The SndObj Sound Object Library

VICTOR E. P. LAZZARINI

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

The SndObj Sound Object Library is a C++ object-oriented audio processing framework and toolkit. This article initially examines some of the currently available sound processing packages, including sound compilers, programming libraries and toolkits. It reviews the processes involved in the use of these systems and their strengths and weaknesses. Some application examples are also provided. In this context, the SndObj library is presented and its components are discussed in detail. The article shows the library as composed of a set of classes that encapsulate all processes involved in synthesis, processing and IO operations. Programming examples are included to show some uses of the system. These, together with library binaries, source code and complete documentation, are included in the downloadable package, available on the Internet. Possible future developments and applications are considered. The library is demonstrated to provide a useful base for research and development of audio processing software.

1. BACKGROUND: SOUND COMPILERS, LIBRARIES AND TOOLKITS

Since the development, by Max Mathews, of the MUSIC-series of sound synthesis programs (Mathews 1960, 1961) in the early 1960s, several programs for synthesis and processing of audio in a general-purpose computer were developed. They were mostly based, directly or indirectly, on the MUSIC IV model. Programs such as these are sometimes referred to as *Computer Music Languages, Acoustic Compilers* or *Sound Compilers*. Some of them were developed for specific hardware and are now obsolete, whereas others, which enjoyed greater success, were developed under high-level languages, like MUSIC V (Mathews 1969), written in Fortran. Among these, *cmusic* (Moore 1990) and *csound* (Vercoe and Piche 1997), written in C, are two important examples of sound compilers. Csound is one of the most interesting sound processing systems available today, not only in terms of widespread use and portability, but also because of its continuous expansion and support by a large development community. Other MUSIC-derived sound compilers still in use include *clm* (Schottstaedt 1992), which runs in a Common Lisp environment, and *cmix* (Lansky 1990), which consists of a command parser and a library of C sound-manipulating functions. In fact, both clm and cmix can be considered hybrids of a sound compiler and a *programming library*, the former using Common Lisp, and the latter using C as the implementation language.

Programming libraries constitute a lower-level in computer music practice, when compared to sound compilers. The context where they appear is more general, and consequently their use is more complex. This can be easily demonstrated by the differences in coding of a similar instrument under csound and cmix (or clm), which will be shown later. It involves not only the knowledge of the syntax of a particular language, but also the grasp of concepts not necessarily needed when using a sound compiler. For instance, with cmix, the user has to understand variable declaration, memory allocation, header files, object code linking and other C-related concepts in order to build a sound processing or synthesis 'instrument'. The advantage is that, in this case, the programmer has more control over the processes involved in sound manipulation and can design applications that will suit better his/her needs. Apart from the mentioned use of libraries as part of packages such as clm and cmix, there are several other sound-manipulation programming libraries which were developed for various applications. As examples, two of these can be mentioned: the *SndLib*, developed at CCRMA for cross-platform sound input/output, and the *Sfsys*, developed by the CDP (Atkins *et al*. 1987), originally for their Atari-based sound filing system and then ported to other platforms (PC and UNIX).

Another related type of programming software library is the *toolkit*, normally associated with the object-oriented paradigm. Toolkit objects are normally used in programs by direct reference and, sometimes, by composition. In general, its use is more straightforward than the standard procedural- or modular-programming language libraries. Examples of toolkits are the NeXT-based Music and SoundKit (Jaffe and Boynton 1991), written in ObjectiveC, and the Synthesis Toolkit (Cook 1996), developed by Perry Cook in C++. The SndObj library (Lazzarini and Accorsi 1998), although not designed to be used solely as a toolkit, is also an example.

This section will examine in detail some aspects of sound compilers and libraries. First, csound is introduced as an example of a sound compiler, followed by a small programming example. It is then compared with two library-compiler hybrids, cmix and clm. Examples of similar code for these systems are also shown. This is followed by an overview of the object-oriented programming paradigm. Completing the section, two sound

Organised Sound **5**(1): 35–49 © 2000 Cambridge University Press. Printed in the United Kingdom.

manipulation toolkits are discussed, the Sound/MusicKit and the Synthesis Toolkit.

1.1. Csound, cmix and clm

Csound was originally developed by Barry Vercoe at the MIT, but it has been updated and expanded by several contributors, this author included. Cmix is the product of the work of Paul Lansky and others at Princeton University, and clm originated at Stanford, mainly as a result of Bill Schottstaedt's research. These packages are very different from each other, except for the fact that they have common predecessors, the MUSIC-family of programs, and similar applications. The main difference between these packages is the user-interface. Csound uses basically two text files of coded information as its input, one containing an 'orchestra' file, with signal-processing definitions organised in terms of 'instruments' and a 'score' file, with performance instructions for the 'instruments'. These two files are given as arguments to the csound command which compiles the audio. Cmix uses only one 'score' file, the 'instruments' are C-coded and pre-compiled as part of the library. The score file is passed to the cmix command (or directly to the instrument command if you are using only one instrument) which calls the C programs responsible for the sound processing. In fact, the score file is not strictly required because cmix works by interpreting commands that are passed to it. These commands are written in cmix's parsing language *MINC*, which is very similar in structure to C. Clm works in a Common Lisp environment, which is interpreter based. Calls to clm 'instruments' are interpreted similarly to any lisp function. Prior to its use, a clm instrument must be compiled and loaded. A 'score' file, based on lisp code, can be written to perform all the necessary calls to generate audio. This file can be loaded like any other file and the lisp interpreter will compile the sound.

Csound scores and orchestras are written using a very straightforward syntax. The orchestra file is normally divided into two types of statements: header and instrument block. The use of a header is not compulsory. In case of its absence, default parameters are used. The instrument block statements are preceded by the code **instr** *instrument__number*, and the instrument block is closed by **endin**. The general form of an instrument block statement is csound syntax can be defined as follows:

output__var **ugen** *input__args*

where *output__var* is any type of output variable (a, k, or i) and *input__args* is any number of input arguments to the *unit generator* **ugen**. A unit generator is a signal processing algorithm (such as an oscillator or envelope generator) which is used as a building block for an instrument. The output of a csound instrument is defined by the code:

out *asignal*

which can be thought of as a **ugen** without an output variable. The input argument is a variable of the type **a**, which is used to hold audio signals (actually a vector). A simple sine-wave csound instrument is shown below:

```
instr 1
asig oscil p4, p5, 1
out asig
endin
```

The arguments to **oscil** are amplitude, frequency in Hz and function table number (which stores a sine-wave shape). The variables of the type P, p4 and p5, and the function tables are defined in the score file. This will have a number of **f** statements which will define the function tables used in the synthesis process and a number of **i** statements. These define calls to instruments to generate/process audio. For example,

```
f1 0 1024 10 1
i1 0 2 16000 440
```

is a score that will create a sine-wave function table, defined as number 1, and call instrument 1 to generate a 440 Hz signal, with a peak amp of 16,000, from 0 to 2 seconds. The command

```
csound -odevaudio sine.orc sine.sco
```

will play that sound in real time (sine.orc and sine.sco being the orchestra and score files with the code shown above).

