LAC2005 Proceedings

3rd International Linux Audio Conference

April 21 – 24, 2005

ZKM | Zentrum für Kunst und Medientechnologie

Karlsruhe, Germany

Zentrum für Kunst und Medientechnologie Karlsruhe

Linux Audio Developers

Published by
ZKM | Zentrum für Kunst und Medientechnologie
Karlsruhe, Germany
April, 2005
All copyright remains with the authors
www.zkm.de/lac/2005

Content

Preface

We are very happy to welcome you to the 3rd International Linux Audio Conference. It takes place again at the ZKM | Institute for Music and Acoustics in Karlsruhe/Germany, on April 21-24, 2005.

The "Call for Papers" which has resulted in the proceedings you hold in your hands has changed significantly compared to the previous LAC conferences because this time we were asking for elaborated papers rather than short abstracts only. We are very glad that in spite of this new hurdle we have received quite a lot of paper submissions and we are confident that many people will appreciate the efforts which the authors have put into them. Each paper has been reviewed by at least 2 experts. Many thanks go to the authors and reviewers for their great work!

We hope that the 2005 conference will be as successful and stimulating as the previous ones and we wish you all a pleasant stay.

Frank Neumann and Götz Dipper
Organization Team LAC2005
Karlsruhe, April 2005

The International Linux Audio Conference 2005 is sponsored by

ALCATEL SPACE

KEYBOARDS RECORDING & COMPUTER

Staff

Organization Team LAC2005

Götz Dipper	ZKM | Institute for Music and Acoustics
Frank Neumann	LAD

ZKM

Marc Riedel	Organization of LAC2005 Concerts/Call for Music
Jürgen Betker	Graphic Artist
Hartmut Bruckner	Sound Engineer
Ludger Brümmer	Head of the Institute for Music and Acoustics
Uwe Faber	Head of the IT Department
Hans Gass	Technical Assistant
Joachim Goßmann	Tonmeister
Achim Heidenreich	Event Management
Martin Herold	Technical Assistant
Martin Knötzele	Technical Assistant
Andreas Liefländer	Technical Assistant
Philipp Mattner	Technical Assistant
Alexandra Mössner	Assistant of Management
Caro Mössner	Event Management
Chandrasekhar Ramakrishnan	Software Developer
Martina Riedler	Head of the Event Department
Thomas Saur	Sound Engineer
Theresa Schubert	Event Management
Joachim Schütze	IT Department
Bernhard Sturm	Production Engineer
Manuel Weber	Technical Director of the Event Department
Monika Weimer	Event Management
Susanne Wurmnest	Event Management

LAD

Matthias Nagorni	SUSE LINUX Products GmbH
Jörn Nettingsmeier	Coordination of the Internet Audio Stream

Relay Servers

Marco d'Itri	Italian Linux Society
Eric Dantan Rzewnicki	Radio Free Asia, Washington

Chat Operator

Sebastian Raible

Icecast/Ices Support

Jan Gerber	
Karl Heyes	Xiph.org

Paper Reviews

Fons Adriaensen	Alcatel Space, Antwerp/Belgium
Frank Barknecht	Deutschlandradio, Köln/Germany
Ivica Ico Bukvic	University of Cincinnati, Ohio/USA
Paul Davis	Linux Audio Systems, Pennsylvania/USA
François Déchelle	France
Steve Harris	University of Southampton, Hampshire/UK
Jaroslav Kysela	SUSE LINUX Products GmbH, Czech Republic
Fernando Lopez-Lezcano	CCRMA/Stanford University, California/USA
Jörn Nettingsmeier	Folkwang-Hochschule Essen/Germany
Frank Neumann	Karlsruhe/Germany
Dave Phillips	Findlay, Ohio/USA

(In alphabetical order)

MidiKinesis — MIDI controllers for (almost) any purpose

Peter Brinkmann
Technische Universität Berlin
Fakultät II – Mathematik und Naturwissenschaften
Institut für Mathematik
Sekretariat MA 3-2
Straße des 17. Juni 136
D-10623 Berlin
`brinkman@math.tu-berlin.de`

Abstract

MidiKinesis is a Python package that maps MIDI control change events to user-defined X events, with the purpose of controlling almost any graphical user interface using the buttons, dials, and sliders on a MIDI keyboard controller such as the Edirol PCR-30. Key ingredients are Python modules providing access to the ALSA sequencer as well as the XTest standard extension.

Keywords

ALSA sequencer, MIDI routing, X programming, Python

1 Introduction

When experimenting with Matthias Nagorni's AlsaModularSynth, I was impressed with its ability to bind synth parameters to MIDI events on the fly. This feature is more than just a convenience; the ability to fine-tune parameters without having to go back and forth between the MIDI keyboard and the console dramatically increases productivity because one can adjust several parameters almost simultaneously, without losing momentum by having to navigate a graphical user interface. Such a feature can make the difference between settling for a "good enough" choice of parameters and actually finding the "sweet spot" where everything sounds just right.

Alas, a lot of audio software for Linux does not expect any MIDI input, and even in programs that can be controlled via MIDI, the act of setting up a MIDI controller tends to be less immediate than the elegant follow-and-bind approach of AlsaModularSynth. I set out to build a tool that would map MIDI control change events to GUI events, and learn new mappings on the fly. The result was MidiKinesis, the subject of this note. MidiKinesis makes it possible to control almost any graphical user interface from a MIDI controller keyboard.

2 Basics

Before implementing MidiKinesis, I settled on the following basic decisions:

- MidiKinesis will perform all its I/O through the ALSA sequencer (in particular, no reading from/writing to `/dev/midi*`), and it will act on graphical user interfaces by creating plain X events.
- MidiKinesis will be implemented in Python (Section 8), with extensions written in C as necessary. Dependencies on nonstandard Python packages should be minimized.

Limiting the scope to ALSA and X effectively locks MidiKinesis into the Linux platform (it might work on a Mac running ALSA as well as X, but this seems like a far-fetched scenario), but I decided not to aim for portability because MidiKinesis solves a problem that's rather specific to Linux audio.[1] The vast majority of Mac or Windows users will do their audio work with commercial tools like Cubase or Logic, and their MIDI support already is as smooth as one could possibly hope. The benefit of limiting MidiKinesis in this fashion is a drastic simplification of the design; at the time of this writing, MidiKinesis only consists of about 2000 lines of code.

When I started thinking about this project, my first idea was to query various target programs in order to find out what widgets their user interfaces consist of, but this approach turned out to be too complicated. Ultimately, it would have required individual handling of toolkits like GTK, Qt, Swing, etc., and in many cases the required information would not have been forthcoming. So, I decided to settle for

[1] I did put a thin abstraction layer between low-level implementation details and high-level application code (Section 3), so that it is theoretically possible to rewrite the low-level code for Windows or Macs without breaking the application code.

the lowest common denominator — MidiKinesis directly operates on the X Window System, using the XTest standard extension to generate events. I expected this approach to be somewhat fragile as well as tedious to calibrate, but in practice it works rather well (Section 5).

For the purposes of MidiKinesis, Python seemed like a particularly good choice because it is easy to extend with C (crucial for hooking into the ALSA sequencer and X) and well suited for rapidly building MIDI filters, GUIs, etc. Python is easily fast enough to deal with MIDI events in real time, so that performance is not a concern in this context. On top of Python, MidiKinesis uses Tkinter (the de facto standard toolkit for Python GUIs, included in many Linux distributions), and ctypes (Section 8) (not a standard module, but easy to obtain and install).

3 The bottom level

At the lowest level, the modules `pyseq.py` and `pyrobot.py` provide access to the ALSA sequencer and the XTest library, respectively. There are many ways of extending Python with C, such as the Python/C API, Boost.Python, automatic wrapper generators like SIP or SWIG, and hybrid languages like pyrex. In the end, I chose ctypes because of its ability to create and manipulate C structs and unions. This is crucial for working with ALSA and X since both rely on elaborate structs and unions (`snd_seq_event_t` and `XEvent`) for passing events.

3.1 Accessing the ALSA sequencer

The module `pyseq.py` defines a Python shadow class that provides access to the full sequencer event struct of ALSA (`snd_seq_event_t`). Moreover, the file `pyseq.c` provides a few convenience functions, most importantly `midiLoop(...)`, which starts a loop that waits for MIDI events, and calls a Python callback function when a MIDI event comes in.

The module `pyseq.py` also provides an abstraction layer that protects application programmers from such implementation details. To this end, `pyseq.py` introduces the following classes:

PySeq manages ALSA sequencer handles, and it provides methods for creating MIDI ports, sending MIDI events, etc. Application programmers will subclass PySeq and override the methods `init` (called by the constructor) and `callback` (called when a MIDI event arrives at an input port of the corresponding sequencer).

MidiThread is a subclass of `threading.Thread` that provides support for handling incoming MIDI events in a separate thread. An instance of MidiThread keeps a pointer to an instance of PySeq whose callback method is called when a MIDI event comes in.

Using this class structure, a simple MIDI filter might look like this:

```
from pyseq import *

class MidiTee(PySeq):
  def init(self, *args):
    self.createInPort()
    self.out=self.createOutPort()
  def callback(self, ev):
    print ev
    self.sendEvent(ev, self.out)
    return 1

seq=MidiTee('miditee')
t=MidiThread(seq)
t.start()
raw_input('press enter to finish')
```

This filter acts much like the venerable `tee` command. It reads MIDI events from its input port and writes them verbatim to its output port, and it prints string representations of MIDI events to the console. The last line is necessary because instances of MidiThread are, by default, daemon threads.

Once started, an instance of MidiThread will spend most of its time in the C function `midiLoop`, which in turn spends most of its time waiting for events in a `poll(...)` system call. In other words, instances of MidiThread hardly put any strain on the CPU at all.

3.2 Capturing and sending X events

Structurally, the module `pyrobot.py` is quite similar to `pyseq.py`. It uses ctypes to create a Python shadow class for the XEvent union of X, and it introduces a simple abstraction layer that protects application programmers from such details. The main classes are as follows:

PyRobot is named after Java's java.awt.Robot. It provides basic functionality for capturing and sending X events.

Script uses PyRobot to record and play sequences (scripts) of mouse and keyboard events.

The following code records a sequence of mouse and keyboard events and replays it.

```
from pyrobot import *

R=PyRobot()
S=Script()
print 'record script, press Escape'
S.record(R)
raw_input('press enter to replay')
S.play(R)
```

4 The top level

The module `midikinesis.py` is the main application of the package. It waits for incoming MIDI control change events. If it receives a known event, it triggers the appropriate action. Otherwise, it asks the user to assign one of six possible mappings to the current event:

Button maps a button on the MIDI keyboard to a sequence of X events (Section 3.2) that the user records when setting up this sort of mapping. This sequence will typically consist of mouse motions, clicks, and keystrokes, but mouse dragging events are also admissible.

It is also possible to assign Button mappings to dials and sliders on the keyboard. To this end, one defines a threshold value between 0 and 127 (64 is the default) and chooses whether a rising edge or a falling edge across the threshold is to trigger the recorded sequence of X events.

Slider maps MIDI events from a slider or dial on the MIDI keyboard to mouse dragging events along a linear widget (such as a linear volume control or scrollbar). For calibration purposes, `midikinesis.py` asks the user to click on the bottom, top, and current location of the widget. Hydrogen and jamin are examples of audio software with widgets of this kind. The button used to click on the bottom location determines the button used for dragging.

Selection divides the range of controller values (0 . . . 127) into brackets, with each bracket corresponding to a mouse click on a user-defined location on the screen. It primarily targets radio buttons.

Counter handles counter widgets whose values are changed by repeatedly clicking on up/down arrows. Specimen uses widgets of this kind. Counter mappings are calibrated by clicking on the location of the up/down arrows and by specifying a step size, i.e., the number of clicks that a unit change of the controller value corresponds to.

Circular Dial behaves much like Slider, except it drags the mouse in a circular motion. amSynth has dials that work in this fashion.

Linear Dial sounds like an oxymoron, but the name merely reflects the dual nature of the widgets that it targets. To wit, there are widgets that look like dials on the screen, but they get adjusted by pressing the mouse button on the widget and dragging the mouse up or down in a linear motion. Rosegarden4, ZynAddSubFX, and Hydrogen all have widgets of this kind.

These six mappings cover most widgets that commonly occur in Linux audio software. Button mappings are probably the most general as well as the least obvious feature. Here is a simple application of some of Button mappings, using Rosegarden4:

- Map three buttons on the MIDI keyboard to mouse clicks on Start, Stop, and Record in Rosegarden4.
- Map a few more buttons to mouse clicks on different tracks in Rosegarden4, followed by the Delete key.

Pressing one of the latter buttons will activate and clear a track, so that it is possible to record a piece consisting of several tracks (and to record many takes of each track) without touching the console after the initial calibration.

5 Subtleties

One might expect the approach of mapping MIDI events to plain X events to be rather fragile because simple actions like moving, resizing, or covering windows may break existing mappings. In practice, careful placement of application windows on the desktop will eliminate most problems of this kind. Moreover, MidiKinesis provides a number of mechanisms that increase robustness:

- MidiKinesis computes coordinates of X events relative to a reference point. If all mappings target just one window, then one makes the upper left-hand corner of this window the reference point, and if the window moves, one only needs to update the reference point.

- The notion of reference points also makes it possible to save mappings to a file and restore them later.
- It helps to start one instance of MidiKinesis for each target window, each with its individual reference point. This will resolve most window placement issues. In order to make this work, one launches and calibrates them individually. When an instance of MidiKinesis has been set up, one tells it to ignore unknown events and moves on to the next instance. Like this, instances of MidiKinesis won't compete for the same MIDI events.
- Instances of MidiKinesis only accept events whose channel and parameter match certain patterns.[2] By choosing different patterns for different instances of MidiKinesis, one keeps them from interfering with each other, while each still accepts new mappings.

6 Fringe benefits

Using the module `pyseq.py`, one can rapidly build MIDI filters. I even find the simple MidiTee example from Section 3.1 useful for eavesdropping on conversations between MIDI devices (tools like amidi serve a similar purpose, but I like being able to adjust the output format on the fly, depending on what I'm interested in).

One of the first applications I built on top of `pyseq.py` was a simple bulk dump handler for my MIDI keyboard. It shouldn't be much harder to build sophisticated configuration editors for various MIDI devices in a similar fashion.

I was surprised to find out that there seemed to be no approximation of java.awt.Robot for Python and Linux before `pyrobot.py`, so that `pyrobot.py` might be useful in its own right, independently of audio applications.

7 Where to go from here

While the focus of MidiKinesis has squarely been on audio applications, it also opens the possibility of using MIDI events to control all kinds of software. For instance, when I was implementing Slider mappings, I used the vertical scrollbar of Firefox as a test case.

In order to illustrate the basic idea of creative misuse of MIDI events, I implemented a simple game of Pong, controlled by a slider or dial on a MIDI keyboard. Generally speaking, a mouse is a notoriously sloppy input device, while MIDI controllers tend to be rather precise. So, a tool like MidiKinesis might be useful in a nonaudio context requiring speed and accuracy.

Finally, I feel that the applications of MidiKinesis that I have found so far barely scratch the surface. For instance, the ability to record and replay (almost) arbitrary sequences of X events has considerable potential beyond the examples that I have tried so far.

8 Resources

MidiKinesis is available at `http://www.math.tu-berlin.de/~brinkman/software/midikinesis/midikinesis.tgz`. MidiKinesis requires Python, Tkinter, and ctypes, as well as an X server that supports the XTest standard extension.

Python is a powerful scripting language, available at `http://www.python.org/`. Chances are that you are using a Linux distribution that already has Python installed.

Tkinter is the de facto standard toolkit for Python GUIs, available at `http://www.python.org/moin/TkInter`. It is included in many popular Linux distributions.

ctypes is a package to create and manipulate C data types in Python, and to call functions in shared libraries. It is available at `http://starship.python.net/crew/theller/ctypes/`.

9 Acknowledgements

Special thanks go to Holger Pietsch for getting me started on the X programming part of the project, and to the members of the LAU and LAD mailing lists for their help and support.

[2]By default, MidiKinesis accepts events from General Purpose Controllers on any channel.

Extensions to the Csound Language: from User-Defined to Plugin Opcodes and Beyond.

Victor Lazzarini
Music Technology Laboratory
National University of Ireland, Maynooth
Victor.Lazzarini@nuim.ie

Abstract

This article describes the latest methods of extending the csound language. It discusses these methods in relation to the two currently available versions of the system, 4.23 and 5. After an introduction on basic aspects of the system, it explores the methods of extending it using facilities provided by the csound language itself, using user-defined opcodes. The mechanism of plugin opcodes and function table generation is then introduced as an external means of extending csound. Complementing this article, the fsig signal framework is discussed, focusing on its support for the development of spectral-processing opcodes.

Keywords: Computer Music, Music Processing Languages, Application Development, C / C++ Programming

1 Introduction

The csound (Vercoe 2004) music programming language is probably the most popular of the text-based audio processing systems. Together with cmusic (Moore 1990), it was one of the first modern C-language-based portable sound compilers (Pope 1993), but unlike it, it was adopted by composers and developers world-wide and it continued to develop into a formidable tool for sound synthesis, processing and computer music composition. This was probably due to the work of John Ffitch and others, who coordinated a large developer community who was ultimately responsible for the constant upgrading of the system. In addition, the work of composers and educators, such as Richard Boulanger, Dave Phillips and many others, supported the expansion of its user base, who also has been instrumental in pushing for new additions and improvements. In summary, csound can be seen as one of the best examples of music open-source software development, whose adoption has transcended a pool of expert-users, filtering into a wider music community.

The constant development of csound has been partly fuelled by the existence of a simple opcode API (Fftich 2000) (Resibois 2000), which is easy to understand, providing a good, if basic, support for unit generator addition. This was, for many years, the only direct means of extending csound for those who were not prepared to learn the inside details of the code. In addition, the only way of adding new unit generators to csound was to include them in the system source code and re-build the system, as there was no support for dynamically-loadable components (csound being from an age where these concepts had not entered mainstream software development). Since then, there were some important new developments in the language and the software in general providing extra support for extensions. These include the possibility of language extension both in terms of C/C++-language loadable modules and in csound's own programming language. Another important development has been the availability of a more complete C API (Goggins et al 2004), which can be used to instantiate and control csound from a calling process, opening the door for the separation of language and processing engine.

2 Csound versions

Currently there are two parallel versions of the so-called canonical csound distribution, csound 4.23, which is a code-freeze version from 2002 , and csound 5, a re-modelled system, still in beta stage of development. The developments mentioned in the introduction are present in csound 4.23, but have been further expanded in version 5. In this system, apart from the core opcodes, most of the unit generators are now in loadable library modules and further opcode addition should be in that format. The plugin opcode mechanism is already present in version 4.23, although some differences exist between opcode formats for the two versions. These are mainly to do with arguments to functions and return types. There is also now a mechanism for dynamic-library function tables and an improved/expanded csound API. Other changes brought about in csound 5 are the move to the use of external libraries for soundfile, audio IO and MIDI.

