Sound Object Library". *Proceedings of the VII Brazilian Computer Music Symposium*. Curitiba: Editora da PUCPr, 2000, 152 and CD-ROM.

[5] Lazzarini, V and Accorsi, F, "Designing a Sound Object Library". *Proceedings of the V Brazilian Computer Music Symposium*. Belo Horizonte: Editora da UFMG, 1998, pp. 95-103.

[6] Moore, FR, *Elements of Computer Music*, Englewood Cliffs: Prentice-Hall, 1990.

[7] Pope, ST, "Machine Tongues XI: Object-Oriented software design". In: ST Pope(ed.), *The Well-Tempered Object*, Cambridge, Mass.: MIT Press, 1991.

Sterling, TL, Salmon, J, Becker, CJ and Savarese, DF, *How to Build a Beowulf: A guide to the Implementation and Application of PC Clusters*, Cambridge, MS.: MIT Press, 1999