Cmix uses a different principle. Instruments are defined as commands called by the cmix environment, using minc syntax (Garton 1994). A C compiler is necessary, since instruments are coded as C functions and linked to the main cmix libraries and the Minc user-data parsing language. They make use of some cmix C functions, such as **setnote()**, **endnote()** and **ADDOUT()**, which provide basic soundfile and housekeeping functions. Unit generators are also provided, in the form of C library functions which can be employed in a user-defined instrument. The instrument designer has to provide a more detailed processing algorithm, including the necessary initialisation routines and a processing loop. The example below shows a simple cmix instrument which generates a simple audio signal:

```
double sine(float *p, int n__args) {
/* p[0]=start,p[1]=dur,p[2]=amp,p[3]=
freq,p[4]=function table */
int n, durs, length, outfile;
float fr, sr, amp, ndx, *wavetable,
output[1];

/* initialisation */
```


https://doi.org/10.1017/S1355771800001060

```
outfile = 1;
ndx = 0;
amp = p[2];
fr = p[3];
durs = setnote (p[0], p[1], outfile);
wavetable = floc((int)p[4]);
length = fsize((int)p[4]);

/* processing loop */
for (n = 0; n < durs; n++) {
output[0] = oscil (amp, fr* (length/SR),
length, wavetable, &ndx);
ADDOUT (output, outfile);
}
endnote(outfile);
return(1.);
}
```

The user-defined instrument is always a function with two arguments (an array of floats, which hold the instrument input parameters and an int number of parameters) returning a double. The **setnote()** function sets the initial position of the file read/write pointer and the duration of time for reading or writing to the file. It also returns the number of samples for the specified duration, according to the sampling rate defined in the soundfile header. One of the cmix particularities is that the soundfile header must exist on disk prior to synthesis. The **oscil()** function is the basic unit generator used by the instrument, an oscillator. Its arguments are amplitude, sampling increment, table length, pointer to table and an index to the current phase position. **ADDOUT()** reads an array, the length of which is determined by the number of output channels, and writes it to a file. In this case, because the output is mono, the output array is unity length. After compiling and linking this code, the following cmix command-line, using Minc commands, can be used to invoke it:

```
cmix output(''out__sfile'')
makegen(1,10,1024,1)
sine(1,1,16000,440,1)
```

These commands would open an output soundfile *out__sfile* and write 1 second of sine wave sound to it. They could also be saved as score file and the standard input redirected from that file. Minc also features C-like control-of-flow structures which can be used in a score file to control the processing done by the different instruments.

As mentioned before, clm works in a Common Lisp environment. It is in fact an extension to that programming language. Clm instruments are similar to lisp functions, although they are not defined by **defun**, but by **definstrument**. Nevertheless, the call syntax is the same. An instrument in clm can be created by using unit-generators (which are themselves proper lisp functions) and one of the output functions, which adds the instrument output into the current output soundfile. Similarly to cmix, the instrument designer has to provide all the initialisation and processing loop code. The following code shows a sine-wave instrument (generating similar output to the csound and cmix examples shown above):

```
(definstrument sine (start dur fr amp)
  (let*    ((beg    (floor    (*    start
sampling-rate)))
       (end   (+   beg   (floor   (*   dur
sampling-rate))))
       (sinusoidal (make-oscil :
frequency fr)))
     (run (loop for i from beg to end do
        (outa    i    (*    amp    (oscil
sinusoidal)))))))
```

This instrument, sine, uses an oscillator, created by **make-oscil**, called sinusoidal to generate (dur – start)* sampling-rate samples. These are added to the output file using **outa**. The **run** macro wraps the processing portion of the instrument, so that the code can be optimised. This generally involves the use of a C program to perform the processing, in which case a compiler is needed. It generally speeds up the process, since lisp code is too slow for signal processing. The C program is written, compiled and run by the system when the instrument is called. Before it is used the instrument must be either typed at the lisp listener or compiled and loaded from a file. The basic clm lisp macro used to open a soundfile for writing is **with-sound**, which receives the output of an instrument (or instruments) and writes it to a file, generally playing it back straight after the process. For example, typing

```
(with-sound () (sine 0 1 440 .1))
```

at the lisp listener will make the above defined instrument generate a 1-second A-440 sine-wave sound, which will be written to the default file, generally *test.snd* or *test.wav*. This file will be played back immediately by the system (if the playback is enabled). As the whole system is based on the Common Lisp language, any lisp-style control-of-flow is possible. 'Score' files can be created using lisp code which can be loaded to generate a soundfile.

As has been shown, the design of sound synthesis/processing instruments using csound is somewhat easier and faster than with the other two computer music languages. Its learning curve is also less steep. This simplicity is at the expense of flexibility and programming power. Clm and cmix have indeed a better integration of implementation, orchestra and score languages, which is favoured by some authors (Pope 1993). Also, because they are language/library hybrids, they are more easily extendable. Their use as development tools for DSP applications benefits very much from this fact. On the other hand, both systems need compilers/interpreters for the development of instruments, whereas csound is a complete sound compiler driver.

Another issue concerning the use of sound compilers and libraries is their portability. As mentioned before, the most successful systems are the ones based on portable code. Csound seems to have been ported to most of the popular platforms: UNIX, MacOS and MS-Windows. Cmix and clm, although now somewhat out of the NeXT/Macintosh niche, are not as portable as csound. The clm source at CCRMA is supposed to be portable to Windows, under Clisp. This author's experience is that a lot of editing of the lisp source code is necessary for a successful build under Windows (using freeware gcc compiler and cygwin32). Cmix is not at all portable to MS-Windows. A truly portable sound processing library would be greatly welcome. One of the goals of the SndObj project is the development of a portable core of sound processing objects, thus answering the need for such tools.

1.2. Object-oriented design

A different solution for the design of sound processing systems and libraries is found under the object-oriented programming paradigm. The definition of object-oriented programming is not very clear cut (Pope 1991), although the concepts it involves are well defined. The first one is that of data abstraction, whereby new data types, called *abstract data* or *user-defined types*, or classes, can be designed to behave similarly to built-in ones (Stroustrup 1995). Moreover, they can also include a full set of operations (methods) that can be used to manipulate that particular data type. An abstract data type can be considered as a kind of a black box which can model a real-life object. Some elements that compose the characteristics of an object can be hidden, in what is called encapsulation, allowing easier and safer handling of information and processes. Also crucial to the object-oriented paradigm is another mechanism, that of inheritance, which allows abstract data types to pass on their characteristics (and operations) to other types which extend/refine the definition of that type. The inheritance concept allows for trees of classes, which can be very functional and useful for applications, such as sound processing systems.