Csound 4.23 is the stable version of csound, so at this moment, it would be the recommended one for general use and, especially, for new users. Most of the mechanisms of language extension and unit generator development discussed in this paper are supported by this version. For Linux users, a GNU building system-based source package is available for this version, making it simple to configure and install the program on most distributions. It is important to also note that csound 5 is fully operational, although with a number of issues still to be resolved. It indeed can be used by anyone, nevertheless we would recommend it for more experienced users. However, the user input is crucial to csound 5 development, so the more users adopting the new version, the better for its future.

3 Extending the language

As mentioned earlier, csound has mechanisms for addition of new components both by writing code in the csound language itself and by writing C/C++ language modules. This section will concentrate on csound language-based development, which takes the basic form of user-defined opcodes. Before examining these, a quick discussion of csound data types, signals and performance characteristics is offered

3.1 Data types and signals

The csound language provides three basic data types: i-, k- and a-types. The first is used for initialisation variables, which will assume only one value in performance, so once set, they will usually remain constant throughout the instrument code. The other types are used to hold scalar (k-type) and vectorial (a-type) variables. The first will hold a single value, whereas the second will hold an array of values (a vector) and internally, each value is a floating-point number, either 32- or 64-bit, depending on the version used.

A csound instrument code can use any of these variables, but opcodes might accept specific types as input and will generate data in one of those types. This implies that opcodes will execute at a certain update rate, depending on the output type (Ekman 2000). This can be at the audio sampling rate (sr), the control rate (kr) or only at initialisation time. Another important aspect is that csound instrument code effectively has a hidden processing loop, running at the control-rate and affecting (updating) only control and audio signals. An instrument will execute its code lines in that loop until it is switched off
Under this loop, audio variables, holding a block of samples equivalent to sr/kr (ksmps), will have their whole vector updated every pass of the loop:

```
instr 1    /* start of the loop */

iscl = 0.5 /* i-type, not affected by
              the loop */
asig  in  /* copies ksmps  samples from
             input buffer into asig  */
atten = asig*iscl /* scales every sample
                    of  asig with iscl */
out  atten /* copies kmsps samples from
              atten into output buffer */

endin     /* end of the loop */
```

This means that code that requires sample-by-sample processing, such as delays that are smaller than one control-period, will require setting the a-rate vector size, ksmps, to 1, making kr=sr. This will have a detrimental effect on performance, as the efficiency of csound depends a lot on the use of different control and audio rates.

3.2 User-defined opcodes

The basic method of adding unit generators in the csound language is provided by the user-defined opcode (UDO) facility, added by Istvan Varga to csound 4.22. The definition for a UDO is given using the keywords opcode and endop, in a similar fashion to instruments:

```
opcode  NewUgen  a,aki
/* defines an a-rate opcode, taking a,
   k and i-type inputs */
endop
```

The number of allowed input argument types is close to what is allowed for C-language opcodes. All p-field values are copied from the calling instrument. In addition to a-,k- and i-type arguments (and 0, meaning no inputs), which are audio, control and initialisation variables, we have: K, control-rate argument (with initialisation); plus o, p and j (optional arguments, i-type variables defaulting to 0,1 and -1). Output is permitted to be to any of a-, k- or i-type variables. Access to input and output is simplified through the use of a special pair of opcodes, `xin` and `xout`. UDOs will have one extra argument in addition to those defined in the declaration, the internal number of the a-signal vector samples `iksmps`. This sets the value of a local control rate (sr/iksmps) and defaults to 0, in which case the `iksmps` value is taken from the caller instrument or opcode.

The possibility of a different a-signal vector size (and different control rates) is an important aspect of UDOs. This enables users to write code that requires the control rate to be the same as audio rate, without actually having to alter the global values for these parameters, thus improving efficiency. An opcode is also provided for setting the iksmps value to any given constant:

```
setksmps 1 /* sets a-signal vector to 1,
            making kr=sr */
```

The only caveat is that when the local ksmps value differs from the global setting, UDOs are not allowed to use global a-rate operations (global variable access, etc.). The example below implements a simple feedforward filter, as an example of UDO use:

```
#define LowPass 0
#define HighPass 1

opcode  NewFilter  a,aki

 setksmps  1     /* kr = sr */
 asig,kcoef,itype  xin
 adel init 0

 if itype == HighPass then
  kcoef = -kcoef
 endif

 afil  =  asig + kcoef*adel
 adel = asig  /* 1-sample delay,
             only because kr = sr */
     xout   afil

endop
```

Another very important aspect of UDOs is that recursion is possible and only limited to available memory. This allows, for instance, the implementation of recursive filterbanks, both serial or parallel, and similar operations that involve the spawning of unit generators. The UDO facility has added great flexibility to the csound language, enabling the fast development of musical signal processing operations. In fact, an on-line UDO database has been made available by Steven Yin, holding many interesting new operations and utilities implemented using this facility (www.csounds.com/udo). This possibly will form the foundation for a complete csound-language-based opcode library.

3.3 Adding external components

Csound can be extended in variety of ways by modifying its source code and/or adding elements to it. This is something that might require more than a passing acquaintance with its workings, as a rebuild of the software from its complete source code. However, the addition of unit generators and function tables is generally the most common type of extension to the system. So, to facilitate this, csound offers a simple opcode development API, from which new dynamically-loadable (‘plugin’) unit generators can be built. In addition, csound 5 also offers a similar mechanism for function tables. Opcodes can be written in the C or C++ language. In the latter, the opcode is written as a class derived from a template (‘pseudo-virtual’) base class `OpcodeBase`, whereas in the former, we normally supply a C module according to a basic description. The following sections will describe the process of adding an opcode in the C language. An alternative C++ class implementation would employ a similar method.

3.3.1 Plugin opcodes

C-language opcodes normally obey a few basic rules and their development require very little in terms of knowledge of the actual processes involved in csound. Plugin opcodes will have to provide three main programming components: a data structure to hold the opcode internal data, an initialising function or method, and a processing function or method. From an object-oriented perspective, all we need is a simple class, with its members, constructor and perform methods. Once these elements are supplied, all we need to do is to add a line telling csound what type of opcode it is, whether it is an i-, k- or a-rate based unit generator and what arguments it takes.

The data structure will be organised in the following fashion:

1. The OPDS data structure, holding the common components of all opcodes.
2. The output pointers (one MYFLT pointer for each output)
3. The input pointers (as above)
4. Any other internal dataspace member.

The csound opcode API is defined by csdl.h, which should be included at the top of the source file. The example below shows the data structure for same filter implemented in previous sections:

```
#include "csdl.h"

typedef struct  _newflt {
OPDS  h;
MYFLT *outsig;/* output pointer  */
MYFLT *insig,*kcoef,*itype;/* input
                              pointers */
MYFLT  delay;  /* internal variable,
                 the 1-sample delay */
int    mode;   /* filter mode */
} newfilter;
```

The initialisation function is only there to initialise any data, such as the 1-sample delay, or allocate memory, if needed. The new plugin opcode model in csound5 expects both the initialisation function and the perform function to return an int value, either OK or NOTOK. In addition, both methods now take a two arguments: pointers to the csound environment and the opcode dataspace. In version 4.23 the opcode function will only take the pointer to the opcode dataspace as argument. The following example shows an initialisation function in csound 5 (all following examples are also targeted at that version):

```
int newfilter_init(ENVIRON *csound,
                    newfilter *p){
p->delay = (MYFLT) 0;
p->mode = (int) *p->itype;
return OK;
}
```

The processing function implementation will depend on the type of opcode that is being created. For audio rate opcodes, because it will be generating audio signal vectors, it will require an internal loop to process the vector samples. This is not necessary with k-rate opcodes, as we are dealing with scalar inputs and outputs, so the function has to process only one sample at a time. This means that, effectively, all processing functions are called every control period. The filter opcode is an audio-rate unit generator, so it will include the internal loop.

```
int newfilter_process(ENVIRON *csound,
                       newfilter *p){
int i;
/* signals in, out */
MYFLT *in = p->insig;
MYFLT *out = p->outsig;
/* control input */
MYFLT  coef = *p->kcoef;
/* 1-sample  delay  */
MYFLT delay = *p->delay;
MYFLT temp;

if(p->mode)coef = -coef;

/* processing loop    */
for(i=0; i < ksmps; i++){
    temp = in[i];
    out[i] = in[i] + delay*coef ;
    delay = temp;
}
/* keep delayed sample for next time */
*p->delay = delay;

return OK;
}
```

To complete the source code, we fill an opcode registration structure OENTRY array called localops (static), followed by the LINKAGE macro:

```
static OENTRY localops[] = {
{ "newfilter", S(newfilter),  5,  "a",
"aki",  (SUBR)newfilter_init,  NULL,
(SUBR)newfilter_process }
};

LINKAGE
```

The OENTRY structure defines the details of the new opcode:

1. the opcode name (a string without any spaces).
2. the size of the opcode dataspace, set using the macro S(struct_name), in most cases; otherwise this is a code indicating that the opcode will have more than one implementation, depending on the type of input arguments.
3. An int code defining when the opcode is active: 1 is for i-time, 2 is for k-rate and 4 is for a-rate. The actual value is a combination of one or more of those. The value of 5 means active at i-time (1) and a-rate (4). This means that the opcode has an init function and an a-rate processing function.
4. String definition the output type(s): a, k, s (either a or k), i, m (multiple output arguments), w or f (spectral signals).
5. Same as above, for input types: a, k, s, i, w, f, o (optional i-rate, default to 0), p (opt, default to 1), q (opt, 10), v(opt, 0.5), j(opt, –1), h(opt, 127), y (multiple inputs, a-type), z (multiple inputs, k-type), Z (multiple inputs, alternating k- and a-types), m (multiple inputs, i-type), M (multiple inputs, any type) and n (multiple inputs, odd number of inputs, i-type).
6. I-time function (init), cast to (SUBR).
7. K-rate function.
8. A-rate function.

The LINKAGE macro defines some functions needed for the dynamic loading of the opcode. This macro is present in version 5 csdl.h, but not in 4.23 (in which case the functions need to be added manually):

```
#define LINKAGE long opcode_size(void) \
{ return sizeof(localops);}   \
OENTRY *opcode_init(ENVIRON *xx)   \
{ return localops;}           \
```

The plugin opcode is build as a dynamic module, and similar code can be used both with csound versions 4.23 or 5:

```
gcc -02 -c opsrc.c -o opcode.o
ld -E --shared opcode.o –o opcode.so
```

However, due to differences in the interface, the binaries are not compatible, so they will need to built specificially for one of the two versions.Another difference is that csound 5 will load automatically all opcodes in the directory set with the environment variable OPCODEDIR, whereas version 4.23 needs the flag –opcode-lib=*myopcode.so* for loading a specific module.

3.3.2 Plugin function tables

A new type of dynamic module, which has been introduced in csound 5 is the dynamic function table generator (GEN). Similarly to opcodes, function table GENs were previously only included statically with the rest of the source code. It is possible now to provide them as dynamic loadable modules. This is a very recent feature, introduced by John Ffitch at the end of 2004, so it has not been extensively tested. The principle is similar to plugin opcodes, but the implementation is simpler. It is only necessary to provide the GEN routine that the function table implements. The example below shows the test function table, written by John Ffitch, implementing a hyperbolic tangent table:

```
#include "csdl.h"
#include <math.h>

void tanhtable(ENVIRON *csound,
      FUNC *ftp, FGDATA *ff,)
{
/* the function table */
MYFLT fp = ftp->ftable;
/* f-statement p5, the range */
MYFLT range = ff->e.p[5];
/* step is range/tablesize */
double step = (double)
              range/(ff->e.p[3]);
int i;
double x;
  /* table-filling loop   */
   for(i=0, x=FL(0.0); i<ff->e.p[3];
       i++,x+=step)
        *fp++ = (MYFLT)tanh(x);
}
```

The GEN function takes three arguments, the csound environment dataspace, a function table pointer and a gen info data pointer. The former holds the actual table, an array of MYFLTs, whereas the latter holds all the information regarding the table, e.g. its size and creation arguments. The `FGDATA` member `e` will hold a numeric array (`p`) with all p-field data passed from the score f-statement (or `ftgen` opcode).

```
static NGFENS localfgens[] = {
   { "tanh", (void(*)(void))tanhtable},
   { NULL, NULL}
};
```

The structure NFGENS holds details on the function table GENs, in the same way as OENTRY holds opcode information. It contains a string name and a pointer to the GEN function. The localfgens array is initialised with these details and terminated with NULL data. Dynamic GENs are numbered according to their loading order, starting from GEN 44 (there are 43 'internal' GENs in csound 5).

```
#define S sizeof
static OENTRY *localops = NULL;
FLINKAGE
```

Since opcodes and function table GENs reside in the same directory and are loaded at the same time, setting the *localops array to NULL, will avoid confusion as to what is being loaded. The FLINKAGE macro works in the same fashion as LINKAGE.

4 Spectral signals

As discussed above, Csound provides data types for control and audio, which are all time-domain signals. For spectral domain processing, there are two separate signal types, 'wsig' and 'fsig'. The former is a signal type introduced by Barry Vercoe to hold a special, non-standard, type of logarithmic frequency analysis data and is used with a few opcodes originally provided for manipulating this data type. The latter is a self-describing data type designed by Richard Dobson to provide a framework for spectral processing, in what is called streaming phase vocoder processes (to differentiate it from the original csound phase vocoder opcodes). Opcodes for converting between time-domain audio signals and fsigs, as well as a few processing opcodes, were provided as part of the original framework by Dobson. In addition, support for a self-describing, portable, spectral file format PVOCEX (Dobson 2002) has been added to csound, into the analysis utility program pvanal and with a file reader opcode. A library of processing opcodes, plus a spectral GEN, has been added to csound by this author. This section will explore the fsig framework, in relation to opcode development.

Fsig is a self-describing csound data type which will hold frames of DFT-based spectral analysis data. Each frame will contain the positive side of the spectrum, from 0 Hz to the Nyquist (inclusive). The framework was designed to support different spectral formats, but at the moment, only an amplitude-frequency format is supported, which will hold pairs of floating-point numbers with the amplitude and frequency (in Hz) data for each DFT analysis channel (bin). This is probably the most musically meaningful of the DFT-based output formats and is generated by Phase Vocoder (PV) analysis. The fsig data type is defined by the following C structure:

```
typedef struct pvsdat {
/* framesize-2, DFT length */
long N;
/* number of frame overlaps */
long overlap;
/* window size */
long winsize;
/* window type: hamming/hanning */
int  wintype;
/* format: cur. fixed to AMP:FREQ */
long format;
/* frame counter   */
unsigned long   framecount;
/* spectral sample is a 32-bit float */
AUXCH frame;
} PVSDAT;
```

The structure holds all the necessary data to describe the signal type: the DFT size (N), which will determine the number of analysis channels (N/2 + 1) and the framesize; the number of overlaps, or decimation, which will determine analysis hopsize (N/overlaps); the size of the analysis window, generally the same as N; the window type, currently supporting PVS_WIN_HAMMING or PVS_WIN_HANN; the data format, currently only PVS_AMP_FREQ; a frame counter, for keeping track of processed frames; and finally the AUXCH structure which will hold the actual array of floats with the spectral data. The AUXCH structure and associated functions are provided by csound as a mechanism for dynamic memory allocation and are used whenever such operation is required. A number of other utility functions are provided by the csound opcode API (in csdl.h), for operations such as loading, reading and writing files, accessing function tables, handling string arguments, etc.. Two of these are used in the code below to provide simple error notification and handling (`initerror()` and `perferror()`).

A number of implementation differences exist between spectral and time-domain processing opcodes. The main one is that new output is only produced if a new input frame is ready to be processed. Because of this implementation detail, the processing function of a streaming PV opcode is actually registered as a k-rate routine. In addition, opcodes allocate space for their fsig frame outputs, unlike ordinary opcodes, which simply take floating-point buffers as input and output. The fsig dataspace is externally allocated, in similar fashion to audio-rate vectors and control-rate scalars; however the DFT frame allocation is done by the opcode generating the signal. With that in mind, and observing that type of data we are processing is frequency-domain, we can implement a spectral unit generator as an ordinary (k-rate) opcode. The following example is a frequency-domain version of the simple filter implemented in the previous sections:

```
#include "csdl.h"
#include "pstream.h" /* fsig definitions
*/

typedef struct _pvsnewfilter {
OPDS    h;
/* output fsig, its frame needs to be
   allocated */
PVSDAT  *fout;
PVSDAT  *fin;   /* input fsig */
/* other opcode args  */
MYFLT   *coef, *itype;
MYFLT    mode;  /* filter type */
unsigned long lastframe;
} pvsnewfilter;

int pvsnewfilter_init(ENVIRON *csound,
                    pvsnewfilter *p)
{
long N = p->fin->N;
p->mode = (int) *p->itype;
/* this allocates an AUXCH struct, if
   non-existing */
if(p->fout->frame.auxp==NULL)
  auxalloc((N+2)*sizeof(float),
          &p->fout->frame);
/* output fsig description */
p->fout->N =  N;
p->fout->overlap = p->fin->overlap;
p->fout->winsize = p->fin->winsize;
p->fout->wintype = p->fin->wintype;
p->fout->format = p->fin->format;
p->fout->framecount = 1;
p->lastframe = 0;

/* check format */
if (!(p->fout->format==PVS_AMP_FREQ ||
      p->fout>format==PVS_AMP_PHASE))
return  initerror("wrong format\n");
/* initerror is a utility csound
   function */

return OK;
}
```

The opcode dataspace contains pointers to the output and input fsig, as well as the k-rate coefficient and the internal variable that holds the filter mode. The init function has to allocate space for the output fsig DFT frame, using the csound opcode API function `auxalloc()`, checking first if it is not there already.

```
int pvsnewfilter_process(ENVIRON *csound,
                          pvsnewfilter p)
{
 long i,N = p->fout->N;
 MYFLT cosw, tpon;
 MYFLT coef = *p->kcoef;
 float *fin = (float *)
          p->fin >frame.auxp;
float *fout = (float *)
          p->fout->frame.auxp;

if(fout==NULL)
 return perferror("not initialised\n");
/* perferror is a utility csound
   function */

if(mode) coef = -coef;
/* if a new input frame is ready  */
if(p->lastframe <
   p->fin->framecount) {
 /* process the input, filtering */
  pon = pi/N; /* pi is global*/
  for(i=0;i < N+2;i+=2) {
    cosw = cos(i*pon);
    /* amps */
    fout[i] =  fin[i] *
       sqrt(1+coef*coef+2*coef*cosw);
    /* freqs: unchanged */
    fout[i+1] = fin[i+1];
      }
  /* update the framecount  */
   p->fout->framecount =
   p->lastframe = p->fin->framecount;
    }
return OK;
}
```

The processing function keeps track of the frame count and only processes the input, generating a new output frame, if a new input is available. The framecount is generated by the analysis opcode and is passed from one processing opcode to the next in the chain. As mentioned before, the processing function is called every control-period, but it is independent of it, only performing when needed. The only caveat is that the fsig framework requires the control period in samples (ksmps) to be smaller or equal to the analysis hopsize. Finally, the localops OENTRY structure for this opcode will look like this:

```
static OENTRY localops[] = {
 {"pvsnewfilter", S(pvsnewfilter), 3,
  "f", "fkp", (SUBR)pvsnewfilter_init,
  (SUBR)pvsnewfilter_process}
};
```

From the above, it is clear to see that the new opcode is called pvsnewfilter and its implementation is made of i-time and k-rate functions. It takes fsig, ksig and one optional i-time arguments and it outputs fsig data.