The design of a system using the object-oriented paradigm is very much based on decomposition and composition processes, recognition of likeness and behaviour of objects. Another idea involved in this process is that of the level of reuse or sharing and maintenance. Software based on object-oriented concepts is very much directed towards the reuse of algorithms and data structures. Four basic techniques can be defined for object-oriented design (Pope 1991): compositions, refinement, factorisation and abstraction. Composition is basically the reuse of existing classes as instance variables of new classes, composing a new object out of other objects it should contain. Refinement is the use of inheritance to create a new class which has specialised characteristics but is basically similar to its parent class. Factorisation is the description of a class in terms of a hierarchy of aspects, using the inheritance mechanism. Instead of defining a class complete with all its concrete attributes, a hierarchy of class can be defined, which will factorise the characteristics and behaviour of the original class. Finally, abstraction uses the idea of finding commonalities between concrete objects and defining an abstract class which embodies these common characteristics. This class will then serve as the base from which the others are created.

Object-oriented programming packages can be grouped into two main categories, according to their use: *toolkits* and *frameworks*. The former is based on reuse by direct reference and composition. Users will basically employ the existing code by declaring instances of classes and using them to perform the wanted actions. Two examples of this type of package are discussed below. Frameworks are used, in general, by refinement and composition of existing classes. They are based on class hierarchies created by factorisation and abstraction. The most common uses of frameworks are in the development of graphical user interfaces: V (Wampler 1998) is a typical example of a GUI framework. Kyma (Scalletti 1991) is an example of a sound processing system which incorporates a framework architecture, in its sound model classes. The SndObj library was designed with both framework and toolkit uses in mind.

1.3. Sound processing toolkits

The NeXT **Music** and **SoundKit**s, and the **Synthesis Toolkit** are two examples of sound synthesis/processing toolkits. The first two were implemented in ObjectiveC for the NeXT computer using a Motorola DSP 56001 chip. They were designed to provide the software library support for music applications development. It includes classes for, amongst other things, 56001-based synthesis/processing, audio recording, playback and editing. The Synthesis Toolkit is a sound synthesis library written in C++, having ports across UNIX and MS Windows platforms. The case for a more portable system is evident here. The MusicKit (and SoundKit) had to endure the fate of being associated with a particular hardware which was not very popular outside some computer music centres (esp. in the USA). On the other hand, the Synthesis Toolkit seems to have a more promising future.

The SoundKit, developed by Lee Boynton, was designed for recording, playback, editing and graphic display of digital audio. It is based on the **Sound** class, an object which is wrapped around the sound data structure and provides the methods for performing the above-mentioned actions. The sound data structure can contain sampled audio or DSP code images and data streams, depending whether it has been used for manipulating audio or controlling DSP-based synthesis. In addition to

https://doi.org/10.1017/S1355771800001060

the Sound class, a **SoundView** class was designed to provide a mechanism for sound graphic display. The MusicKit, developed by David Jaffe, provides the other tools for music application development, including performance, music representation and synthesis. Relevant to this discussion is the synthesis part of the MusicKit. Its model is based on three main classes: **SynthElement, SynthPatch** and **SynthInstrument**. SynthElement is an abstract class which has two subclasses: **UnitGenerator** and **SynthData**. The former is the base class for all signal processing functions and the latter is used to provide patchcords between UnitGenerator objects. These objects include oscillators, envelopes and other standard signal processing algorithms. The SynthPatch-derived classes are collections of SynthElement objects which define a certain processing configuration. SynthInstrument classes manage the use of SynthPatch classes. The Music and SoundKits model seems very sound, its major weakness being the dependence on a specific hardware/OS.

The Synthesis Toolkit, v.1.0, is a port of algorithms and instrument models developed by its author, Perry Cook, on many different platforms and languages, including the mentioned NeXT MusicKit. The motivations for creating this toolkit, according to the author, are based on a desire for portability and extensibility, taking into account the evolution in efficiency and power of modern CPUs. The entire Synthesis Toolkit is derived from a base class called **Object**. This class controls the basic behaviour of the system, including the type of IO used (file format, realtime, etc.). Floating-point numbers, either double- or float-precision, are used to represent audio samples. The processing is implemented, on the audio sample based unit generator classes, by a fundamental **tick()** method. This function causes the unit generator to compute one audio sample, returning it as its output. The sound-generating objects, such as oscillators and envelope generator, which act only as sources of audio, implement a tick() that does not take any arguments. Other objects, such as sound output classes, only receive samples, returning void. Other processing objects take audio samples as arguments to tick() and return their output. A **LastOut()** method, implemented by objects that are sources of audio, can be used to feed multiple sample consuming objects. An example of a simple sound processing algorithm, given by the author in Cook (1996), is transcribed below:

```
ExampleClass() {
envelope = new Envelope;
waveIn = new RawWvIn ("infile.raw");
filter = new OnePole;
output = new RawWvOut ("outfile.snd");
}

MY_FLOAT ExampleClass::tick(void) {
Output->tick( envelope->tick() *
filter->tick(waveIn->tick()));
}
```

This algorithm reads an input file, applies a filter and an envelope to it and writes it out to another file. Only the class constructor and the tick() function are shown. This toolkit hosts a great number of synthesis and processing objects, roughly 60 C++ classes, which makes it a very impressive library. The portability and extensibility are also valuable aspects of this system. These features were also included in the design of the SndObj library, along with some other desirable characteristics. The next sections of this paper will introduce and discuss the library in closer detail.

2. THE SOUND OBJECT LIBRARY VERSION 1.0

The initial motivation for the development of this library evolved during work on the software *Audio Workshop* (Lazzarini 1998). It was observed that the creation of a set of objects for audio signal processing could be very useful in the design of new applications. It could help provide higher-level tools for audio programming and software with a more intuitive user interface. This set of objects would be designed as a toolkit, to be employed directly to build a DSP program (or in a visual patching application, to create an instrument patch), and as a framework, to which developers could add their own specialised code. Programs to carry out specific tasks could be easily developed by connecting the available objects and providing standard control-of-flow. Using derivation and inheritance, new objects could be created from the existing ones, thus leaving the development possibilities open. The framework would thus be a useful tool for research in signal processing, acoustics and psychoacoustics.

The project was based on three basic principle, as explained in the preliminary account of this research work (Lazzarini and Accorsi 1998): (i) encapsulation, (ii) universal patch-ability, and (iii) portability. All the processes involved with production, manipulation and storage of audio data should be encapsulated by the classes. Patching of objects should be unrestricted, as if they were modules in an analog synthesizer, or unit generators in systems such as the sound compilers examined earlier on in this article. As a final premise, the main processing and input/output code should be portable. This would also allow for machine-dependent specialisation when necessary. The project was developed under C++, which seemed a very good choice, mainly for its C-compatibility and good support for object-oriented programming.