5 Conclusion

Csound is regarded as one of the most complete synthesis and processing languages in terms of its unit generator collection. The introduction of UDOs, plugin opcode and function table mechanisms, as well as a self-describing spectral signal framework, has opened the way for further expansion of the language. These methods provide simpler and quicker ways for customisation. In fact, one of the goals of csound 5 is to enhance the possibilities of extension and integration of the language/processing engine into other systems. It is therefore expected that the developments discussed in this article are but only the start of a new phase in the evolution of csound.

6 References

Richard Dobson. 2000. PVOCEX: File format for Phase Vocoder data, based on WAVE FORMAT EXTENSIBLE. . http://www.bath.ac.uk/~masrwd/pvocex/pvocex.html.

Rasmus Ekman. 2000. Csound Control Flow. http://www.csounds.com/internals/index.html.

John Ffitch. Extending Csound. In R. Boulanger, editor, *The Csound Book*, Cambridge, Mass., MIT Press.

Michael Goggins et Al. 2004. *The Csound API*. http://www.csounds.com/developers/html/csound_8h.html

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

Stephen T Pope. 1993. Machine Tongues XV: Three Packages for Software Sound Synthesis. *Computer Music Journal 17 (2).*

Mark Resibois. 2000. Adding New Unit Generators to Csound. In R. Boulanger, editor, *The Csound Book*, Cambridge, Mass., MIT Press.

Barry Vercoe. 2004. *The Csound and VSTCsound Reference Manual*, http://cvs.sourceforge.net/viewcvs.py/csound/csound5/csound.pdf.

Q: A Functional Programming Language for Multimedia Applications

Albert GRÄF
Department of Music-Informatics
Johannes Gutenberg University
55099 Mainz
Germany
ag@muwiinfa.geschichte.uni-mainz.de

Abstract

Q is a functional programming language based on term rewriting. Programs are collections of equations which are used to evaluate expressions in a symbolic fashion. Q comes with a set of extension modules which make it a viable tool for scientific programming, computer music, multimedia, and other advanced applications. In particular, Q provides special support for multimedia applications using PortAudio, libsndfile, libsamplerate, FFTW, MidiShare and OSC (including a SuperCollider interface). The paper gives a brief introduction to the Q language and its multimedia library, with a focus on the facilities for MIDI programming and the SuperCollider interface.

Keywords

Computer music, functional programming, multimedia programming, Q programming language, SuperCollider

1 Introduction

The pseudo acronym "Q" stands for "equational programming language". Q has its roots in term rewriting, a formal calculus for the symbolic evaluation of expressions coming from universal algebra and symbolic algebra systems (Dershowitz and Jouannaud, 1990). It builds on Michael O'Donnell's ground-breaking work on equational programming in the 1980s (O'Donnell, 1985) and the author's own research on efficient term pattern matching and rewriting techniques (Gräf, 1991).

In a sense, Q is for modern functional programming languages what BASIC is for imperative ones: It is a fairly simple language, thus easy to learn and use, yet powerful enough to tackle most common programming tasks; it is an interpreted (rather than compiled) language, offering adequate (though not C-like) execution speed; and it comes with a convenient interactive environment including a symbolic debugger, which lets you play with the parts of your program to explore different solution approaches and to test things out.

Despite its simplicity, Q should not be mistaken for a "toy language"; in fact, it comes with a fairly comprehensive collection of libraries which in many areas surpasses what is currently available for its bigger cousins like ML and Haskell. Moreover, Q's SWIG interface makes it easy to interface to additional C and C++ libraries if needed.

The Q programming environment is GPL'ed software which has been ported to a large variety of different platforms, including Linux (which has been the main development platform since 1993), FreeBSD, Mac OS X, BeOS, Solaris and Windows. Q also has a cross-platform multimedia library which currently comprises MIDI (via Grame's MidiShare), audio (providing interfaces to PortAudio v19, libsndfile, libsamplerate and FFTW) and software synthesis (via OSC, the "Open Sound Control" protocol developed by CNMAT, with special support for James McCartney's SuperCollider software). Additional modules for 3D graphics (OpenGL) and video (libxine) are currently under development.

In the following we give a brief overview of the language and the standard library, after which we focus on Q's multimedia facilities. More information about Q can be found on the Q homepage at `http://q-lang.sourceforge.net`.

2 The language

At its core, Q is a fairly simple language which is based entirely on the notions of *reductions* and *normal forms* pertaining to the term rewriting calculus. A Q program or *script* is simply a collection of equations which establish algebraic identities. The equations are interpreted as rewriting rules in order to reduce expressions to normal forms. The syntax of the language was inspired by the first edition of Bird and Wadler's influential book on functional programming (Bird and Wadler, 1988) and thus is similar to other modern functional languages such as Miranda and Haskell. For instance, here is how you define a function sqr which squares its argument by multiplying it with itself:

```
sqr X = X*X;
```

When this equation is applied to evaluate an expression like sqr 2, the interpreter performs the reduction sqr 2 => 2*2. It then goes on to apply other equations (as well as a number of built-in rules implementing the primitive operations such as arithmetic) until a normal form is reached (an expression is said to be in normal form if no more equations or built-in rules can be applied to it). In our example, the interpreter will invoke the rule which handles integer multiplication: 2*2 => 4. The resulting expression 4 is in normal form and denotes the "value" of the original expression sqr 2.

Note that, as in Prolog, capitalized identifiers are used to indicate the variables in an equation, which are bound to the actual values when an equation is applied. We also remark that function application is denoted simply by juxtaposition. Parentheses are used to group expressions and to indicate "tuple" values, but are *not* part of the function application syntax. This "curried" form of writing function applications is ubiquitous in modern functional languages. In addition, the Q language also supports the usual infix notation for operators such as + and *. As in other modern functional languages, these are just "syntactic sugar" for function applications; i.e., X*X is just a convenient shorthand for the function application (*) X X. Operator "sections" are also supported; e.g., (+1) denotes the function which adds 1 to its argument, (1/) the reciprocal function.

Equations may also include a condition part, as in the following (recursive) definition of the factorial function:

```
fact N = N*fact (N-1) if N>0;
       = 1 otherwise;
```

Another useful extension to standard term rewriting are the "where clauses" which allow you to bind local variables in an equation. For instance, the following equation defines a function for solving quadratic equations $x^2 + px + q = 0$. It first checks whether the discriminant $D = p^2/4 - q$ is nonnegative before it uses this value to compute the two real solutions of the equation.

```
solve P Q = (-P/2+sqrt D,-P/2-sqrt D)
  if D >= 0 where D = P^2/4-Q;
```

You can also define global variables using a def statement. This is useful if a value is used repeatedly in different equations and you don't want to recalculate it each time it is needed.

```
def PI = 4*atan 1;
```

Functions on structured arguments are defined by "pattern matching". E.g., the quicksort function can be implemented in Q with the following two equations. (Note that lists are written in Prolog-like syntax, thus [] denotes the empty list and [X|Xs] a list starting with the head element X and continuing with the list of remaining elements Xs. Furthermore, the ++ operator denotes list concatenation.)

```
qsort []     = [];
qsort [X|Xs] = qsort (filter (<X) Xs) ++
  [X] ++ qsort (filter (>=X) Xs);
```

Higher-order functions which take other functions as arguments can also be programmed in a straightforward way. For instance, the filter function used above is defined in the standard library as follows. In this case, the function argument P is a predicate expected to return the value true if an element should be included in the result list, false otherwise.

```
filter P []     = [];
filter P [X|Xs] = [X|filter P Xs] if P X;
                = filter P Xs otherwise;
```

In contrast to "pure" functional languages such as Haskell, Q takes the pragmatic route in that it also provides imperative programming features such as I/O operations and mutable data cells ("references"), similar to the corresponding facilities in the ML programming language. While one may argue about the use of such "impure" operations with side-effects in a functional programming language, they certainly make life easier when dealing, e.g., with complex I/O situations and thread synchronization. The || operator can be employed to execute such actions in sequence. For instance, using the built-in reads ("read string") and writes ("write string") functions, a simple prompt/input interaction would be written as follows:

```
prompt = writes "Input: " || reads;
```

References work like pointers to expressions. Three operations are provided: `ref` which creates a reference from its initial value, `put` which changes the referenced value, and `get` which returns the current value. With these facilities you can realize mutable data structures and maintain hidden state in a function. For instance, the following function `counter` returns the next integer at each invokation, starting at zero:

```
def COUNTER = ref 0;
counter = put COUNTER (N+1) || N
            where N = get COUNTER;
```

Despite its conceptual simplicity, Q is a full-featured functional programming language which allows you to write your programs in a concise and abstract mathematical style. Since it is an interpreted language, programs written in Q are definitely not as fast as their counterparts in C, but they are much easier to write, and the execution speed is certainly good enough for practical purposes (more or less comparable to interpreted Lisp and Haskell).

Just like other languages of its kind, Q has automatic memory management, facilities for raising and handling exceptions, constructs for defining new, application-specific data types, and means for partitioning larger scripts into separate modules. Functions and data structures using "lazy" evaluation can be dealt with in a direct manner. Q also uses dynamic typing, featuring a Smalltalk-like object-oriented type system with single inheritance. This has become a rare feature in contemporary functional languages which usually employ a static Hindley/Milner type system to provide more safety at the expense of restricting polymorphism. Q gives you back the flexibility of good old Lisp-style ad-hoc polymorphism and even allows you to extend the definition of existing operations (including built-in functions and operators) to your own data types.

3 The library

No modern programming or scripting language is complete without an extensive software library covering the more mundane programming tasks. In the bad old times of proprietary software, crafting such a library has always been a major undertaking, since all these components had to be created from scratch. Fortunately, nowadays there is a large variety of open source software providing more or less standardized solutions for all these areas, so that "reinventing the wheel" can mostly be avoided.

This is also the approach taken with the Q programming system, which acts as a kind of "nexus" connecting various open source technologies. To these ends, Q has an elaborate C/C++ interface including support for the SWIG wrapper generator (www.swig.org), which makes it easy to interface to existing C/C++ libraries. This enabled us to provide a fairly complete set of cross-platform extension modules which, while not as comprehensive as the facilities of other (much larger) language projects such as Perl and Python, make it possible to tackle most practical programming tasks with ease. This part of the Q library also goes well beyond what is offered with most other modern functional languages, especially in the multimedia department.

The core of the Q programming system includes a standard library, written mostly in Q itself, which implements a lot of useful Q types and functions, such as complex numbers, generic list processing functions (including list comprehensions), streams (a variant of lists featuring lazy evaluation which makes it possible to represent infinite data structures), container data structures (sets, dictionaries, hash tables, etc.), the lambda calculus, and a PostScript interface. Also included in the core is a POSIX system interface which provides, e.g., lowlevel I/O, process and thread management, sockets, filename globbing and regular expression matching.

In the GUI department, Q relies on Tcl/Tk (www.tcl.tk). While Tk is not the prettiest toolkit, its widgets are adequate for most purposes, it can be programmed quite easily, and, most importantly, it has been ported to a large variety of platforms. Using SWIG, it is also possible to embed GTK- and Qt-based interfaces, if a prettier appearance and/or more sophisticated GUI widgets are needed. (Complete bindings for these "deluxe" toolkits are on the TODO list, but have not been implemented yet.)

For basic 2D graphics, Q uses GGI, the "General Graphics Interface" (www.ggi-project.org), which has been augmented with a FreeType interface to add support for advanced font handling (www.freetype.org). Moreover, a module with bindings for the ImageMagick library (www.imagemagick.org) allows you to work with virtually all popular image file formats and provides an abundance of basic and advanced image manipulation functions.

To facilitate scientific programming, Q has interfaces to Octave, John W. Eaton's well-known MATLAB-like numerical computation software (www.octave.org), and to IBM's "Open Data Explorer", a comprehensive software for doing data visualization (www.opendx.org).

Web programming is another common occupation of the contemporary developer. In this realm, Q provides an Apache module and an XML/XSLT interface (xmlsoft.org) which allow you to create dynamic web content with ease. Moreover, an interface to the Curl library enables you to perform automated downloads and spidering tasks (curl.haxx.se). If you need database access, an ODBC module (www.iodbc.org, www.unixodbc.org) can be used to query and modify RDBMSs such as MySQL and PostgreSQL.

4 MIDI programming

Q's MIDI interface, embodied by the `midi` module, is based on Grame's MidiShare library (Fober et al., 1999). We have chosen MidiShare because it has been around since the time of the good old Atari and thus is quite mature, it has been ported to a number of different platforms (including Linux, Mac OS X and Windows), it takes a unique "client graph" approach which provides flexible dynamic routing of MIDI data between different applications, and, last but not least, it offers comprehensive support for handling standard MIDI files.

While MidiShare already abstracts from all messy hardware details, Q's `midi` module even goes one step further in that it also represents MIDI messages not as cryptic byte sequences, but as a high-level "algebraic" data type which can be manipulated easily. For instance, note on messages are denoted using data terms of the form `note_on CHANNEL NOTE VELOCITY`. The functions `midi_get` and `midi_send` are used to read and write MIDI messages, respectively. For example, Fig. 1 shows a little script for transposing MIDI messages in realtime.

The `midi` module provides all necessary data types and functions to process MIDI data in any desired way. It also gives access to MidiShare's functions to handle standard MIDI files. In order to work with entire MIDI sequences, MIDI messages can be stored in Q's built-in list data structure, where they can be manipulated using Q's extensive set of generic list operations. Q's POSIX multithreading support allows you to run multiple MIDI processing algorithms concurrently and with realtime scheduling priorities, which is useful or even essential for many types of MIDI applications.

These features make it possible to implement fairly sophisticated MIDI applications with moderate effort. To demonstrate this, we have employed the `midi` module to program various algorithmic composition tools and step sequencers, as well as a specialized graphical notation and sequencing software for percussion pieces. The latter program, called "clktrk", was used by the composer Benedict Mason for one of his recent projects (felt | ebb | thus | brink | here | array | telling, performed by the Ensemble Modern with the Junge Deutsche Philharmonie at the Donaueschingen Music Days 2004 and the Maerzmusik Berlin 2005).

Other generally useful tools with KDE/Qt-based GUIs can be found on the Q homepage. For instance, Fig. 2 shows the QMidiCC program, a MidiShare patchbay which can be configured to take care of your MidiShare drivers and to automatically connect new clients as soon as they show up in the MidiShare client list. QMidiCC can also be connected to other MidiShare applications to print their MIDI output and to send them MIDI start and stop messages.

5 Audio and software synthesis

The audio interface consists of three modules which together provide the necessary facilities for processing digital audio in Q. The `audio` module is based on PortAudio (v19), a cross-platform audio library which provides the necessary operations to work with the audio interfaces of the host operating system (www.portaudio.com). Under Linux this module gives access to both ALSA (www.alsa-project.org) and Jack (jackit.sf.net). The `sndfile` module uses Erik de Castro Lopo's libsndfile library which allows you to read and write sound files in a variety of formats (www.mega-nerd.com/libsndfile). The `wave` module provides basic operations to create, inspect and manipulate wave data represented as "byte strings" (a lowlevel data structure provided by Q's system interface which is used to store raw binary data). It also includes operations for sample rate conversion (via libsam-

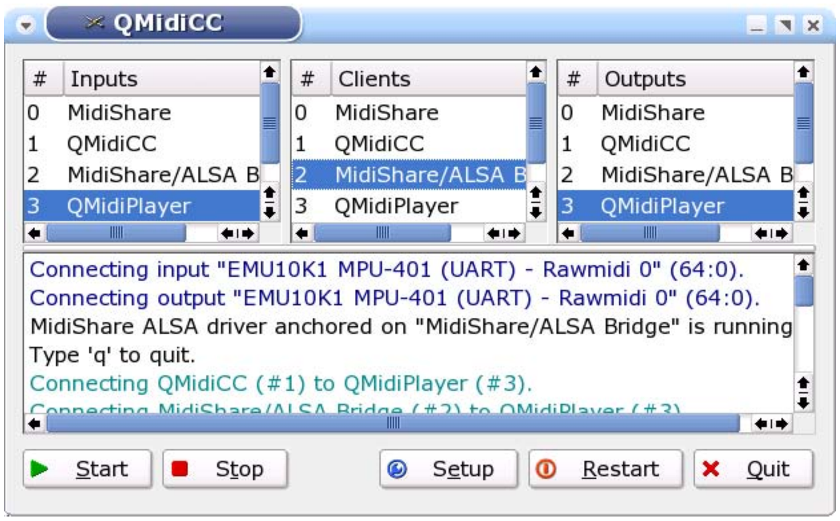
```
import midi;

/* register a MidiShare client and establish I/O connections */
def REF = midi_open "Transpose",
  IO = midi_client_ref "MidiShare/ALSA Bridge",
  _ = midi_connect IO REF || midi_connect REF IO;

/* transpose note on and off messages, leave other messages unchanged */
transp K (note_on CH N V)
                = note_on CH (N+K) V;
transp K (note_off CH N V)
                = note_off CH (N+K) V;
transp K MSG    = MSG otherwise;

/* the following loop repeatedly reads a message, transposes it and
   immediately outputs the transformed message */
transp_loop K   = midi_send REF 0 (transp K MSG) || transp_loop K
                    where (_,_,_,MSG) = midi_get REF;
```

Figure 1: Sample MIDI script.



#	Inputs	#	Clients	#	Outputs
0	MidiShare	0	MidiShare	0	MidiShare
1	QMidiCC	1	QMidiCC	1	QMidiCC
2	MidiShare/ALSA B	2	MidiShare/ALSA B	2	MidiShare/ALSA B
3	QMidiPlayer	3	QMidiPlayer	3	QMidiPlayer

```
Connecting input "EMU10K1 MPU-401 (UART) - Rawmidi 0" (64:0).
Connecting output "EMU10K1 MPU-401 (UART) - Rawmidi 0" (64:0).
MidiShare ALSA driver anchored on "MidiShare/ALSA Bridge" is running
Type 'q' to quit.
Connecting QMidiCC (#1) to QMidiPlayer (#3).
Connecting MidiShare/ALSA Bridge (#2) to QMidiPlayer (#3)
```

Figure 2: QMidiCC program.

plerate, www.mega-nerd.com/SRC) and fast Fourier transforms (via FFTW, www.fftw.org), as well as a function for drawing waveforms in a GGI visual.

Q's audio interface provides adequate support for simple audio applications such as audio playback and recording, and provides a framework for programming more advanced audio analysis and synthesis techniques. For these you'll either have to provide your own C or C++ modules to do the necessary processing of wave data, or employ Q's `osc` module which allows you to drive OSC-aware software synthesizers (www.cnmat.berkeley.edu/OpenSoundControl). We also offer an `sc` module which provides special support for James McCartney's SuperCollider (McCartney, 2002).

The `osc` module defines an algebraic data type as a high-level representation of OSC packets which can be manipulated easily. All standard OSC features are supported, including OSC bundles. The module also implements a simple UDP transport layer for sending and receiving OSC packets. In addition, the `sc` module offers some convenience functions to control SuperCollider's `sclang` and `scsynth` applications.

Fig. 3 shows a little Q script implementing some common OSC messages which can be used to control the SuperCollider sound server. Using these facilities in combination with the `midi` module, it is a relatively straightforward matter

```
import osc, sc;

// load a synthdef into the server
d_load NAME     = sc_send (osc_message CMD_D_LOAD NAME);

// create a new synth node (add at the end of the main group)
s_new NAME ID ARGS
                = sc_send (osc_message CMD_S_NEW (NAME,ID,1,0|ARGS));

// free a synth node
n_free ID       = sc_send (osc_message CMD_N_FREE ID);

// set control parameters
n_set ID ARGS   = sc_send (osc_message CMD_N_SET (ID|ARGS));
```

Figure 3: Sample OSC script.

```
/* get MIDI input */

midiin          = (TIME,MSG) where (_,_,TIME,MSG) = midi_get REF;

/* current pitch wheel value and tuning table */

def WHEEL =  ref 0.0, TT = map (ref.(*100.0)) [0..127];

/* calculate the frequency for a given MIDI note number N */

freq N          = 440*2^((get (TT!N)-6900)/1200+get WHEEL/6);

/* The MIDI loop: Assign voices from a queue Q of preallocated SC synth units
   in a round-robin fashion. Keep track of the currently assigned voices in a
   dictionary P. The third parameter is the MIDI event to be processed next. */

/* note offs: set the gate of the synth to 0 and put it at the end of the queue */

loop P Q (_,note_on _ N 0)
                = n_set I ("gate",0) || loop P Q midiin
                    where (I,_) = P!N, P = delete P N, Q = append Q I;
                = loop P Q midiin otherwise;

loop P Q (T,note_off CH N _)
                = loop P Q (T,note_on CH N 0);

/* note ons: turn note off if already sounding, then get a new voice from the
   queue and set its gate to 1 */

loop P Q (T,note_on CH N V)
                = n_set I ("gate",0) || loop P Q (T,note_on CH N V)
                    where (I,_) = P!N, P = delete P N, Q = append Q I;
                = n_set I ("freq",FREQ,"gain",V/127,"gate",1) ||
                  loop P Q midiin
                    where [I|Q] = Q, FREQ = freq N,
                      P = insert P (N,(I,FREQ));
```

Figure 4: Excerpt from a MIDI to OSC processing loop.

to implement software synthesizers which can be played in realtime via MIDI. All actual audio processing takes place in the synthesis engine, the Q script only acts as a kind of “MIDI to OSC” translator. For instance, Fig. 4 shows an excerpt from a typical MIDI processing loop.

An example of such a program, called “QSCSynth”, can be found on the Q homepage (cf. Fig. 5). QSCSynth is a (KDE/Qt based) GUI frontend for the `sclang` and `scsynth` programs which allows you to play and control SuperCollider synthesizers defined in an SCLang source file. It implements a monotimbral software synth which can be played via MIDI input and other MidiShare applications. Moreover, with MidiShare’s ALSA driver, QSCSynth can easily be wired up with ALSA-based sequencer applications like Rosegarden, employing it as a fully programmable realtime software synthesizer. The audio stream generated by SuperCollider can be watched in an integrated waveform/FFT display, and can also be recorded in an audio file. QSCSynth can also be configured to map arbitrary MIDI controller messages to corresponding OSC messages which change the control parameters of the synthesizer and effect units defined in the SCLang source file. Moreover, QSCSynth also provides its own control surface (constructed automatically from the parameter descriptions found in the binary synth definition files) which lets you control synth and effect units from the GUI as well.

6 The future

While Q’s multimedia library already provides a fairly complete framework for programming multimedia and computer music applications on Linux, there still remain a few things to be done:

- Finish the OpenGL and video support.
- Provide modules for some Linux-specific libraries such as Jack, LADSPA and DSSI.
- Provide high-level interfaces for computer music applications such as algorithmic composition. There are a few lessons to be learned from existing environments here, such as Rick Taube’s Common Music (Taube, 2005), Grame’s Elody (Letz et al., 2000) and Paul Hudak’s Haskore (Hudak, 2000b).
- Add graphical components for displaying and editing music (piano rolls, notation, etc.). For this we should try to reuse parts from existing open source software, such as Lilypond (lilypond.org), the GUIDO library (www.salieri.org/guido) and Rosegarden (www.rosegardenmusic.com).
- Add a “patcher”-like visual programming interface, such as the one found in IRCAM’s OpenMusic.

7 Conclusion

Functional programming has always played an important role in computer music, because it eases the symbolic manipulation of complex structured data. However, to our knowledge no other “modern-style” functional language currently provides the necessary interfaces to implement sophisticated, realtime-capable multimedia applications. We therefore believe that Q is an interesting tool for those who would like to explore MIDI programming, sound synthesis and other multimedia applications, in the context of a high-level, general-purpose, non-imperative programming language.

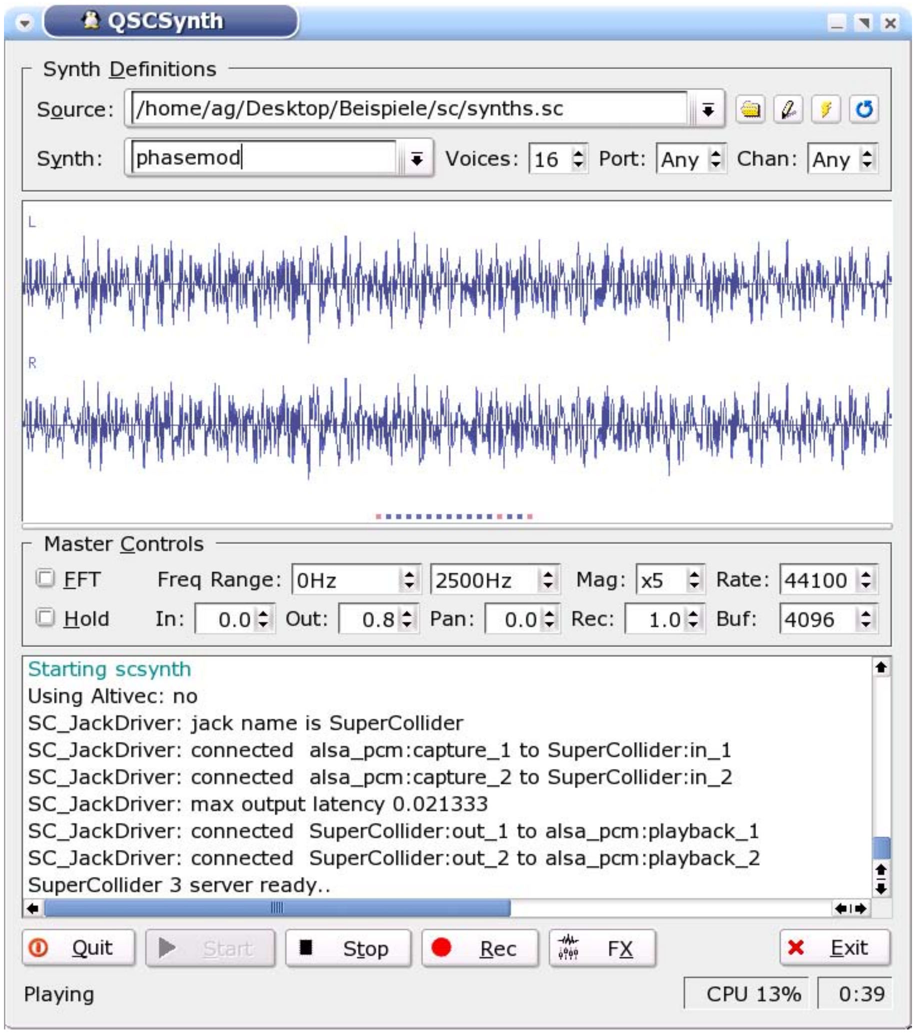
While the Q core system is considered stable, the language and its libraries continue to evolve, and it is our goal to turn Q into a viable tool for rapid application development in many different areas. We think that multimedia is an attractive playground for functional programming, because modern FP languages allow many problems in this realm to be solved in new and interesting ways; see in particular Paul Hudak’s book on multimedia programming with Haskell (Hudak, 2000a) for more examples. As the multithreading and realtime capabilities of mainstream functional languages mature, it might also be an interesting option to port some of Q’s libraries to other environments such as the Glasgow Haskell compiler which offer better execution speed than an interpreted language, for the benefit of both the functional programming community and multimedia application developers.

References

Richard Bird and Philip Wadler. 1988. *Introduction to Functional Programming.* Prentice Hall, New York.

Nachum Dershowitz and Jean-Pierre Jouannaud. 1990. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, chapter 6, pages 243–320. Elsevier.

Dominique Fober, Stephane Letz, and Yann Orlarey. 1999. MidiShare joins the open sources



Source: /home/ag/Desktop/Beispiele/sc/synths.sc
Synth: phasemod Voices: 16 Port: Any Chan: Any
FFT Freq Range: 0Hz 2500Hz Mag: x5 Rate: 44100
Hold In: 0.0 Out: 0.8 Pan: 0.0 Rec: 1.0 Buf: 4096

```
Starting scsynth
Using Altivec: no
SC_JackDriver: jack name is SuperCollider
SC_JackDriver: connected  alsa_pcm:capture_1 to SuperCollider:in_1
SC_JackDriver: connected  alsa_pcm:capture_2 to SuperCollider:in_2
SC_JackDriver: max output latency 0.021333
SC_JackDriver: connected  SuperCollider:out_1 to alsa_pcm:playback_1
SC_JackDriver: connected  SuperCollider:out_2 to alsa_pcm:playback_2
SuperCollider 3 server ready..
```

Playing CPU 13% 0:39

Figure 5: QSCSynth program.

softwares. In *Proceedings of the International Computer Music Conference*, pages 311–313, International Computer Music Association. See also http://www.grame.fr/MidiShare.

Albert Gräf. 1991. Left-to-right tree pattern matching. In Ronald V. Book, editor, *Rewriting Techniques and Applications*, LNCS 488, pages 323–334. Springer.

Paul Hudak. 2000a. *The Haskell School of Expression: Learning Functional Programming Through Multimedia*. Cambridge University Press.

Paul Hudak. 2000b. Haskore Music Tutorial. Yale University, Department of Computer Science. See http://www.haskell.org/haskore.

Stephane Letz, Dominique Fober, and Yann Orlarey. 2000. Realtime composition in Elody. In *Proceedings of the International Computer Music Conference*, International Computer Music Association. See also http://www.grame.fr/Elody.

James McCartney. 2002. Rethinking the computer music language: SuperCollider. *Computer Music Journal*, 26(4):61–68. See also http://supercollider.sourceforge.net.

Michael O'Donnell. 1985. *Equational Logic as a Programming Language*. Series in the Foundations of Computing. MIT Press, Cambridge, Mass.

Heinrich K. Taube. 2005. *Notes from the Metalevel: Introduction to Algorithmic Music Composition*. Swets & Zeitlinger. To appear. http://pinhead.music.uiuc.edu/~hkt/nm.

jackdmp: Jack server for multi-processor machines

S.Letz, D.Fober, Y.Orlarey
Grame - Centre national de création musicale
{letz, fober, orlarey}@grame.fr

Abstract

jackdmp is a C++ version of the Jack low-latency audio server for multi-processor machines. It is a new implementation of the jack server core features that aims in removing some limitations of the current design. The activation system has been changed for a data flow model and lock-free programming techniques for graph access have been used to have a more dynamic and robust system. We present the new design and the implementation for MacOSX.

Keywords

real-time, data-flow model, audio server, lock-free

1 Introduction

Jack is a low-latency audio server, written for POSIX conformant operating systems such as GNU/Linux. It can connect a number of different applications to an audio device, as well as allowing them to share audio between themselves (Vehmanen, Wingo and Davis 2003). The current code base written in C, developed over several years, is available for GNU/Linux and MacOSX systems. An additional integration with the MacOSX CoreAudio architecture has been realized (Letz, Fober and Orlarey 2004).

The system is now a fundamental part of the Linux audio world, where most of music-oriented audio applications are now Jack compatible. On MacOSX, it has extended the CoreAudio architecture by adding low-latency inter-application audio routing capabilities in a transparent manner. [1]

The new design and implementation aims in removing some limitations of the current version, by isolating the "heart" of the system and simplifying the implementation:

- the sequential activation model has been changed to a new graph activation scheme based on a data-flow model, that will naturally take profit of multi-processor machines

- a more robust architecture based on *lock-free* programming techniques has been developed to allow the server to keep working (not interrupting the audio stream) when the client graph changes or in case of client execution failure, especially interesting in live situations.

- various simplifications have been done in the internal design.

The section 2 explains the requirements, section 3 describes the new design, section 4 describes the implementation, and finally section 5 describes the performances.

2 Multi-processing

Taking profit of multi-processor architectures usually requires applications to be adapted. A natural way is to develop multi-threaded code, and in audio applications a usual separation consists in executing audio DSP code in a real-time thread and normal code (GUI for instance) in one or several standard threads. The scheduler then activates all runnable threads in parallel on available processors.

In a Jack server like system, there is a natural source of parallelism when Jack clients depend of the same input and can be executed on different processor at the same time. The main requirement is then to have an activation model that allows the scheduler to correctly activate parallel runnable clients. Going from a sequential activation model to a completely distributed one also raise synchronization issues that can be solved using *lock-free* programming techniques.

[1]All CoreAudio applications can take profit of Jack features without any modification

3 New design

3.1 Graph execution

In the current activation model (either on Linux or MacOSX), knowing the data dependencies between clients allows to sort the client graph to find an activation order. This topological sorting step is done each time the graph state changes, for example when connections are done or removed or when a new client opens or closes. This order is used by the server to activate clients in sequence.

Forcing a complete serialization of client activation is not always necessary: for example clients A and B (Fig 1) could be executed at the same time since they both only depend of the "Input" client. In this graph example, the current activation strategy choose an arbitrary order to activate A and B. This model is adapted to mono-processor machines, but cannot exploit multi-processor architectures efficiently.

3.2 Data flow model

Data flow diagrams (DFD) are an abstract general representation of how data flows around a system. In particular they describe systems where the ordering of operations is governed by *data dependencies* and by the fact that only the availability of the needed data determines the execution of one of the process.

A graph of Jack clients typically contains *sequencial* and *parallel* sub-parts (Fig 1). When parallel sub-graph exist, clients can be executed on different processors at the same time. A data-flow model can be used to describe this kind of system: a node in a data-flow graph becomes *runnable* when all inputs are available. The client ordering step done in the mono-processor model is not necessary anymore. Each client uses an *activation counter* to count the number of input clients which it depends on. The state of client connections is updated each time a connection between ports is done or removed.

Activation will be transfered from client to client during each server cycle as they are executed: a suspended client will be resumed, executes itself, propagates activation to the output clients, go back to sleep, until all clients have been activated. [2]

[2]The data-flow model still works on mono-processor machines and will correctly guaranty a minimum global number of context switches like the "sequential" model.

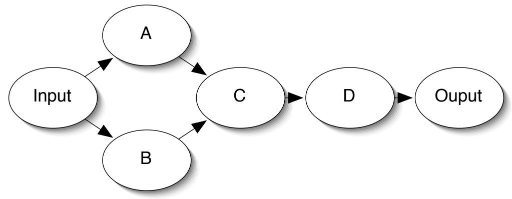


Figure 1: *Client graph: Client A and B could be executed at the same time, C must wait for A and B end, D must wait for C end.*

3.2.1 Graph loops

The Jack connection model allows loops to be established. Special *feedback connections* are used to close a loop, and introduce a one buffer latency. We currently follow Simon Jenkins [3] proposition where the feedback connection is introduced at the place where the loop is established. This scheme is simple but has the drawback of having the activation order become sensitive to the connection history. More complex strategies that avoid this problem will possibly be tested in the future.

3.3 Lock-free programming

In classic lock-based programming, access to shared data needs to be serialized using mutual exclusion. Update operations must appear as *atomic*. The standard way is to use a mutex that is locked when a thread starts an update operation and unlocked when the operation is finished. Other threads wanting to access the same data check the mutex and possibly suspend their execution until the mutex becomes unlocked. Lock based programming is sensitive to priority inversion problems or deadlocks. Lock-free programming on the contrary allows to build data structures that are safe for concurrent use without needing to manage locks or block threads (Fober, Letz, and Orlarey 2002).

Locks are used at several places in the current Jack server implementation. For example, the client graph needs to be locked each time a server update operation access it. When the real-time audio thread runs, it also needs to access the client graph. If the graph is already locked and to avoid waiting an arbitrary long time, the Real-Time (RT) thread generates an empty buffer for the given audio cycle, causing an annoying interruption in the audio stream.

A lock-free implementation aims at remov-

[3]Discussed on the jack-dev mailing list

ing all locks (and particularly the graph lock) and allowing all graph state changes (add/remove client, add/remove ports, connection/disconnection...) to be done *without interrupting the audio stream.* [4] As described in the implementation section, this new constraint requires also some changes in the client side threading model.

3.3.1 Lock-free graph state change

All update operations from clients are serialized through the server, thus only one thread updates the graph state. RT threads from the server and clients have to see the *same coherent* state during a given audio cycle. Non RT threads from clients may also access the graph state at any time. The idea is to use two states: one *current* state and one *next* state to be updated. A state change consists in *atomically* switching from the current state to the next state. This is done by the RT audio server thread at the beginning of a cycle, and other clients RT threads will use the same state during the entire cycle. All state management operations are implemented using the CAS [5] operation and are described with more details in the implementation section.

3.4 A "robust" server

Having a robust system is especially important in live situations where one can accept a temporary graph execution fault, which is usually better that having the system totally failing with a completely silent buffer and an audio stream interruption for example. In the current sequential version, the server waits for the client graph execution end before in can produce the output audio buffers. Thus a client that does not run during one cycle will cause the complete failure of the system.

In a multi-processor context, it is interesting to have a more *distributed* system, where a part of the graph may still run on one processor even if another part is blocked on the other one.

3.4.1 Engine cycle

The engine cycle has been redesigned. The server no longer waits for the client execution end. It uses the buffers computed at the previous cycle. The server cycle is fast and take almost constant time since it is totally decoupled from the clients execution. This allows the system to keep running even if a part of the graph can not be executed during the cycle for whatever reason (too slow client, crash of a client...).

The server is more robust: the resulting output buffer may be incomplete, if one or several clients have not produced their contribution, but the output audio stream will still be produced. The server can detect abnormal situations by checking if all clients have been executed during the previous cycle and possibly notify the faulty clients with an *XRun* event.