The SndObj library was initially developed by this author, assisted by Fernando Accorsi, at the Núcleo de Música Contemporânea and the Department of Computer Science, Universidade Estadual de Londrina, in Brazil. The first versions of the library were built under AIX onan IBM Risc2000 machine (using the g++ compiler), as well as on a Pentium PC under

https://doi.org/10.1017/S1355771800001060

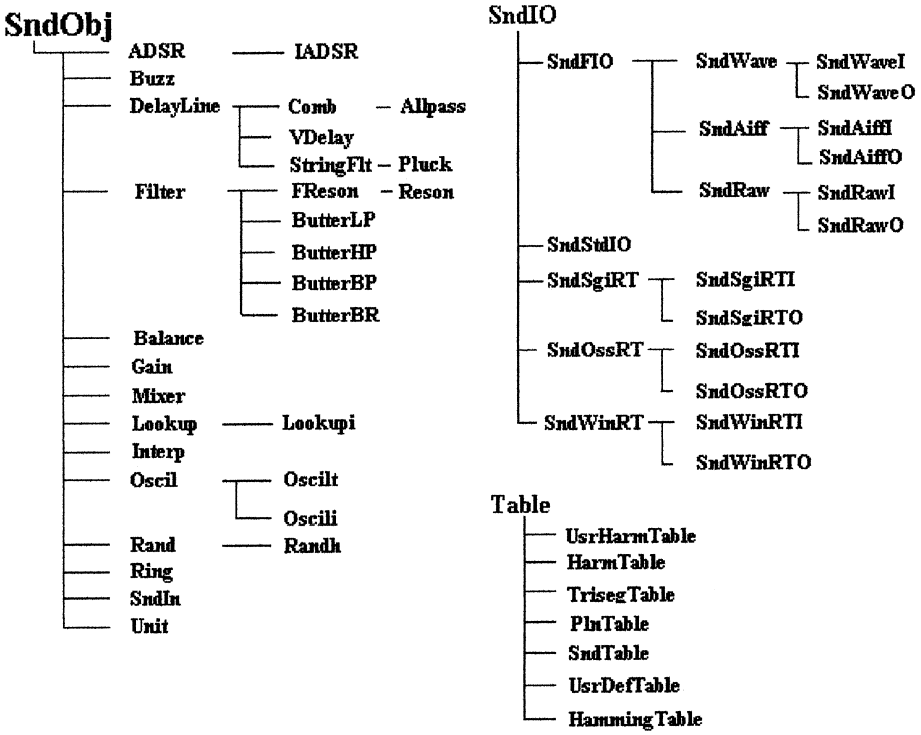

Figure 1. The SndObj library class trees.

Windows95 (using both MS Visual C++ and Cygwin g++). The development work continued at the National University of Ireland, Maynooth. Versions for Solaris (Sparc) and IRIX (MIPSPro) were developed, as well as updated MS Windows versions and, more recently, a Linux one. SndObj library version 1.0 binaries are available for these three platforms and it is expected that the source code can be built on any other UNIX platform.

2.1. The class 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, mathematical function-table and sound input/output objects. The SndObj library version 1.0 comprises more than fifty classes (cf. Appendix A) organised in three main class trees. A diagram showing the inheritance relationships between the classes in the library is shown in figure 1. The classes derived from SndObj are involved in the production and manipulation of sound samples. Some of them make use of the Table classes, when some sort of function table is needed. The SndIO tree is dedicated to all the actions involving sound input or output: disk, ADC/DAC, standard IO, etc. These classes use a SndObj as their input. A SndObj-derived **SndIn** class was designed to receive an input from a SndIO-derived object. This would enable audio from input sources to be inserted in the processing chain.

2.2. The SndObj-derived classes

A SndObj object, as modelled for this library, has one output, a sampling rate, an on/off switch and an error code, as shown in figure 2. The output is defined as a pointer to a **float** location, ***m__output**. SndObj-derived classes also have a **DoProcess()** method that carries out all the processing duties and other methods to access its member variables. The derived classes have an undefined number of inputs (objects and/or offset values). As mentioned before, some of them also rely on maths function-table objects to do their processing. Sound processing objects are designed to be easily interconnected. This is done by passing the address of a SndObj-derived class instance to the object receiving its output, for example:

```
object2.SetInput(&object1);
```

Most of the objects can receive one or more SndObj inputs. As an example, 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. This ability to easily make patches of processing boxes gives the flexibility necessary for users to create a great number of applications.

SndObj-derived classes in general have two constructors, a default constructor which builds a bare object and another one that initialises the object parameters to its input arguments. They also allocate a

https://doi.org/10.1017/S1355771800001060

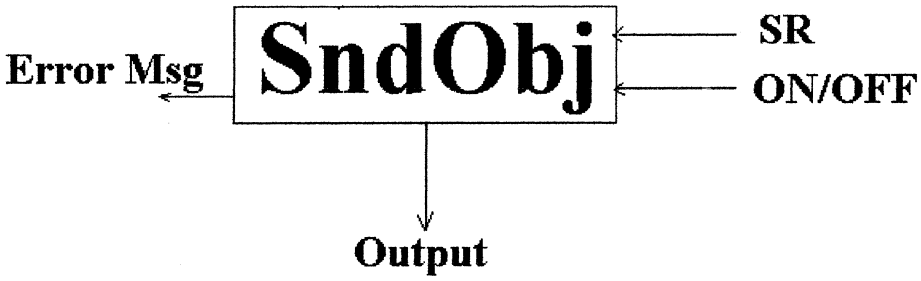

Figure 2. The SndObj model.

memory space for the output sample and switch the object on by calling the **Enable()** method. Methods for setting or updating class members are also available for each parameter (as in the above example for setting the input of an object). The DoProcess() method is a virtual function declared in SndObj and implemented differently on each of its derived classes. This function would be equivalent to the tick() method of the Synthesis Toolkit, discussed earlier in this article. It calculates one sample every time it is called and places this sample at the output of the object. If another object is patched to it, its DoProcess() method will look at that output and use it to compute its own output sample. The way patching works in the SndObj library is completely different from the quoted example of the Synthesis Toolkit. The DoProcess() method never accepts any arguments (its input sample, or samples, is/are taken directly from another object). It returns unity, when successful, and null, if some problem has occurred. As the DoProcess() calculates one sample every time it is called, it should be placed in a processing loop, after calls to other DoProcess() belonging to input objects, as shown below:

```
for(int n = 0; n < end*sampling__rate;
n++)
{ // processing loop
(. . .)
object1.DoProcess();  //  object1  is
patched to
object2.DoProcess(); // the input of
object2
(. . .)
}
```

Other virtual classes declared in SndObj are **ErrorMessage()**, which returns an error string according to an internal error code, **SetSr()**, which sets or updates the sampling rate, and a virtual destructor. This enables the destruction mechanism of derived classes to work properly. The SetSr() method is declared virtual because in some cases there is the need to perform object-specific operations after an alteration to the sampling rate.