3.4.2 Latency

Since the server uses the output buffers produced during the previous cycle, this new model adds a *one buffer more latency in the system.*[6] But according to the needs, it will be possible to choose between the current model where the server is synchronized on the client graph execution end and the new more robust distributed model with higher latency.

4 Implementation

The new implementation concentrates on the core part of the system. Some part of the API like the *Transport* system are not implemented yet.

4.1 Data structure

Accessing data in shared memory using pointers on the server and client side is usually complex: pointers have to be described as offset related to a base address local to each process. Linked lists for example are more complex to manage and usually need locked access method in multi-thread cases. We choose to simplify data structures to use fixed size preallocated arrays that will be easier to manipulate in a lock free manner.

4.2 Shared Memory

Shared memory segments are allocated on the server side. A reference (index) on the shared segment must be transfered on the client side. Shared memory management is done using two classes:

- On the server side, the **JackShmMem** class overloads **new** and **delete** operators. Objects of sub-classes of JackShmMem will

[4] Some operations like buffer size change will still interrupt the audio stream.

[5] CAS is the basic operation used in lock-free programming: it compares the content of a memory address with an expected value and if success, replaces the content with a new value.

[6] At least on OSX where the driver internal behaviour concerning input and output latencies values cannot be precisely controlled

be automatically allocated in shared memory. The **GetShmIndex** method retrieves the corresponding index to be transfered and used on the client side.

- Shared memory objects are accessed using a standard pointer on the server side. On the client side, the **JackShmPtr** template class allows to manipulate objects allocated in shared memory in a transparent manner: initialized with the index obtained from the server side, a JackShmPtr pointer can be used to access data and methods [7] of the corresponding server shared memory object.

Shared memory segments allocated on the server will be transfered from server to client when a new client is registered in the server, using the corresponding shared memory indexes.

4.3 Graph state

Connection state was previously described as a list of connected ports for a given port. This list was duplicated both on the server and client side thus complicating connection/disconnection steps. Connections are now managed in shared memory in fixed size arrays.

The **JackConnectionManager** class maintains the state of connections. Connections are represented as an array of port indexes for a given port. Changes in the connection state will be reflected the next audio cycle.

The **JackGraphManager** is the global graph management object. It contains a connection manager and an array of preallocated ports.

4.4 Port description

Ports are a description of data type to be exchanged between Jack clients, with an associated buffer used to transfer data. For audio input ports, this buffer is typically used to mix buffers from all connected output ports. Audio buffers were previously managed in a independent shared memory segment.

For simplification purpose, each audio buffer is now associated with a port. Having all buffers in shared memory will allow some optimizations: an input port used at several places with the same data dependencies could possibly be *computed once and shared*. Buffers are preallocated with the maximum possible size, there is no re-allocation operation needed anymore. Ports are implemented in the **JackPort** class.

[7] Only non virtual methods

4.5 Client activation

At each cycle, clients that only depend of the input driver and clients without inputs have to be activated first. To manage clients without inputs, an internal *freewheel* driver is used: when first activated, the client will be connected to it. At the beginning of the cyle, each client has its activation counter containing the number of input client it depends on. After being activated, the client decrements the activation counter of all its connected output. The *last* activated input client will resume the following client in the graph. (Fig 2)

Each client uses an inter-process *suspend/resume* primitive associated with an *activation counter*. An implementation could be described with the following pseudo code. Execution of a server cycle follows several steps:

- read audio input buffers
- write output audio buffers computed the previous cycle
- for each client in client list, reset the activation counter to its initial value
- activate all clients that depends on the input driver client or without input
- suspend until next cycle

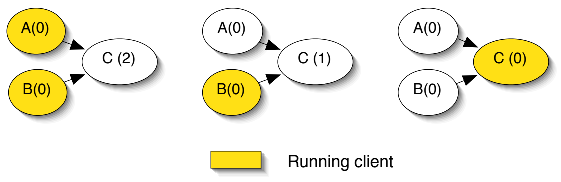


Figure 2: *Example of graph activation: C is activated by the last running of its A and B input.*

After being resumed by the system, execution of a client consists of:

- call the client process callback
- propagate activation to output clients
- suspend until the next cycle

On each platform, an efficient synchronization primitive is needed to implement the suspend/resume operation. Mach semaphores are used on MacOSX. They are allocated and published by the server in a global namespace (using the *mach bootstrap service* mechanism). Running clients are notified when a new client is opened and access the corresponding semaphore.

Linux kernel 2.6 features the Fast User space mutEx (futex), a new facility that allows two process to synchronize (including blocking and waking) with either no or very little interaction with the kernel. It seems likely that they are better suited to the task of coordinating multiple processes than the FIFO's that the Linux implementation currently uses.

4.6 Lock-free graph access

Lock-free graph access is done using the **JackAtomicState** template class. This class implement the two state pattern. Update methods use on the *next state* and read methods access the *current state*. The two states can be atomically exchanged using a CAS based implementation.

- code updating the next state is *protected* using the **WriteNextStateStart** and **WriteNextStateStop** methods. When executed between these two methods, it can freely update the next state and be sure that the RT reader thread can not switch to the next state.[8]
- the RT server thread switch to the new state using the **TrySwitchState** method that returns the current state if called concurrently with a update operation and switch to the next state otherwise.
- other RT threads read the current state, valid during the given audio cycle using the **ReadCurrentState** method.
- non RT threads read the current state using the **ReadCurrentState** method and have to check that the state was not changed during the read operation (using the **GetCurrentIndex** method):

```
void ClientNonRTCode(...)
{
    int cur_index,next_index;
    State* current_state;
    next_index = GetCurrentIndex();
    do {
       cur_index = next_index;
       current_state = ReadCurrentState();
       ...
       < copy current_state >
       ...
       next_index = GetCurrentIndex();
    } while (cur_index != next_index);
}
```

[8]The programming model is similar to a lock-based model where the update code would be written inside a *mutex-lock/mutex-unlock* pair.

4.7 Server client communications

A global *client registration* entry point is defined to allow client code to register a new client (a **JackServerChannel** object). A private communication channel is then allocated for each client for all *client requests*, and remains until the client quits. Possible crash of a client is detected and handled by the server when the private communication channel is abnormally closed. A notification channel is also allocated to allow the server to notify clients: graph reorder, xrun, port registration events...

Running clients can also detect when the server no more runs as soon as waiting on the input suspend/resume primitive fails. (Fig 3)

The current version uses socked based channels. On MacOSX, we use MIG (Mach Interface Generator), a very convenient way to define new Remote Procedure Calls (RPC) between the server and clients. [9]

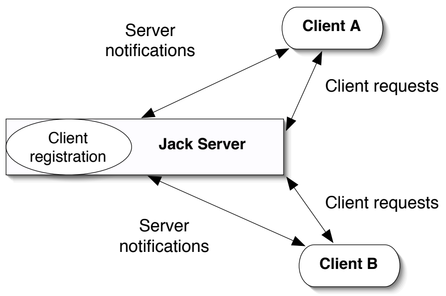


Figure 3: *The server defines a public "client registration" channel. Each client is linked with the server using two "request "and "notification" channels.*

4.8 Server

The Jack server contains the global client registration channel, the drivers, an engine, and a graph manager. It receives requests from the global channel, handle some of them (BufferSize change, Freewheel mode..) and redirect other ones on the engine.

4.8.1 Engine

The engine contains a **JackEngineControl**, a global shared server object also visible for clients. It does the following:

[9]Both synchronous and asynchronous function calls can be defined

- handles requests for new clients through the global client registration channel and allocates a server representation of new external clients
- handles request from running clients
- activates the graph when triggered by the driver and does various timing related operations (CPU load measurement, detection of late clients...)

4.8.2 Server clients

Server clients are either internal clients (a **JackInternalClient** object) when they run in the server process space[10] or external clients (a **JackExternalClient** object) as a server representation of an external client. External clients contain the local data (for example the notification channel, a **JackNotifyChannel** object) and a **JackClientControl** object to be used by the server and the client.

4.8.3 Library Client

On the client side, the current Jack version uses a one thread model: real-time code and notifications (graph reorder event, xrun event...) are treated in a unique thread. Indeed the server stops audio processing while notifications are handled on the client side. This has some advantages: a much simpler model for synchronization, but also some problematic consequences: since notifications are handled in a thread with real-time behaviour, a non real-time safe notification may disturb the whole machine.

Because the server audio thread is not interrupted anymore, most of server notifications will typically be delivered while the client audio thread is also running. A two threads model for client has to be used:

- a real-time thread dedicated to the audio process
- a standard thread for notifications

The client notification thread is started in **jack-client-new** call. Thus clients can already receive notifications when they are in the *opened* state. The client real-time thread is started in **jack-activate** call. A connection manager client for example does not need to be activated to be able to receive *graphreorder*, or *portregistration* like notifications (Fig 4).

[10]Drivers are a special subclass of internal clients

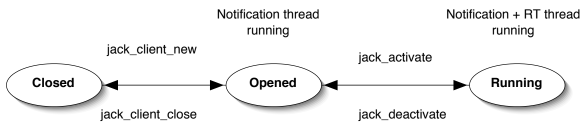


Figure 4: *Client life cycle*

This two threads model will possibly have some consequences for existing Jack applications: they may have to be adapted to allow a notification to be called while the audio thread is running.

The library client (a **JackLibClient** object) redirects the external Jack API to the Jack server. It contains a **JackClientChannel** object that implements both the request and notification channels, local client side resources as well as access to objects shared with the server like the graph manager or the server global state.

4.8.4 Drivers

Drivers are needed to activate the client graph. Graph state changes (new connections, port, client...) are done by the server RT thread. When several drivers need to be used, one of them is called the *master* and updates the graph. Other one are considered as *slaves*.

The **JackDriver** class implements common behaviour for drivers. Those that use a blocking audio interface (like the **JackALSADriver** driver) are subclasses of the **JackThreadedDriver** class. A special **JackFreewheelDriver** (subclass of JackThreadedDriver) is used to activate clients without inputs and to implement the freewheel mode (see 4.8.5). The **JackAudioDriver** class implements common code for audio drivers, like the management of audio ports. Callback based drivers (like the **JackCoreAudioDriver** driver, a subclass of JackAudioDriver) can directly trigger the Jack engine.

When the graph is synchronized to the audio card, the audio driver is the master and the freewheel driver is a slave.

4.8.5 Freewheel mode

In freewheel mode, Jack no longer waits for any external event to begin the start of the next process cycle thus allowing faster than real-time execution of Jack graph. Freewheel mode is implemented by switching from the audio and freewheel driver synchronization mode to the freewheel driver only:

- the global connection state is saved
- all audio driver ports are deconnected, thus there is no more dependancies with the audio driver
- the freewheel driver is synchronized with the end of graph execution: all clients are connected to the freewheel driver
- the freewheel driver becomes the master

Normal mode is restored with the connections state valid before freewheel mode was done. Thus one consider that no graph state change can be done during freewheel mode.

4.9 XRun detection

Two kind of XRun can be detected:

- XRun reported by the driver
- XRun detected by the server when a client has not be executed the previous cycle: this typically correspond to abnormal scheduler latencies

On MacOSX, the CoreAudio HAL system already contains a XRun detection mechanism: a *kAudioDeviceProcessorOverload* notification is triggered when the HAL detects an XRun. The notification will be redirected to all running clients. All clients that have not been executed the previous cycle will be notified individually.

5 Performances

The multi-processor version has been tested on MacOSX. Preliminary benchmarks have been done on a mono and dual 1.8 Ghz G5 machine. Five *jack-metro* clients generating a simple bip are running.

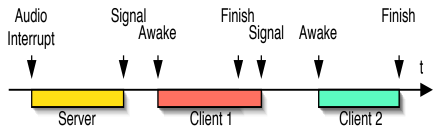


Figure 5: *Timing diagram for a two clients in sequence example*

For a server cycle, the *signal date* (when the client resume semaphore is activated), the *awake date* (when the client actually wakes up) and the *finish date* (when the client ends its processing and go back to suspended state) relative to the server cycle start date *before reading and writing audio buffers* have been measured. The first slice in the graph also reflects the server behavior: the duration to read and write the audio buffers can be seen as the *signal* date curve offset on the Y-coordinate. After having signaled the first client, the server returns to the CoreAudio HAL (Hardware Abstract Layer), which mix the output buffers in the kernel driver (offset between the first client *signal* date and its *awake* date (Fig 5)). The first client is then resumed.

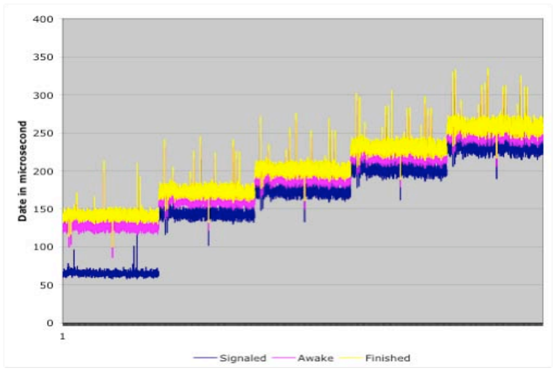


Figure 6: *Mono G5, clients connected in sequence. For a server cycle: signal (blue), awake (pink) and finish (yellow) date. End date is about 250 microsecond on average.*

With all clients running at the same time, the measure is done during 5 seconds. The behavior of each client is then represented as a 5 seconds "slice" in the graph and all slices have been concatenated on the X axis, thus allowing to have a global view of the system.

Two benchmarks have been done. In the first one, clients are connected in sequence (client 1 is connected to client 2, client 2 to client 3 and so on), thus computations are inevitably serialized. One can clearly see that the *signal* date of client 2 happens after the *finished* date of client 1 and the same behavior happens for other clients. Measures have been done on the mono (Fig 6) and dual machine (Fig 7).

In the second benchmark, all clients are only connected to the input driver, thus they can possibly be executed in parallel. The input driver client signal all clients at (almost) the same date [11]. Measures have been done on the mono (Fig 8) and dual (Fig 9) machine. When parallel clients are executed on the dual

[11] Signaling a semaphore has a cost that appears as the slope of the signal curve.

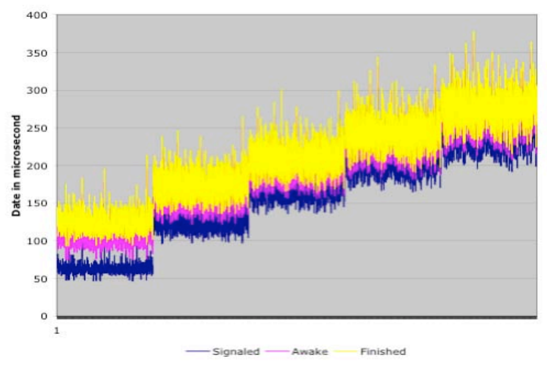


Figure 7: *Dual G5. Since clients are connected in sequence, computations are also serialized, but client 1 can start earlier on the second processor. End date is about 250 microsecond on average.*

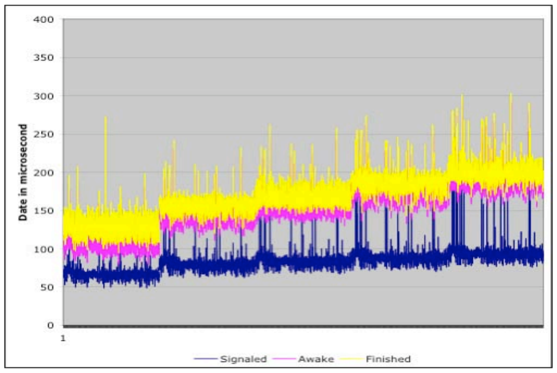


Figure 9: *Parallel clients on a dual G5. Client 1 can start earlier on the second processor before all clients have been signalled. Computations are done in parallel. End date is about 200 microsecond on average.*

machine, one see clearly that computations are done at the same time on the 2 processors and the end date is thus lowered.

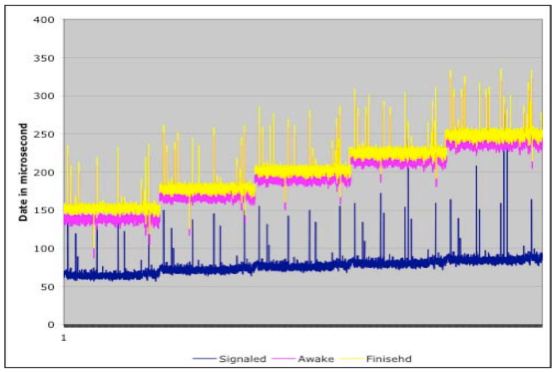


Figure 8: *Parallel clients on a mono G5. Although the graph can potentially be parallelized, computations are still serialized. End date is about 250 microsecond on average.*

Other benchmarks with different parallel/sequence graph to check their correct activation behavior and comparaison with the same graphs runned on the mono-processor machine have been done. A worst case additional latency of 150 to 200 microseconds added to the average finished date of the last client has been measured.

6 Conclusion

With the development of multi-processor machines, adapted architectures have to be developed. The Jack model is particularly suited to this requirement: instead of using a "monolithic" general purpose heavy application, users can build their setup by having several smaller and goal focused applications that collaborate, dynamically connecting them to meet their specific needs.

By adopting a data flow model for client activation, it is possible to let the scheduler naturally distribute parallel Jack clients on available processors, and this model works for the benefit of all kind of client aggregation, like *internal clients in the Jack server*, or *multiple Jack clients in an external process.*

A Linux version has to be completed with an adapted primitive for inter process synchronization as well as socket based communication channels between the server and clients. The multi-processor version is a first step towards a completely distributed version, that will take advantage of multi-processor on a machine and could run on multiple machines in the future.

References

D.Fober, S.Letz, Y.Orlarey "Lock-Free Techniques for Concurrent Access to Shared Objects", *Actes des Journes d'Informatique Musicale JIM2002, Marseille, pages 143–150*

S.Letz, D.Fober, Y.Orlarey, P.Davis "Jack Audio Server: MacOSX port and multi-processor version, *Proceedings of the first Sound and Music Computing conference - SMC'04", pages 177–183*

Vehmanen Kai, Wingo Andy and Davis Paul "Jack Design Documentation", *http://jackit.sourceforge.net/docs/design/*

On The Design of Csound5

John ffitch
Department of Computer Science
University of Bath
Bath BA2 7AY,
UK,
jpff@cs.bath.ac.uk

Abstract

Csound has been in existence for many years, and is a direct descendant of the MusicV family. For a decade development of the system has continued, via some language changes, new operations and the necessary bug fixes. Two years ago a small group of us decided that rather than continue the incremental process, a code freeze and rethink was needed. In this paper we consider the design and aims for what has been called Csound5, and describe the processes and achievements of the implementation.

Keywords

Synthesis language, Csound.

1 Introduction and Background

The music synthesis language Csound (Boulanger, 2000) was produced by Barry Vercoe(Vercoe, 1993) and was available under the MIT Licence on a small number of platforms. The current author ported the code to the Windows environment in the early 1990s, whereupon a self-defining team of programmers, DSP experts and musicians emerged who have continued to maintain and extend the software package ever since. The original synthesis engine has remained largely unchanged, while a significant number of new operations (opcodes) and table creation routines have been added. Despite various suggestions over the years, the two languages — the score language and the orchestra language — have remained unaltered until very recently, when user-defined opcodes, `if..else` and score looping constructs were introduced.