2.3. Tables

The table classes were developed to supply certain SndObj-derived objects with tabulated mathematical functions. Tables are modelled as wrappers around a floating-point vector. Their basic attribute is a table length, which determines the size of the array. A basic **MakeTable()** method is implemented in the Table-derived classes. This method is called by their constructors. A **GetTable()** method provides the basic access to the table itself, returning a pointer to its first location. **GetLen()** returns the length of the table. A common use for tables 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 object by passing its location to the constructor or to a SetTable() method:

```
HarmTable sawtable (1024, 25, SAW); //
saw wave with 25 harmonics
oscillator.SetTable(&sawtable);     //
oscillator is an Oscilt or Oscili
                                    //
object
```

Similar to this is the UsrHarmTable, which allows the user to define the relative amplitude of the individual harmonics. SndTable stores sampled sound, input from a SndIO-derived object. 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 generalised Hamming window function table is supplied, as the HammingTable object, as well as a polynomial drawing function, PlnTable. Table objects are very useful and can be employed in a variety of ways. New table-derived objects are constantly added to the library implementation.

2.4. Input and Output

The SndIO classes are designed to deal with all the input and output services needed by the sound objects. Central to their operation are the **Read()** and **Write()** methods. They are used to perform the IO functions, regardless of whether the target is a soundfile, ADC/DAC, computer screen or some other device. The derived classes can fit

https://doi.org/10.1017/S1355771800001060

into any main category of input/output: soundfile (which has derived classes for different formats), DAC/ADC, screen output, MIDI. Some SndIO-derived objects are expected to be platform dependent – the ones that rely on platform-specific features such as realtime IO and graphics.

Output SndIO-derived classes can receive one SndObj input per channel. This patching can be done by the class constructor, or by a **SetOutput()** method. It is done in a similar way to any other patching operation throughout the library, by passing the address of an object:

```
output.SetOutput(&sound); // sound is a
SndObj object
                          // and output
a SndIO object
```

Conversely, a SndIn object can receive a SndIO-derived input:

```
SndIn sound(&input, 1); // input is a
SndIO object
                        // from which
sound is reading channel 1
```

The Read() and Write() methods are built in such way to work transparently in a processing loop with the SndObj DoProcess() methods. Although they do not always read and write one sample at a time, they are designed to behave as if they did. The SndIO buffer can be set, in the class constructor, to any size desired. Depending on certain hardware conditions, they can alter the performance of the read/write operation. A processing loop reading from one input and writing to another is shown below:

```
for(int n = 0; n < end*sampling_rate;
n++)
{
input.Read();
sound.DoProcess(); // SndIn object
(. . .)
                   // any processing
output.Write();
}
```

At the present version the SndIO hierarchy has four main subclasses: **SndFIO**, soundfile IO; **SndStdIO**, which sends and receives samples from the standard IO; **SndSgiRT**, to perform realtime audio IO on Silicon Graphics; **SndWinRT**, realtime audio IO on Windows; and **SndOssRT**, Open Sound System realtime audio IO.

3. PROGRAMMING EXAMPLES

A number of sample applications, in the form of console programs, were developed to demonstrate some uses of the library. These programs can also be used as tutorials and therefore are included in the documentation (Lazzarini 1999). This article will first examine three programming examples which employ SndObj library classes. They should demonstrate the encapsulation, relative user-friendliness and modular aspects of the system. As a final example, this article will explore the steps involved in the development of new SndObj-derived classes.

3.1. A simple program

The first example shows a very simple sine-wave synthesis routine using the SndObj-derived class **Oscilt**. The Table-derived **HarmTable** and the SndIO-derived **SndWaveO** classes are also employed, providing a tabulated sine function and file output services, respectively. The comments (starting with a '//') explain the constructs used in the program.

```
#include <stdlib.h>
#include "AudioDefs.h" // SndObj
headers and definitions

int main (int argc, char* argv[]) {

float dur=(float)atof(argv[2]); // dur:
second command-line argument
float amp=(float)atof(argv[3]); // amp:
third command-line arg
float fr=(float)atof(argv[4]); // fr:
fourth command-line arg
HarmTable table(1024, 1, SINE); // sine-
wave table
Oscilt oscil(&table, fr, amp); //
oscillator
SndWaveO output(argv[1], 1, 16); //
1-channel 16-bit precision
                                // RIFF-
Wave file output
                                //
filename: first command-line arg
output.SetOutput(1, &oscil); // assign
the oscillator to the file
                                // output
channel 1

for(int n = 0; n < dur*oscil.GetSr();
n++) // processing loop
{
oscil.DoProcess();
output.Write();
}
return 1;
}
```

This code can be compiled and linked, generating a program called **sine**. The command-line

```
sine test.wav 1 16000 440
```

would generate a RIFF-Wave soundfile named 'test.wav' containing 1 second of 440 Hz sinusoidal sound.

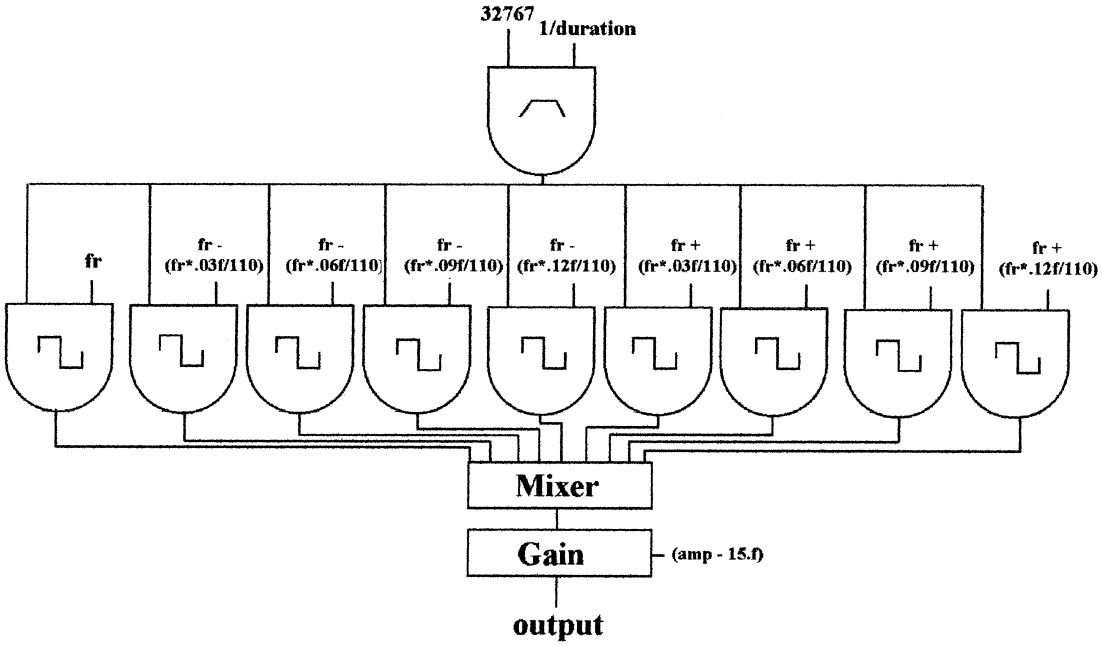

Figure 3. Risset signal flowchart.

3.2. Risset

The next example, a program called **risset**, is a slightly more complex synthesis application, based on the ingenious Risset design (Lorrain 1980), where nine oscillators are combined to generate a cascading harmonics drone. This is done by the minute differences in frequency of the oscillators, which creates a sequence of phase cancellations/sums. This example uses a square wave as the basic oscillator shape. The signal flowchart of this program is shown in figure 3. The program code is shown below (excluding the usage() function):

```
#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
Mixer mix;
mix.AddObj(&oscil1);
```


https://doi.org/10.1017/S1355771800001060

```
mix.AddObj(&oscil2);
mix.AddObj(&oscil3);
mix.AddObj(&oscil4);
mix.AddObj(&oscil5);
mix.AddObj(&oscil6);
mix.AddObj(&oscil7);
mix.AddObj(&oscil8);
mix.AddObj(&oscil9);

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

// output to an AIFF-format soundfile
SndAiffO output(argv[1], 1, 16);
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;
}
```

The program accepts a command-line of the following form:

```
risset filename.aiff dur(secs) amp(dB)
freq(Hz) no__of__harmonics
```

It uses a truncating oscillator as an envelope and interpolating oscillators as audio generators. A **Mixer** object sums the outputs of the oscillators and a **Gain** object attenuates and controls the overall amplitude of the output. An instance of the **SndAiffO** class writes to an AIFF-format soundfile.

3.3. A string resonator box

The last example is based on the **StringFlt** class. This implements the model of a sympathetically vibrating string. This program uses a number of these objects to simulate a box containing a number of strings tuned to different fundamental frequencies. It leaves the number of strings and their tuning for the user to decide. It can be used interactively or its parameters can be defined in a text datafile supplied as an argument to the program. The version shown below demonstrates the realtime input and output, as implemented on Silicon Graphics machines, under IRIX 6.5. A signal flowchart for this application is also shown in figure 4.

```
#include <iostream.h>
#include <fstream.h>
#include <stdlib.h>
#include <SndObj/AudioDefs.h>
#include <SndObj/SndSgiRTO.h>
#include <SndObj/SndSgiRTI.h>
void usage ();

int main (int argc, char *argv[]){

int nstrs;
float fdbgain;
float gain;
float dur;
float sr;
float* fr;
SndSgiRTI *input;
SndSgiRTO *output;
if(argc = 1) {    // prompt for
parameters
cout << "Enter duration: ";
cin >> dur;
cout >> "Enter sampling rate: ";
cin >> sr;
cout << "Enter number of strings: ";
cin >> nstrs;
cout << "Enter feedback gain (0 < fbd <
1): ";
cin >> fdbgain;
cout << "Enter gain attenuation(dB): ";
cin >> gain;
fr = new float[nstrs];
for(int i=0; i<nstrs; i++){
  cout << "String" << (i+1) <<
"frequency: ";
  cin >> fr[i];
  }
 }

else if(argc = 3) {    // get the
parameters from file
  ifstream datafile(argv[3]);
  datafile >> dur;
  datafile >> sr;
```


https://doi.org/10.1017/S1355771800001060

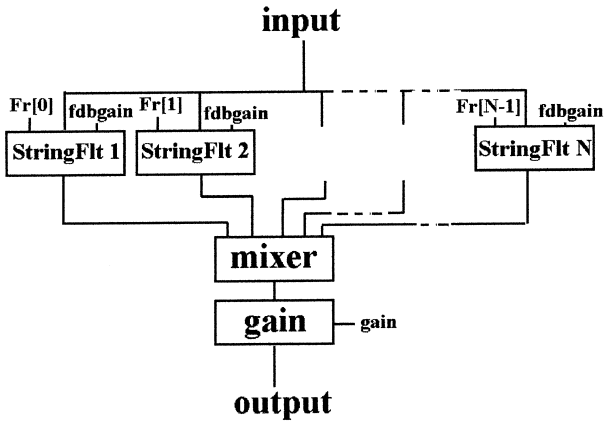

Figure 4. Streson signal flowchart.

```
  datafile >> nstrs;
  datafile >> fdbgain;
  datafile >> gain;
  fr = new float[nstrs];
  for(int i=0;i<nstrs; i++)
    datafile >> fr[i];
 }

  else {    // display usage message
  usage ();
  return 0;
 }

// realtime IO objects
SndSgiRTI input(1, 512, 1024, SHORTSAM,
sr);
SndSgiRTO    output(1,    512,    1024,
SHORTSAM);

// Sound input, attenuation, string
filters
SndIn sound(input, 1);
StringFlt*      strings      =      new
StringFlt[nstrs];
Mixer mix;
// String Filters parameters set-up
for(int i=0;i<nstrs; i++) {
  strings[i].SetFreq(fr[i]);
  strings[i].SetFdbgain(fdbgain);
  strings[i].SetInput(&sound);
  mix.AddObj(&strings[i]);
     }

// Gain attentuation
Gain atten(gain, &mix);
Output->SetOutput(1, &atten);

// processing loop
unsigned   long   n,   end   =   (unsigned
long)dur*sr;
for(n=0; n < end; n++) {
  input->Read();     // input from ADC
  sound.DoProcess(); // sound input
  for(int   i=0;   i   <
nstrs; i++)

strings[i].DoProcess();// string filters
  mix.DoProcess();   // mixer
  atten.DoProcess(); // attenuation
  output->Write();   // output to DAC
        }

delete input; // delete IO objects to
delete output; // clean-up the hardware
buffers

return 1;
}
```

The command line for this program has the form:

streson [datafile]

The program can be activated just by typing its name at the UNIX shell or by supplying a datafile as argument. This is a text (ASCII) file containing the following parameters, each on a new line:

```
duration
sampling rate
number of strings
feedback gain
attenuation(dB)
string 1 frequency
string 2 frequency
(. . .)
string N frequency
```

The output of the program is the input passed through a resonating box which simulates the vibration of a number of strings.