The user base of Csound is large, and as we have maintained a free download policy we do not know how many copies there are in existence or how many are being used. What is clear from the Csound mailing lists is that the community is very varied, and while some of us think of ourselves as classical "art" composers, there are also live performers, techno and ambient composers, and many other classifications.

The subject of this paper is Csound5, and in particular how its design has evolved from the current Csound. But there are two particular phenomena that have had a direct influence on the need for the re-think.

The first was legal; Csound had been distributed under the MIT licence since 1986, which stipulates some freedoms and some restrictions. The freedoms are expressed as *Permission to use, copy, or modify these programs and their documentation for educational and research purposes only and without fee is hereby granted, provided that this copyright and permission notice appear on all copies and supporting documentation.* There was clarification that this should be taken to allow composers to use it without imposing any restriction on the resulting music. However the licence continues *For any other uses of this software, in original or modified form, including but not limited to distribution in whole or in part, specific prior permission from M.I.T. must be obtained.* When Csound was first made available this was considered a free licence, but with the growth of the Free Software movement, and much wider availability of computers, the restriction stopped developers making use of Csound in larger software systems if they were intending to distribute the resulting system. It also acted to prevent some kinds of publicity, as might be engendered by inclusion in books and magazines. Early attempts to resolve these problems failed, mainly though incomprehension. The publication of Phillips' book(Phillips, 2000) was a further call to address the problem. The change which influenced the whole approach to the development of Csound was the adoption by MIT of the Lesser GNU Public Licence. The *de facto* monopoly allowing distribution was gone.

The second phenomenon was the apparently remorseless improvements in technology. Csound was conceived as an off-line program, rendering a sound description over however long

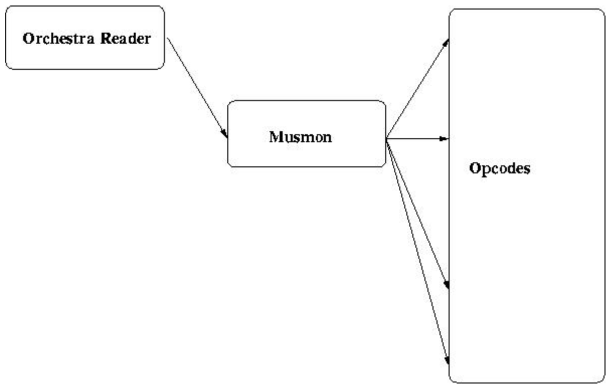


Figure 1: Architecture of original Csound

it took. In the mid 1990s there was a project to recreate Csound for an embedded DSP processor(Vercoe, 1996) as a means of making a real-time synthesis system. This has been overtaken by the increase in machine speeds, and this speed has resulted in the Csound community calling for real-time performance, performer interfaces and MIDI controls. While some users had been wanting this for years, the availability of processors that were nearly capable of real-time rendering made all too clear the shortcomings of the 15- year-old design.

At the end of 2002 we imposed a code freeze to allow the developer community to catch up with their modifications, and in particular to allow larger scale changes to be made on a fixed target. The previous version was still subjected to bug fixes but mainstream development ceased as we moved to Sourceforge and opened up the system even further.

This paper gives one person's view of the system we are building, usually called Csound5, as we froze at version 4.23. As the system is now running largely satisfactorily it is a good time to reflect on the aims of this major reconstruction, and to what extent our aspirations have been matched by our achievements.

2 Requirements

The developers had a number of (distributed) discussions of what was needed in any revision. The strongest requirement was the ability to embed Csound within other systems, be they performance system or experimental research testbeds(ffitch and Padget, 2002). This has a number of software implications. The most significant one is perhaps the need for an agreed application process interface (API) which would allow the controlling program access to some of the internal operations of Csound, and also separate the compilation processes from the execution. Also in the scope of the API is the possibility of adding new opcodes and generators which have access to the opcode mechanisms, memory allocation, navigation of internal structures and so on.

Related to the requirement for a documented software interface is a call to make Csound re-entrant. This would allow multiple uses both serially and in parallel. The original code was written with no thought for such niceties, and there is a plethora of static variables throughout the system. Removing these would be a major step towards re-entrance, and encapsulating the state within a single structure was the proposed solution, a structure that could also carry parts of the API.

A possible lesser goal was to improve the internal reporting of errors. The original system set a global variable to indicate an initialisation error or a performance error, and this is checked at the top event loop. A simpler and more respectable process is for each initialiser and operator to return an error code; such a system can be extended to make use of the error codes.

Csound originally generated IRCAM format sound files, and AIFF. Later WAV was added and some variants of AIFC. The code was all *ad hoc* and as audio formats are continually being developed, it seemed an ideal opportunity to capitalise on the work of others, and to use an external library to provide audio filing.

In a similar way the real-time audio output is specially written for each platform, and maintaining this reduces the time available for development and enhancement. Since Csound was written, cross-platform libraries to encapsulate real-time audio have been developed, and while using an external library for files it seemed natural to investigate the same for sound.

Another aspect where there was platform-dependent code is in graphics. Csound has been able to display waveforms and spectral frames from the beginning, but there are a large number of optional files for DOS, Windows, Macintosh, SGI, SUN, X, and so forth. Using a general graphical system would move this complication into someone else's hands. It would also be useful if the graphical activity were made external, using the API, so a variety of graphical packages could be used in a fashion like embedding. This leads to the idea of providing a visible software bus to communicate between the Csound engine and the wider environment.

The last component where an external library could assist is in MIDI. There have been complaints about the support for MIDI for a long time, and so in any reconstruction it was clearly something that should be addressed.

The last major component that is in need of reconstruction is the orchestra parser. The original parser is an *ad hoc* parser very reminiscent of the late 1970s. It is hard to modify and there are bugs lurking there that have evaded all attempts to fix. If a new parser were to be written it could sidestep these problems and also allow things like two-argument functions, which have been requested in the past. Another possible outcome from a new parser might be the ability to experiment with alternative languages which maintain the underlying semantics. That might also incorporate the identification of a parser API.

In all this design we were mindful that Csound was and must remain a cross-platform synthesis system, and should behave the same on all implementations. It would also be convenient if the building system were the same or similar on all platforms, and installation should be simple — accessible to users at any computer-literate level.

The other overriding requirement is that the system must not change externally, in the sense that all old music pieces must still render to the same audio. We can add new functionality, but visible things must not be removed.

3 Implementation

The previous section described the desired features of the new Csound. But they are wishes. In this section we consider the translations of these aspirations to actual code.

The API is largely the work of Gogins, but there is a number of basic concepts in the solution. The implementation is by a global structure that is passed as an argument to most functions. Within the structure there are at least three groups of slots. The first group incorporates the main API functions; functions to control Csound, such as Perform, Compile, PerformKsmps, Cleanup and Reset. There are also functions in this structure to allow the controlling program to interrogate Csound, to determine the sampling rate, the current time position and so forth. These functions are also used by user-defined opcode libraries to link to the main engine. The last group are the state variables for the instantiation of Csound.

The transition to allowing a re-entrant system is largely one of moving static variables into the system-wide structure. Code simplicity is maintained by creating C macros so access can be via the same text as previously. By adding an additional argument to every opcode of this environment structure a great simplification of much of the code is achieved, especially for user-defined opcodes, as described in more detail below (section 4).

Every opcode now returns an error code, or zero if it succeeded. This is a facility that has not been exploited yet, but it should be possible to move more of the error messages from the main engine, and incidentally to facilitate internationalisation.

The decision to use an external library for reading and writing sound files was an easy one; what was less easy was deciding which one to use. A number were considered, both the small and simple, and the all-embracing. The one we chose was Libsndfile (de Castro Lopo, 2005). The library is subject to LGPL, but the deciding factor was the helpful attitude of the author. We have not regretted this decision, and it was moderately easy to replace the complex accumulation of AIFF, AIFC and WAV with the cleaner abstraction. The hardest part was that Libsndfile works in frames and Csound has been written in samples or sometimes bytes. Of particular note was the removal of many lines of code that dealt with multiple formats (alaw, μlaw, signed and unsigned...).

There seemed less choice with the real-time audio library; PortAudio (Bencina and Burk, 2005; Bencina and Burk, 2001) seemed obvious. As the library was in transition from version 18 to 19 we decided to look ahead and use v19. This has been a more problematic decision. For example Csound is written with a blocking I/O model for audio, but to date of writing this is not implemented on all platforms, and we are using a hybrid, implementing blocking via callbacks and threads on some systems, and simple blocking I/O on others. There have even been suggestions that we abandon this library as it has not (yet) delivered the simplicity we seek. I think this can be overcome, and the decision was correct, but there are clearly problems remaining in this area.

The companion to PortAudio in the PortMusic project(Por, 2005) for MIDI is PortMIDI(Dannenberg, 2005). This was the obvious choice to support MIDI. The software models are fairly far apart but it has been incorporated. What we do not seem to be able to find is a library for file-based MIDI. At present we are continuing to use the original Vercoe code, with what looks like duplication in places. This needs to be investigated further.

There is a surfeit of graphical toolkits, at many levels of complexity. Based on previous experience, both outside Csound and inside with CsoundAV(Maldonado, 2005), FLTK was chosen. This is simple and light-weight. There are undoubtedly faster libraries, but graphics performance is not of the essence and the simplicity is worth the loss. A drawback is that this is a C++ library, whereas Csound is still at heart a C program. However in the medium term I still intend that graphics should be moved out of the core Csound and we should use the API and software bus.

A contentious issue (at least within our developer community) has been a framework for common building. For most of the life of Csound there have been three major builds, for Linux, Windows and Macintosh. The Linux and Unix system use a hand crafted makefile; on Windows a Microsoft Visual C++ IDE was used and on Macintosh the Codewarrior IDE. The redesign of Csound coincided with the acceptance of OSX on the Macintosh, and the availability of the MinGW package for Windows. This suggests that it should be possible to have a common tool-chain. Initial experience with the GNU tools (automake, autoconf etc) was highly negative, with incompatibilities between platforms, and between different releases of Linux. We are now using SCons(SCo, 2005) which is a Python-based building system which we have found to work cleanly on our three major platforms, and to have sufficient flexibility.

The first implementation of a software bus has been made, by offering an arbitrary number of uni-directional audio and control buses. This facility remains to be exploited.

The most problematic area of the implementation is the parser. A Flex-based lexer and a Bison parser have been developed[1] and these implement most of the current Csound language. The problem of joining this front-end into the internal structures remains as a major task that has not yet been attempted. The design of the parser will allow user-defined op-

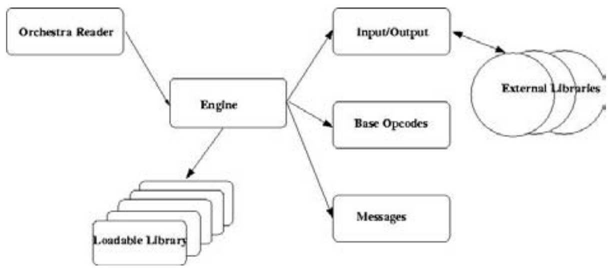


Figure 2: Architecture of Csound5

codes as is essential, as well as functions of one or more arguments. The main incompatibilities are in the enforcement of functions as functions, which is not in the original system. It does however mend known bugs in the lexing phase, and also makes better use of temporary variables.

4 User Defined Libraries

One reason for the redesign was to allow third parties to provide new opodes, either as open source or as compiled libraries that can be loaded into Csound. The user opcodes are compiled into .DLL or shared libraries, and the initialisation of Csound loads libraries as required.

User libraries were introduced in Csound4, but in Csound5 they have been extensively developed. We provide C macros to allow the library to be written in much the same way as base opcodes, and proforma structures to link the opcodes into the system. We have also recently made it possible to have library-defined table generators as well. The macros wrap the usual internal Csound functions as called via the global environment structure.

To prove that the mechanism works, many of the opcodes were remade as loadable code. The final decision as to which opcodes will be in the base and which loadable is not settled, but the overall structure of Csound is now changed from the architecture of figure 1 to that of figure 2. With this architecture we hope that clearer separation will make maintenance simpler.

5 Experience

In many ways it is too early to make informed judgements on the new Csound. On the other hand the system has been in a running state for many months, and on at least the Linux platform it is usable. Despite some rough edges it renders to both file and audio, and there are no appreciable performance issues.

The use of Libsndfile has been a very positive experience on all platforms. PortAudio has

[1]The parse is not based on the earlier Bernardini parser, but created with the support of Epigon Audiocare Pvt Ltd

caused some problems; with ALSA on Linux it is acceptable, but there have been latency problems on Windows and a number of ongoing problems on OSX, with lack of blocking I/O and an apparent need for multiple copying of audio data. There are enough indications from the PortAudio development community to say that this will work to our advantage eventually. It is still too soon to comment on the MIDI components.

There are still questions that arise from graphics and in particular the control of multiple threads. I believe that the solution is to use the software bus and outsource graphical activity to a controlling program. The graphics does work as well as it did on Csound4, but problems arise with the internal generation of GUIs for performance systems.

The code freeze has had a number of minor positive effects; the code has been subjected to coherent scrutiny without pressures for releases. Multiple identical functions have been combined, and many small coding improvements have been made, for both stylistic and accuracy reasons.

The current state is close to release. It might be possible to release before the incorporation of the parser, but this would be a disappointment to me. The other aspect that may delay release is documentation. The manual still needs updating. Basic information on the system already exists.

The decision to use SCons has proved excellent. It is working on Windows and OSX as well as the usual development Linux platforms.

6 Conclusions

In this paper I have described the thoughts behind the creation of the next incarnation of Csound. Evolution rather than revolution has been the key, but we are creating an embeddable system, a system more extensible than previously, and with clear component divisions, while preserving the operations and functionality that our users have learnt to expect. By concentrating on an embeddable core I hope that the tendency to create variants will be discouraged, and from my point of view I will not have to worry about graphics, which interests me not at all!

While the system has remained a cross-platform one, development has been mainly on Linux, and we have seen great benefits from all the tools there. When Csound5 reaches its distribution time soon, the musical community will also see these benefits.

7 Acknowledgements

My thanks go to everyone on the Csound Development mailing list for all their input, comments and reports, and also to all the Csound users who made it so clear what they wanted. Particular thanks are due to Michael Gogins for his insistence on a sane API, and to Richard Boulanger who has been a driving force behind me in the development and maintenance of Csound.

References

Ross Bencina and Phil Burk. 2001. PortAudio – an Open Source Cross Platform Audio API. In *ICMC2001*. ICMA, September.

Ross Bencina and Phil Burk. 2005. PortAudio. `http://www.portaudio.com/`.

Richard Boulanger, editor. 2000. *The Csound Book: Tutorials in Software Synthesis and Sound Design*. MIT Press, February.

Roger B. Dannenberg. 2005. PortMIDI. `http://www-2.cs.cmu.edu/~music/portmusic/portmidi`.

Erik de Castro Lopo. 2005. Libsndfile. `http://www.mega-nerd.com/libsndfile/`.

John ffitch and Julian Padget. 2002. Learning to play and perform on synthetic instruments. In Mats Nordahl, editor, *Voices of Nature: Proceedings of ICMC 2002*, pages 432–435, School of Music and Music Education, Göteborg University, September. ICMC2002, ICMC.

Gabriel Maldonado. 2005. Csoundav. `http://www.csounds.com/maldonado/download.htm`.

Dave Phillips. 2000. *The Book of Linux Music and Sound*. No Starch Press. ISBN: 1886411344.

2005. PortMusic. `http://www-2.cs.cmu.edu/~music/portmusic/`.

2005. SCons. `http://www.scons.org/`.

Barry Vercoe, 1993. *Csound — A Manual for the Audio Processing System and Supporting Programs with Tutorials*. Media Lab, M.I.T.

Barry Vercoe. 1996. Extended Csound. In *On the Edge*, pages 141–142. ICMA, ICMA and HKUST. ISBN 0-9667917-4-2.

CLAM, an Object Oriented Framework for Audio and Music

Pau Arumí and **Xavier Amatriain**
Music Technology Group, Universitat Pompeu Fabra
08003 Barcelona, Spain
{parumi,xamat}@iua.upf.es

Abstract

CLAM is a C++ framework that is being developed at the Music Technology Group of the Universitat Pompeu Fabra (Barcelona, Spain). The framework offers a complete development and research platform for the audio and music domain. Apart from offering an abstract model for audio systems, it also includes a repository of processing algorithms and data types as well as a number of tools such as audio or MIDI input/output. All these features can be exploited to build cross-platform applications or to build rapid prototypes to test signal processing algorithms.

Keywords

Development framework, DSP, audio, music, object-oriented

1 Introduction

CLAM stands for C++ Library for Audio and Music and is a full-fledged software framework for research and application development in the audio and music domain. It offers a conceptual model as well as tools for the analysis, synthesis and transformation of audio signals.

The initial objective of the CLAM project was to offer a complete, flexible and platform independent sound analysis/synthesis C++ platform to meet the needs of all the projects of the Music Technology Group (MTG, 2004) at the Universitat Pompeu Fabra in Barcelona. Those initials objectives have slightly changed since then, mainly because the library is no longer seen as an internal tool for the MTG but as a framework licensed under the GPL (Free Software Foundation, 2004).

CLAM became public and Free in the course of the AGNULA IST European project (Consortium, 2004). Some of the resulting applications as well as the framework itself were included in the Demudi distribution.

Although nowadays most the development is done under GNU/Linux, the framework is cross-platform. All the code is ANSI C++ and it is regularly compiled under GNU/Linux, Windows and Mac OSX using the GNU C++ compiler but also the Microsoft compiler.

CLAM is Free Software and all its code and documentation can be obtained though its web page (www CLAM, 2004).