3.4. Deriving a class from SndObj

In order to develop other classes and extend the library, there are a few simple steps that the user should follow:

(1) Define your SndObj-derived class in a header file ('MyNewClass.h'). Include in the class declaration

https://doi.org/10.1017/S1355771800001060

one **DoProcess()** and one **ErrorMessage()** method:

```
#ifndef MYNEWCLASS__H
#define MYNEWCLASS__H
#include ''SndObj.h''

Class MyNewClass : public SndObj{

protected:

// . . .declare here your member variables
// and protected methods . . .if you are
// creating a processing class, define the
// input as SndObj* input;

public:

// . . .declare at least two constructors:
// the default one and another which initialises
// all parameters to the supplied arguments.
// declare Set. . .() methods for every parameter.

virtual ~MyNewClass();
virtual short DoProcess();
virtual char* ErrorMessage();

}
```

(2) Provide the implementation code for all constructors, destructor and methods you defined. Put them in another file ('MyNewClass.cpp'). The constructors should also initialise member variables inherited from SndObj. The sampling rate, **m__sr** is normally taken from the input object (using **GetSr()**). Memory space for the output location ***m__output** should be properly allocated.

(3) The **DoProcess()** and **ErrorMessage()** methods should be implemented as follows:

```
short MyNewClass :: DoProcess() {
if(m__enable) { // the object is switched on

// . . .define all your processing code to work on a single-sample
// basis. In order to obtain the input signal use
// input->Output(). This returns the input object's output sample
// as a float. Assign the result of your processing to *m__output,
// the location of this object's output sample.

}
else *m__output 0.f; // the object is switched off
return 1;
}

char* MyNewClass :: ErrorMessage(){

  char* message;
  switch(m__error) {

  case 0:
  message = "No error.";
  break;

  // . . . define here the error messages for the error
  // codes you defined in your methods

  }
return message;
}
```

(4) Add the following line to the end of the AudioDefs.h file:

```
#include MyNewClass.h
```

(5) Compile your class and append it to the library binary. You can do that by adding your class to the supplied Makefile and calling `make`. First add '**MyNewClass.o**' to the end of the **EXOBJS** list. Then add the following line at the end of the file:

```
$(oDir)/MyNewClass.o:
MyNewClass.cpp MyNewClass.h SndObj.h
$(CC) -c $(CFLAGS) -o $@
MyNewClass.cpp
```

Now the user can employ a newly created MyNewClass as a processing object together with the other library classes.

4. MUSIC AND RESEARCH APPLICATIONS

The Sound Object Library is being developed with three possible main applications in mind: DSP/audio synthesis research, composition and general music use. The library itself is directly useful for the first two. The software applications created with the library can be employed by anyone with some knowledge of computers in music. For the researcher, the library offers many advantages: easy access to sound IO, a set of pre-built signal generators and modifiers and a stable framework. Composers can use the library as they would use a sound compiler, with the added versatility of an object-oriented design. Finally, since the development curve is very short, applications can be very easily created for general use. Simple examples of these are the previously discussed sample applications and some other GUI-based sound processing programs developed using the library.

The library is designed very much as a workbench for

https://doi.org/10.1017/S1355771800001060

research and composition. While it clearly does not have the variety of generators/modifiers as some sound compilers have, it has a great potential for evolution. By virtue of its object-oriented design, many aspects of a development cycle can be rationalised and improved. For that reason, this library can be used as a useful tool by computer music research groups. In comparison to certain hardware-dependent systems, the library has the advantage of being fully portable across many platforms (and its core elements are portable to any platform with a C++ compiler). This is also very useful since the use of mixed MS-Windows and UNIX environments is widespread in the research community.

The integration of composition and development facilities was the main motivation behind the library project. As mentioned before, the need for this kind of tool came originally from the software development process, but its final application is not limited to it. The use of the library as an aid to electroacoustic music composition has been explored, mainly by this author, in preparing elements for several pieces. Also, the implementation of realtime capabilities makes it possible to use the computer in interactive situations. This can lead to new ways of conceiving the computer as a possible option for electroacoustic instrumentation. For composers with some computer programming skills, the possibility of designing and fine-tuning the sound synthesis/processing routines in relation to compositional needs presents interesting opportunities.

Work on a new piece for live instrument (saxophone) and computer, which illustrates some applications of the library, is under way. For this piece, a special computer *instrument* is being created. This is a GUI program, which performs realtime processing and synthesis of sound, designed to interact with the live instrument. The library is being used (in conjunction with a commercial GUI framework) to generate this application. When finished, this piece will only depend an audio-equipped PC-compatible computer and the live instrument. All the sound processing will be controlled by the SndObj-built program. Also, a cross-platform version of the controlling program can be easily built (using a portable GUI framework). Considering the present generation of microcomputers, such a piece should not represent great practical difficulties for performance.

5. FUTURE PROSPECTS

The SndObj library is available for download at the Maynooth Music Technology Laboratory Web site at http://www.may.ie/academic/music/musictec. A number of SndObj-based GUI applications are also available at the same location. The library is currently being used as a research and composition tool in the Department of Music at NUI, Maynooth. This research includes the development of new classes and applications for the library. A complete HTML reference manual was created by this author to help users creating their own sound processing applications. This documentation includes the code listings of 14 sample programs designed for synthesis, processing and utility applications (cf. Appendix B). The realtime capabilities of the Silicon Graphics and Windows versions are going to be used as a complement to outboard processing and synthesis equipment in our electronic and recording studio at Maynooth. Windows MME and Open Sound System realtime IO classes were recently added to the library. This extends the library realtime capabilities to Windows, Linux and UNIX platforms with OSS-compatible audio hardware. Solaris realtime classes are not being considered because of the poor audio support found on Sun hardware.

Several options were considered to give the library a graphical user interface. Preliminary studies were carried out using the V framework (Wampler 1998). This is a cross-platform framework which provides a good set of tools for building user interfaces. The fact that it is implemented in C++ helps its integration with the library. This is a clear advantage over other possibilities using another implementation language, such as Tcl/Tk or Java. A simple sample application, developed using V, is available for developers interested in examining the use of that framework in conjunction with the SndObj library. An interesting possibility which was considered is the design of a graphical patching application which would use the library as its processing engine. This is another exciting prospect for development opened by the research work which generated the SndObj library.

6. CONCLUSION

As introduction to the subject, this article initially presented some of the currently available systems for computer-generated music. Sound compilers were presented as one option for the development of signal processing applications, together with libraries and toolkits. The SndObj library, version 1.0, was presented as an object-oriented framework and toolkit for audio processing. Its components were discussed in detail and its class hierarchy was explained. It was shown that the library is composed of a set of classes that encapsulate all processes involved in synthesis, processing and IO operations. The library was demonstrated to be a modular and user-friendly object-oriented system. Application examples were provided as an introduction to the design of programs that can make use of the library facilities. A complete reference documentation is available, in HTML format, and the library can be downloaded from the Internet. There are good prospects for the continuous

https://doi.org/10.1017/S1355771800001060

development and expansion of its possibilities and applications.

APPENDIX A

List and description of SndObj Library version 1.0 classes:

SndObj	Abstract base class for all audio processing classes
ADSR	Attack–Decay–Sustain–Release envelope generator/processor
IADSR	Similar to ADSR but including an initial and a final state
Balance	Balances the rms output of one input according to another
Buzz	Discrete summation formula pulse wave generator
DelayLine	Simple delay line
Comb	Comb filter
Allpass	Allpass filter
StringFlt	String resonator combining a feedback delay line with allpass and lowpass filters
Pluck	Karplus–Strong plucked-string sound generator
Vdelay	Variable delay with feedback, feedforward and direct gain controls
Filter	Abstract base class for filter classes
ButtBP	Butterworth band-pass filter
ButtBR	Butterworth band-reject filter
ButtHP	Butterworth high-pass filter
ButtLP	Butterworth low-pass filter
Freson	Fixed 2nd-order resonator
Reson	Variable resonator
Gain	Gain attenuator/booster
Interp	Interpolation between two points
Lookup	Table lookup on behalf of an input index
Lookupi	Interpolating table lookup
Mixer	SndObj mixer class
Oscil	Abstract base class for all oscillator classes
Oscilt	Truncating oscillator
Oscili	Interpolating oscillator
Rand	Random-signal generator
Randh	Sample-and-hold random signal generator
Ring	General purpose multiplier and ring modulator
SndIn	SndIO-based signal input
Unit	Test signal generator
SndIO	Abstract base class for all input/output classes
SndFIO	Abstract base class for all file input/output classes
SndAiff	Abstract base class for AIFF-format classes
SndAiffI	AIFF file input
SndAiffO	AIFF file output
SndRaw	Abstract base class for Raw file classes
SndRawI	Raw file input
SndRawO	Raw file output
SndWave	Abstract base class for RIFF-Wave classes
SndWaveI	RIFF-Wave file input
SndWaveO	RIFF-Wave file output
SndStdIO	Standard input/output class
SndOssRT	Abstract base class for Open Sound System realtime IO
SndOssRTI	Open Sound System realtime input
SndOssRTO	Open Sound System realtime output
SndSgiRT	Abstract base class for Silicon Graphics realtime IO
SndSgiRTI	Silicon Graphics realtime input
SndSgiRTO	Silicon Graphics realtime output
SndWinRT	Abstract base class for Windows MME realtime IO
SndWinRTI	Windows MME realtime input
SndWinRTO	Windows MME realtime output
Table	Abstract base class for all maths function table classes
HarmTable	Harmonic function table
HammingTable	Generalised Hamming window
PinTable	Polynomial table
SndTable	SndIO-audio input table
TrisegTable	Three-segment table
UserHarmTable	User-defined harmonic function table
UserTable	User-defined table

APPENDIX B

List and description of basic SndObj-based sample application programs:

beau	Beauchamp cornet emulation using waveshaping and high-pass filter
cutsf	Soundfile segment cutting
cvoc	Channel vocoder using Butterworth filters
cvoc2	Channel vocoder using fixed resonators
flanger	Flanging and phasing effects processor
karplus	Plucked-string emulation
mixsf	Soundfile mixer
risset	Cascading harmonics drone generator
raw2wave	Raw to RIFF-Wave format conversion
schroeder	Reverberator using Schroeder configuration
slicesf	Soundfile splicing
streson	String resonator
wavegen	Simple wave generator
windharp	5-string windharp using string resonators

https://doi.org/10.1017/S1355771800001060

REFERENCES

Atkins, M., *et al*. 1987. The Composer's Desktop Project. *Proc. of the 1987 Int. Computer Music Conf*., pp. 146–50. San Francisco: International Computer Music Association.

Cook, P. 1996. *Synthesis Toolkit in C++ Version 1.0*. Postscript paper available at http://www.cs.princeton.edu/~prc/STKPaper.ps

Garton, B. 1994. *What is Cmix?* HTML document available at http://silvertone.princeton.edu/winham/Garton.html

Jaffe, D., and Boynton, L. 1991. An overview of the Sound and Music Kits for the NeXT computer. In S. Pope (ed.) *The Well-Tempered Object*, pp. 107–18. Cambridge, MA: MIT Press.

Lansky, P. 1990. *Cmix Release Notes and Manuals*. Princeton: Department of Music, Princeton University.

Lazzarini, V. 1998. A proposed design for an audio processing system. *Organised Sound* **3**(1): 77–84.

Lazzarini, V. 1999. *The SndObj Reference Manual*. HTML reference available online at http://www.may.ie/academic/music/music/man/SndObj/index.html

Lazzarini, V., and Accorsi, F. 1998. Designing a sound object library. In M. Loureiro (ed.) *Proc. of the V Brazilian Computer Music Symp*., pp. 95–104. Belo Horizonte: Editora da UFMG.

Lorrain, D. 1980. Inharmonique, Analyse de la Bande de L'Oeuvre de Jean-Claude Risset. *Rapports Ircam* **26**.

Mathews, M. 1960. Computer program to generate acoustic signals. Abstract in *Journal of the Acoustical Society of America* **32**: 1,493.

Mathews, M. 1961. An acoustical compiler for music and psychological stimuli. *Bell System Technical Journal* **40**: 677–94.

Mathews, M. 1969. *The Technology of Computer Music*. Cambridge, MA: MIT Press.

Moore, F. R. 1990. *Elements of Computer Music*. Englewood Cliffs, NJ: Prentice-Hall.

Pope, S. 1991. Machine Tongues IX: object-oriented software design. In S. Pope (ed.) *The Well-Tempered Object*, pp. 32–48. Cambridge, MA: MIT Press.

Pope, S. 1993. Machine Tongues XV: three packages for software sound synthesis. *Computer Music Journal* **17**(2).

Scalletti, C. 1991. The Kyma/Platypus computer music workstation. In S. Pope (ed.) *The Well-Tempered Object*, pp. 119–40. Cambridge, MA: MIT Press.

Schottstaedt, W. 1992. *CLM Manual*. HTML reference manual available at http://ccrma-www.stanford.edu/CCRMA/software/CLM/clm-manual/clm.html

Stroustrup, B. 1995. *The C++ Programming Language*. Reading, MA: Addison-Wesley Publishing Co.

Vercoe, B., and Piche, J. 1997. *Csound Manual (version 3.47)*. HTML reference manual available at
ftp://ftp.maths.bath.ac.uk/pub/dream

Wampler, B. 1998. *V Reference Manual*. HTML document available at http://www.objectcentral.com