2 What CLAM has to offer ?

Although other audio-related environments exist [1] —see (Amatriain, 2004) for an extensive study and comparison of most of them— there are some important features of our framework that make it somehow different:

- All the code is *object-oriented* and written in C++ for efficiency. Though the choice of a specific programming language is no guarantee of any style at all, we have tried to follow solid design principles like design patterns (Gamma E. and J., 1996) and C++ idioms (Alexandrescu, 2001), good development practices like test-driven development (Beck, 2000) and refactoring (Fowler et al., 1999), as well as constant peer reviewing.
- It is *efficient* because the design decisions concerning the generic infrastructure have been taken to favor efficiency (i.e. inline code compilation, no virtual methods calls in the core process tasks, avoidance of unnecessary copies of data objects, etc.)
- It is *comprehensive* since it not only includes classes for processing (i.e. analysis, synthesis, transformation) but also for audio and MIDI input/output, XML and SDIF serialization services, algorithms, data visualization and interaction, and multi-threading.
- CLAM deals with wide variety of *extensible data types* that range from low-level signals

[1]to cite only some of them: OpenSoundWorld, PD, Marsyas, Max, SndObj and SuperCollider

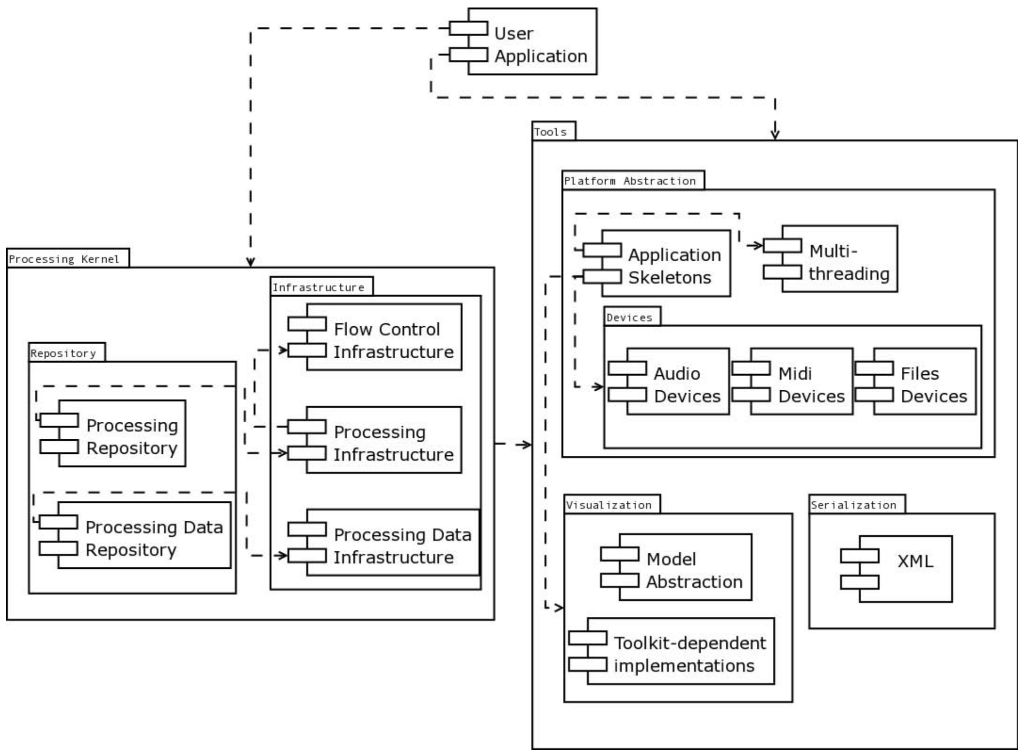


Figure 1: CLAM modules

(such as audio or spectrum) to higher-level semantic-structures (a musical phrase or an audio segment)

- As stated before, it is *cross-platform*
- The project is licensed under the *GPL* terms and conditions.
- The framework can be used either as a regular C++ *library* or as a *prototyping tool*.

In order to organise all these features CLAM is divided into different architectonic modules. Figure 1 shows the modules and submodules that exist in CLAM. The most important ones are those related to the processing kernel, with its *repositories* and *infrastructure* modules. Furthermore, a number of auxiliary *tools* are also included.

In that sense, CLAM is both a *black-box* and a *white-box* framework (Roberts and Johnson, 1996). It is black-box because already built-in components included in the repositories can be connected with minimum programmer effort in order to build new applications. And it is *white-box* because the abstract classes that make up the infrastructure can be easily derived in order to extend the framework components with new processes or data classes.

2.1 The CLAM infrastructure

The CLAM infrastructure is defined as the set of abstract classes that are responsible for the white-box functionality of the framework and define a related *metamodel* [2]. This metamodel is very much related to the Object-Oriented paradigm and to Graphical Models of Computation as it defines the object-oriented encapsulation of a mathematical graph that can be effectively used for modeling signal processing systems in general and audio systems in particular.

The metamodel clearly distinguishes between two different kinds of objects: *Processing* objects and *Processing Data* objects. Out of the two, the first one is clearly more important as the managing of Processing Data constructs can be almost transparent for the user. Therefore, we can view a CLAM system as a set of Processing objects connected in a graph called *Network*.

Processing objects are connected through intermediate channels. These channels are the only mechanism for communicating between Processing objects and with the outside world. Messages are enqueued (produced) and de-

[2]The word *metamodel* is here understood as a "model of a family of related models", see (Amatriain, 2004) for a thorough discussion on the use of metamodels and how *frameworks* generate them.

queued (consumed) in these channels, which acts as FIFO queues.

In CLAM we clearly differentiate two kinds of communication channels: *ports* and *controls*. Ports have a synchronous data flow nature while controls have an asynchronous nature. By synchronous, we mean that messages get produced and consumed at a predictable —if not fixed— rate. And by asynchronous we mean that such a rate doesn't exist and the communication follows an event-driven schema.

Figure 2 is a representation of a CLAM processing. If we imagine, for example, a processing that performs a frequency-filter transformation on an audio stream, it will have an input and an out-port for the incoming audio stream and processed output stream. But apart from the incoming and outcoming data, some other entity —probably the user through a GUI slider— might want to change some parameters of the algorithm.

This control data (also called events) will arrive, unlike the audio stream, sparsely or in bursts. In this case the processing would want to receive these control events through various (input) control channels: one for the gain amount, another for the frequency, etc.

The streaming data flows through the ports when a processing is fired (by receiving a *Do()* message).

Different processings can consume and produce at different velocities or, likewise, a different number of tokens. Connecting these processings is not a problem as long as the ports are of the same data type. The connection is handled by a *FlowControl* entity that figures out how to schedule the firings in a way that avoids firing a processing with not enough data in its input-port or not enough space into its output-ports.

Configurations: why not just controls? Apart from the input-controls, a processing receives another kind of parameter: the configurations.

Configurations parameters, unlike controls parameters, are dedicated to expensive or structural changes in the processing. For instance, a configuration parameter can decide the number of ports that a processing will have. Therefore, a main difference with controls is that they can only be set into a processing when they are not in running state.

Composites: static vs dynamic It is very convenient to encapsulate a group of processings that works together into a new composite processing. Thus, enhancing the abstraction of processes.

CLAM have two kinds of composites: *static* or hardcoded and *dynamic* or nested-networks. In both cases inner ports and controls can *published* to the parent processing.

Choosing between the static vs dynamic composites is a trade-off between boosting efficiency or understandability. See in-band pattern in (Manolescu, 1997).

2.2 The CLAM repositories

The *Processing Repository* contains a large set of ready-to-use processing algorithms, and the *Processing Data Repository* contains all the classes corresponding to the objects being processed by these algorithms.

The Processing Repository includes around 150 different Processing classes, classified in the following categories: Analysis, ArithmeticOperators, AudioFileIO, AudioIO, Controls, Generators, MIDIIO, Plugins, SDIFIO, Synthesis, and Transformations.

Although the repository has a strong bias toward spectral-domain processing because of our group's background and interests, there are enough encapsulated algorithms and tools so as to cover a broad range of possible applications.

On the other hand, in the Processing Data Repository we offer the encapsulated versions of the most commonly used data types such as Audio, Spectrum, SpectralPeaks, Envelope or Segment. It is interesting to note that all of these classes have interesting features such as a homogeneous interface or built-in automatic XML persistence.

2.3 Tools

XML Any CLAM *Component* can be stored to XML as long as `StoreOn` and `LoadFrom` methods are provided for that particular type (Garcia and Amatrian, 2001). Furthermore, Processing Data and Processing Configurations – which are in fact Components– make use of a macro-derived mechanism that provides automatic XML support without having to add a single line of code (Garcia and Amatrian, 2001).

GUI Just as almost any other framework in any domain, CLAM had to think about ways of integrating the core of the framework tools with a graphical user interface that may be used as a front-end of the framework functionalities.

The usual way to work around this issue is to decide on a graphical toolkit or framework and

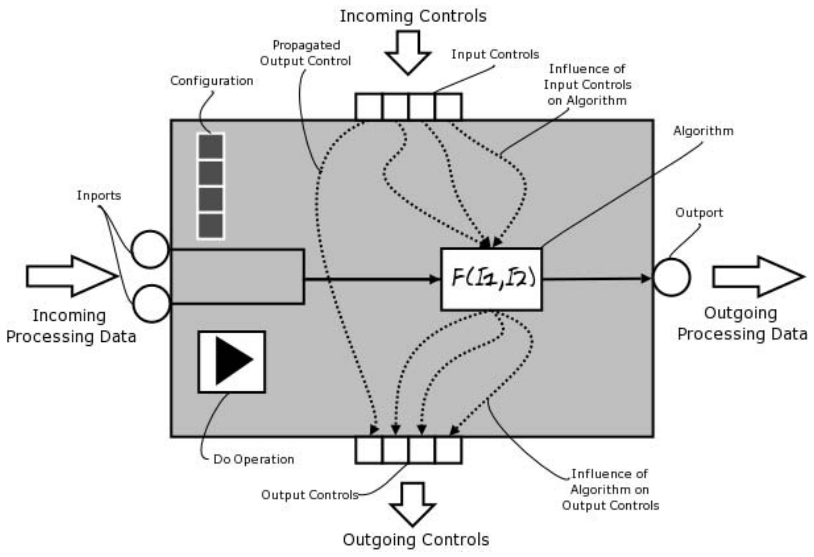


Figure 2: CLAM processing detailed representation

add support to it, offering ways of connecting the framework under development to the widgets and other graphical tools included in the graphical framework. The CLAM team, however, aimed at offering a toolkit-independent support. This is accomplished through the CLAM Visualization Module.

This general Visualization infrastructure is completed by some already implemented presentations and widgets. These are offered both for the FLTK toolkit (FLTK, 2004) and the QT framework (Blanchette and Summerfield, 2004; Trolltech, 2004). An example of such utilities are convenient debugging tools called Plots. Plots offer ready-to-use independent widgets that include the presentation of the main Processing Data in the CLAM framework such as audio, spectrum, spectral peaks…

Platform Abstraction Under this category we include all those CLAM tools that encapsulate system-level functionalities and allow a CLAM user to access them transparently from the operating system or platform.

Using these tools a number of services –such as Audio input/output, MIDI input/output or SDIF file support– can be added to an application and then used on different operating systems without changing a single line of code.

3 Levels of automation

The CLAM framework offers three different levels of automation to a user who wants to use its repositories, which can also be seen as different

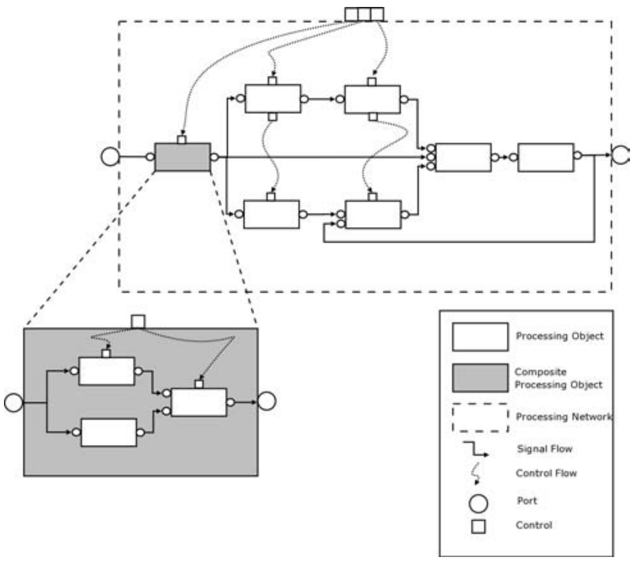


Figure 3: a CLAM processing network

levels of use of the generic infrastructure:

Library functions The user has explicit objects with processings and processing data and calls processings *Do* methods with data as its parameters. Similarly as any function library.

Processing Networks The user has explicit processing objects but streaming data is made implicit, through the use of ports. Nevertheless, the user is in charge of *firing*, or calling a *Do()* method without parameters.

Automatic Processing Networks It offers a higher level interface: processing objects are hidden behind a layer called Network, see Figure 3 Thus, instantiation of processing objects are made through passing strings identifiers to a factory. Static factories are a well documented C++ idiom (Alexandrescu, 2001) that allow us to decouple the factory class with its registered classes in a very convenient way. They makes the process of adding or removing processings to the repository as easy as issuing a single line of code in the processing class declaration.

Apart from *instantiation*, the Network class offers interface for connecting the components processings and, most important, it automatically controls the firing of processings (calling its *Do* method).

Actually, the firing scheduling can follow different strategies, for example a *push strategy* starting firing the up-source processings, or a *pull strategy* where we start querying for data to the most down-stream processings, as well as being dynamic or static (fixed list of firings). See (Hylands and others, 2003; www Ptolemy, 2004) for more details on scheduling dataflow systems.

To accommodate all this variability CLAM offers different FlowControl sub-classes which are in charge of the firing strategy, and are pluggable to the Network processing container.

4 Integration with GNU/Linux Audio infrastructure

CLAM input/output processings can deal with a different kinds of device abstraction architectures. In the GNU/Linux platform, CLAM can use audio and midi devices through the ALSA layer (www ALSA, 2004), and also through the portaudio and portmidi (www PortAudio, 2004; www PortMidi, 2004) layers.

ALSA: ALSA low-latency drivers are very important to obtain real-time input/output processing. CLAM programs using a good soundcard in conjunction with ALSA drivers and a well tuned GNU/Linux system —with the real-time patch— obtains back-to-back latencies around 10ms.

Audio file libraries: Adding audio file writing and reading capability to CLAM has been a very straight-forward task since we could delegate the task on other good GNU/Linux libraries: *libsndfile* for uncompressed audio formats, *libvorbis* for ogg-vorbis format and finally *libmad* and *libid3* for the mp3 format.

Jack: Jack support is one of the big to-dos in CLAM. It's planned for the 1.0 release or before —so in a matter of months. The main problem now is that Jack is callback based while current CLAM I/O is blocking based. So we should build an abstraction that would hide this peculiarity and would show those sources and sinks as regular ones.

LADSPA plugins: LADSPA architecture is fully supported by CLAM. On one hand, CLAM can host LADSPA plugins. On the other hand, processing objects can be compiled as LADSPA plugins.

LADSPA plugins transform buffers of audio while can receive control events. Therefore these plugins map very well with CLAM processings that have exclusively audio ports (and not other data types ports) and controls.

CLAM takes advantage of this fact on two ways: The *LADSPA-loader* gets a *.so* library file and a plugin name and it automatically instantiate a processing with the correspondent audio ports and controls. On the other hand, we can create new LADSPA plugins by just compiling a C++ template class called LadspaProcessingWrapper, where the template argument is the wrapped processing class.

DSSI plugins: Although CLAM still does not have support for DSSI plugins, the recent development of this architecture allowing graphical user interface and audio-instruments results very appealing for CLAM. Thus additions in this direction are very likely. Since CLAM provides visual tools for rapid prototyping applications with graphical interface, these applications are very suitable to be DSSI plugins.

4.1 What CLAM can be used for ?

The framework has been tested on —but also has been driven by— a number of applications, for instance: SMSTools, a SMS Analysis/Synthesis (Serra, 1996) graphical tool; Salto (Haas, 2001), a sax synthesizer; Rappid (Robledo, 2002) a real-time processor used in live performances.

Other applications using CLAM developed at the research group includes: audio features extraction tools, time-stretching plugins, voice effect processors, etc.

Apart from being a programmers framework to develop applications, the latest developments in CLAM have brought important features that fall into the *black-box* and *visual builder* categories.

That lets a user concentrate on the research of algorithms forgetting about application de-

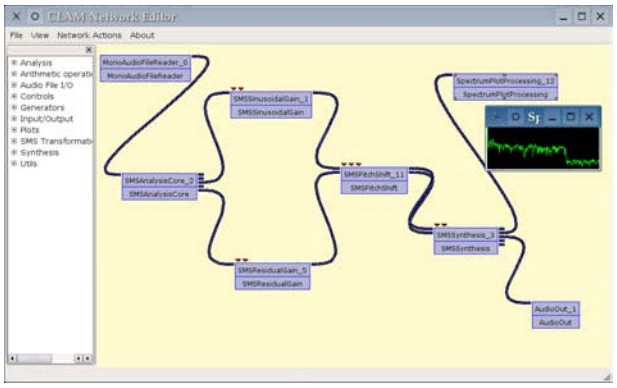


Figure 4: NetworkEditor, the CLAM visual builder

velopment. And, apart from research, it is also valuable for rapid application prototyping of applications and audio-plugins.

5 Rappid Prototyping in CLAM

5.1 Visual Builder

Another important pattern that CLAM uses is the *visual builder* which arises from the observation that in a *black-box* framework, when connecting objects the connection script is very similar from one application to another.

Acting as the *visual builder*, CLAM have a graphical program called NetworkEditor that allows to generate an application –or at least its processing engine– by graphically connecting objects. And another application called Prototyper, that acts as the glue between an application GUI designed with a graphical tool and the processing engine defined with the NetworkEditor.

5.2 An example

Here we will show how we can set up a graphical stand-alone program in just few minutes. The purpose of this program is to make some spectral transformations in real-time with the audio taken from the audio-card, apply the transformations and send the result back to the audio-card. The graphical interface will consist in a simple pane with different animated representations of the result of the spectral analysis, and three sliders to change transformation parameters.

First step: building the processing network (Figure 4) Patching with NetworkEditor is a very intuitive task to do. See Figure 4. We can load the desired processings by dragging them from the left panel of the window. Once in the patching panel, processings are viewed as little boxes with attached inlets and outlets representing its ports and control. The application allows all the typical mouse operations like select, move, delete and finally, connect ports and controls.

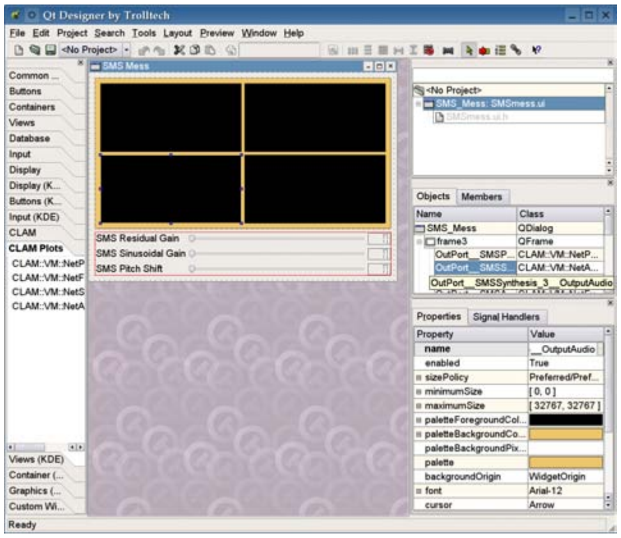
Since CLAM ports are typed, not all outports are compatible with all in-ports. For example in the Figure 4, the second processing in the chain is called SMSAnalysis and receives audio samples and produces: sinusoidal peaks, fundamental, several spectrums (one corresponding to the original audio and another corresponding to the residual resulting of subtracting the sinusoidal component).

Connected to SMSAnalysis out-ports we have placed three processings to perform transformations: one for controlling the gain of the sinusoidal component, another to control the gain of the residual component and the last one for shifting the pitch. The latest modifies both sinusoidal and residual components.

Then the signal chain gets into the SMSSynthesis which output the resynthesizes audio ready to feed the AudioOut (which makes the audio-card to sound)

Before starting the execution of the network, we can right click upon any processing view to open a dialog with its configuration. For instance, the SMSAnalysis configuration includes the window type and window size parameters among many others.

Another interesting feature of the NetworkEditor is that it allows loading visual plots widgets for examining the flowing data in any out-port. And also, slider widgets to connect to



Name	Class
SMS_Mess	QDialog
frame3	QFrame
OutPort__SMSP...	CLAM::VM::NetP...
OutPort__SMSS...	CLAM::VM::NetA...

OutPort__SMSSynthesis_3__OutputAudio

Property	Value
name	__OutputAudio
enabled	True
sizePolicy	Preferred/Pref...
minimumSize	[0, 0]
maximumSize	[32767, 32767]
paletteForegroundCol...	■
paletteBackgroundCo...	■
paletteBackgroundPix...	
palette	■
backgroundOrigin	WidgetOrigin
font	Arial-12
cursor	Arrow

Figure 5: the QT GUI designer tool

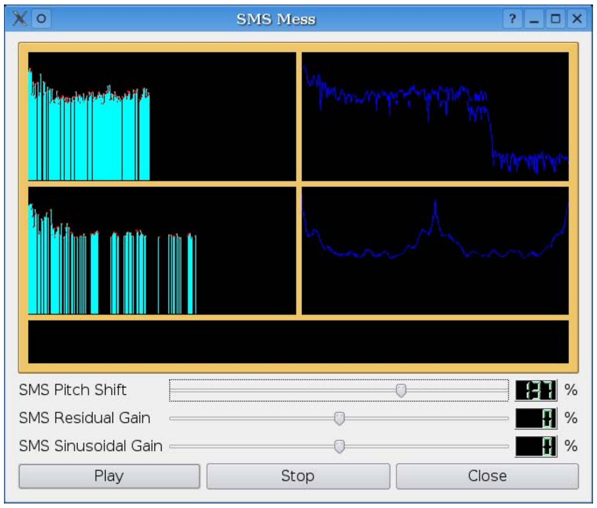


Figure 6: the running prototype

the in-control inlets.

Once the patch is finished we are ready to move on directly to the graphical user interface.

Second step: designing the program GUI (Figure 5) The screen-shot in Figure 5 is taken while creating a front end for our processing network. The designer is a tool for creating graphical user interfaces that comes with the QT toolkit (Blanchette and Summerfield, 2004; Trolltech, 2004).

Normal sliders can be connected to processing in-ports by just setting a suited name in the properties box of the widget. Basically this name specify three things in a row: that we want to connect to an in-control, the name that the processing object have in the network and the name of the specific in-control.

On the other hand we provide the designer with a *CLAM Plots* plugin that offers a set of plotting widgets that can be connected to out-ports.

In the example in Figure 5 the black boxes corresponds to different plots for spectrum, audio and sinusoidal peaks data.

Now we just have to connect the plots widgets by specifying —like we did for the sliders— the out-ports we want to inspect.

We save the designer *.ui* file and we are ready to run the application.

Third step: running the prototype (Figure 6) Finally we run the prototyper program. Figure 6. It takes two arguments, in one hand, the xml file with the network specification and in the other hand, the designer ui file.

This program is in charge to load the network from its xml file —which contains also each processing configuration parameters— and create objects in charge of converting QT signals and slots with CLAM ports and controls.

And done! we have created, in a matter of minutes, a prototype that runs fast C++ compiled code without compiling a single line.

6 Conclusions

CLAM has already been presented in other conferences like the OOPSLA'02 (Amatriain et al., 2002b; Amatriain et al., 2002a) but since then, a lot of progress have been taken in different directions, and specially in making the framework more *black-box* with *visual builder* tools.

CLAM has proven being useful in many applications and is becoming more and more easy to use, and so, we expect new projects to begin using the framework. Anyway it has still not reached a the stable 1.0 release, and some improvements needs to be done.

See the CLAM roadmap in the web (www CLAM, 2004) for details on things to be done. The most prominent are: *Library-binaries* and separate submodules, since at this moment modularity is mostly conceptual and at the source code organization level. Finish the audio *feature-extraction framework* which is work-in-progress. *Simplify parts of the code*, specially the parts related with processing data and configurations classes. Have working *nested networks*

7 Acknowledgements

The authors wish to recognize all the people who have contributed to the development of the CLAM framework. A non-exhaustive list should at least include Maarten de Boer, David Garcia, Miguel Ramírez, Xavi Rubio and Enrique Robledo.

Some of the the work explained in this paper has been partially funded by the Agnula Europan Project num.IST-2001-34879.

References

A. Alexandrescu. 2001. *Modern C++ Design.* Addison-Wesley, Pearson Education.

X. Amatriain, P. Arumí, and M. Ramírez. 2002a. CLAM, Yet Another Library for Audio and Music Processing? In *Proceedings of the 2002 Conference on Object Oriented Programming, Systems and Applica-*

tion (OOPSLA 2002)(Companion Material), Seattle, USA. ACM.

X. Amatriain, M. de Boer, E. Robledo, and D. Garcia. 2002b. CLAM: An OO Framework for Developing Audio and Music Applications. In *Proceedings of the 2002 Conference on Object Oriented Programming, Systems and Application (OOPSLA 2002)(Companion Material)*, Seattle, USA. ACM.

X. Amatriain. 2004. *An Object-Oriented Metamodel for Digital Signal Processing*. Universitat Pompeu Fabra.

K Beck. 2000. *Test Driven Development by Example*. Addison-Wesley.

J. Blanchette and M. Summerfield. 2004. *C++ GUI Programming with QT 3*. Pearson Education.

AGNULA Consortium. 2004. AGNULA (A GNU Linux Audio Distribution) homepage, http://www.agnula.org.

FLTK. 2004. The fast light toolkit (fltk) homepage: http://www.fltk.org.

M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

Free Software Foundation. 2004. Gnu general public license (gpl) terms and conditions. http://www.gnu.org/copyleft/gpl.html.

Johnson R. Gamma E., Helm R. and Vlissides J. 1996. *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

D. Garcia and X. Amatrian. 2001. XML as a means of control for audio processing, synthesis and analysis. In *Proceedings of the MOSART Workshop on Current Research Directions in Computer Music*, Barcelona, Spain.

J. Haas. 2001. SALTO - A Spectral Domain Saxophone Synthesizer. In *Proceedings of MOSART Workshop on Current Research Directions in Computer Music*, Barcelona, Spain.

C. Hylands et al. 2003. Overview of the Ptolemy Project. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berklee, California.

D. A. Manolescu. 1997. A Dataflow Pattern Language. In *Proceedings of the 4th Pattern Languages of Programming Conference*.

MTG. 2004. Homepage of the Music Technology Group (MTG) from the Universitat Pompeu Fabra. http://www.iua.upf.es/mtg.

D. Roberts and R. Johnson. 1996. Evolve Frameworks into Domain-Specific Languages. In *Procedings of the 3rd International Conference on Pattern Languages for Programming*, Monticelli, IL, USA, September.

E. Robledo. 2002. RAPPID: Robust Real Time Audio Processing with CLAM. In *Proceedings of 5th International Conference on Digital Audio Effects*, Hamburg, Germany.

X. Serra, 1996. *Musical Signal Processing*, chapter Musical Sound Modeling with Sinusoids plus Noise. Swets Zeitlinger Publishers.

Trolltech. 2004. Qt homepage by trolltech. http://www.trolltech.com.

www ALSA. 2004. Alsa project home page. http://www.alsa-project.org.

www CLAM. 2004. CLAM website: http://www.iua.upf.es/mtg/clam.

www PortAudio. 2004. PortAudio homepage: www.portaudio.com.

www PortMidi. 2004. Port Music homepage: http://www-2.cs.cmu.edu/ music/portmusic/.

www Ptolemy. 2004. Ptolemy project home page. http://ptolemy.eecs.berkely.edu.

“Made in Linux” — The Next Step

Ivica Ico BUKVIC
College-Conservatory of Music, University of Cincinnati
3346 Sherlock Ave. #21
Cincinnati, OH, U.S.A., 45220
http://meowing.ccm.uc.edu/~ico/
ico@fuse.net

Abstract

It's been over half a decade since the Linux audio began to shape into a mature platform capable of impressing even the most genuine cynic. Although its progress remains unquestionable, the increasing bleed-over of the GNU software onto other platforms, fragmentation of the audio software market, as well as wavering hardware support, pose as a significant threat to its long-term prosperity. “Made in Linux” is a newly proposed incentive to create a non-profit foundation that will bridge the gap between the Linux audio community and the commercial audio market in order to ensure its long-term success.

Keywords

Foundation, Initiative, Exposure, Commercial, Incentives

1 Introduction

While no single event is necessarily responsible for the sudden proliferation of the Linux audio software, it is undeniable that the maturing of the ALSA and JACK frameworks were indispensable catalysts in this process. Yet, what really made this turning point an impressive feat was the way in which the Linux audio community, amidst the seemingly “standardless anarchy,” was able to not only acknowledge their quality, but also wholeheartedly embrace them. Although some users are still standing in denial of the obvious advantages heralded by these important milestones, nowadays they are but a minority. Since, we've had a number of software tools harness the power of the new framework, complementing each other and slowly shaping Linux into a complete Digital Audio Workstation (DAW) solution.

2 The Momentum

Today, while we still enjoy the momentum generated by these important events, increasing worldwide economic problems, bleed-over of the GNU software to closed dominant platforms, as well as the cascading side-effects, such as the questionable pro-audio hardware support, now stand as formidable stepping stones to the long-term success of this platform.

Even though the economic hardship would suggest greater likelihood of Linux adoption for the purpose of cutting costs, this model only works in the cases where Linux has already proven its worth, such as the server market. And while I do not mean to imply that Linux as a DAW has not proven its worth in my eyes (or should I say ears?), it is unquestionable that its value is still a great unknown among the common users who are, after all, the backbone of the consumer market and whose numbers are the most important incentive for the commercial vendors. In addition, Linux audio and multimedia users as well as potential newcomers still face some significant obstacles, such as the impressive but unfortunately unreliable support of the ubiquitous VST standard via the WINE layer, or the lack of a truly complete all-in-one DAW software.

The aforementioned platform transparency of the GNU model is a blessing and a curse. While it may stimulate the user of a closed platform to delve further into the offering of the open-source community, contribute to, and perhaps even switch to an open-source platform, generally such a behavior is still largely an exception. Let us for a moment consider the potential contributors from the two dominant platforms: Microsoft and Apple. The dominant Microsoft platform is architecturally too different, so the contributions from the users of this platform will likely be mostly porting-related and as such will do little for the betterment of the software's core functionalities (as a matter of fact, they may even cause increase in the software maintenance overhead). Similarly, the Apple users as adoring followers of their own platform usually yield similar contributions. Naturally, the exceptions to either trend are noteworthy, yet they remain to be exactly that: exceptions. While it is not my intention to trivialize such contributions nor to question the premise upon which the GNU doctrine has been built, it is quite obvious that the cross-platform model of attracting new users to the GNU/Linux platform currently does not work as expected and therefore should not be counted on as a recipe for the long-term success.

What is unfortunate in this climate of dubious cross-platform contributions and dwindling economic prospects through fragmentation of the audio software industry, is the fact that it generates a cascading set of side-effects. One of such effects is the recently disclosed lack of RME's interest, a long-term supporter of the Linux audio efforts, to provide ALSA developers with the specifications for its firewire audio hardware due to IP concerns. Even though such a decision raises some interesting questions, delving any further into the issue is certainly outside the scope of this paper. Yet, the fact remains that without the proper support for the pro-audio interfaces of tomorrow, no matter how good the software, Linux audio has no future. The situation is certainly not as grim as it seems as there are many other audio devices that are, and will continue to be supported. Nonetheless, this may become one of the most important stepping stones in the years to come and therefore should be used as a warning that may very well require a preemptive (re)action from the Linux audio community before it becomes too late.

3 Counter-Initiatives

Amidst these developments, our community has certainly not been dormant. There were numerous initiatives that were usually spawned by individuals or small groups of like-minded enthusiasts in order to foster greater cooperation among the community members and attract attention from the outsiders, such as the *Linux Audio Consortium* of libre software and companies whose primary purpose is to help steer further developments as well as serve as a liaison between the commercial audio world and the Linux audio community. Another example is the “Made with Linux” CD which is to include a compilation of works made with Linux and whose dissemination would be used as a form of publicity as well as for the fund-raising purposes. Other examples include numerous articles and publications in reputable magazines that expose the true power of Linux as well as recently increased traffic on the Linux-Audio-User and Linux-Audio-Developer mailing lists concerning works made using Linux.

These are by no means the only gems of such efforts. Nonetheless, their cumulative effect has to this day made but a small dent in exposing the true power of Linux as a DAW. To put this statement into perspective, even the tech-savvy and generally pro-Linux audience of the *Slashdot* technology news site is largely still ignorant of Linux's true multimedia content creation and production potential.

All this points to the fact that Linux audio community has reached the point of critical mass at which all involved need to take the next step in organizing efforts towards expanding the audio software market exposure, whether for the reasons of own gratification, financial gain, or simply for the benefit of the overall community. After all, if the Linux audio supporters do not take these steps then it should certainly not be expected from others to follow or even less likely take the steps in their stead.

4 “Made in Linux” to the Rescue

“Made in Linux” is an initiative carrying a deliberate syntactic pun in its title to separate itself from other similar programs and/or initiatives that may have already taken place and/or are associated with the more generalized form of evangelizing Linux. The title obviously plays a pun on the labels commonly found on commercial products in order to identify their country of assembly, less commonly the country of their origin. Such ubiquitous practice nowadays makes it nearly impossible to find a commercial product without such a label. Considering that the initiative I am proposing should be as vigilant and as allencompassing as the aforementioned labels, I felt that the title certainly fit the bill.

The initiative calls for formation of a non-profit foundation whose primary concern will be to oversee the proliferation of Linux as a DAW through widespread publicity of any marketable multimedia work that has utilized Linux, monetary incentives, awards, and perhaps most importantly through establishing of reliable communication channels between the commercial pro-audio market and the Linux audio developers, artists, and contributors. With such an agenda there are superficial but nonetheless pronounced similarities with the function and purpose behind the *Linux Audio Consortium*. However, as we will soon find out, there are some distinguishing differences as well.

One of the most important long-term goals of "Made in Linux" foundation will be to accumulate operating budget through fund-rising. Such budget would enable the foundation to provide incentives towards the development of most sought audio-related software features and/or tools, sponsoring competitions and awards for the recognition of the most important contributions to the community, media exposure, music-oriented incentives (i.e. composition competitions), and beyond.

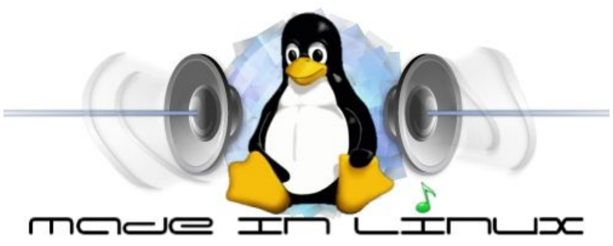
Depending upon the success of the initial deployment, the foundation's programs could easily expand to encompass other possibilities, such as the yearly publications of works in a form of a CD compilation similar to the aforementioned "Made with Linux" collection, as well as other incentives that may help the foundation become more self-dependent while at the same time allowing it to further expand its operations.

While the proposal of creating an entity that would foster Linux as a DAW proliferation certainly sounds very enticing, please let us not be deceived. Linux audio market is currently a niche within a niche, and as such does not suggest that such foundation would boast a formidable operating budget. Nonetheless, it is my belief that in time it may grow into a strong liaison between the commercial world and our, whether we like it or not, still widely questioned "GNU underground."

5 Streamlining Exposure

In order to expedite and streamline the aforementioned efforts, the "Made in Linux" program also calls for an establishment of a clearly distinguishable logo which is to be embedded into any audio software that has been conceived on a Linux platform and is voluntarily endorsing this particular initiative.

The idea to encourage all contributors, developers and artists alike, to seal their work with a clearly identifying logo is a powerful advertising mechanism that should not be taken lightly, especially considering that it suggest their devotion if not indebtedness to the GNU/Linux audio community, whether by developing software tools primarily for this platform, or by using those tools in their artistic work. More importantly, if a software application were to ported to another platform, the logo's required persistence would clearly and unquestionably reveal its origins likely elevating curiosity among users oblivious to the Linux audio scene. Although we already have several logos for the various Linux audio-related groups and/or communities, most of them are denominational and as such may not prove to be the best convergent solution. Therefore, I would like to propose the a creation of a new logo that would be preferably used by anyone who utilizes Linux for multimedia purposes. The following example being a mere suggestion, a starting point if you like, is distributed under the GNU/GPL license (larger version is freely available and downloadable from the author's website — please see references for more info):



It is of utmost importance once the logo's appearance has been finalized and ratified, that it remains constant and its use consistent in order to enable end-users to familiarize themselves with it and grasp the immensity and versatility of Linux audio offering. Naturally, the software applications that do not offer graphical user interface could simply resort to incorporating the title of the incentive. With these simple, yet important steps, the Linux multimedia software would, to the benefit of the entire community, attain greater amount of exposure and publicity.

Contrary to the aforementioned foundation, this measure is neither hard to implement nor does should it generate a significant impact from the developer's standpoint, yet it does pose as a powerful statement to the GNU/Linux cause. Just like the Linux's “Tux” mascot, which now dominates the Internet, this too can become a persistent image associated with the Linux audio software. However, in order to be able to attain this seemingly simple goal, it is imperative that the Linux audio community extends a widespread (if not unanimous) support and cooperation towards this initiative. Only then will this idea yield constructive results. Needless to mention that this prerequisite will not only test the appeal of the initiative, but also through its fruition assess the community's interest (or lack thereof) in instituting the aforementioned foundation. Once the foundation would assume its normal day-to-day operations, this measure would become integral part of the foundation's agenda in its efforts to widen the exposure of the rich Linux audio software offering.

6 Linux Audio Consortium Concerns

By now it is certainly apparent that the aforementioned initiative bears resemblance to the *Linux Audio Consortium* agenda which has been in existence for over a year now. After all, both initiatives share the same goal: proliferation of Linux audio scene by offering means of communication as well as representation. However, there are some key differences between the two initiatives.

In its current state the *Linux Audio Consortium* could conceivably sponsor or at least serve as the host for the “Made in Linux” initiative. Yet, in order for the consortium to be capable of furnishing full spectrum of programs covered by this initiative, including the creation of the aforementioned foundation, there is an unquestionable need for a source of funding. Currently, the consortium does not have the facilities that would enable such steady source of income. As such, the additional programs proposed as part of this initiative, should they be implemented under the patronage of the consortium, they would require a reasonably substantial alterations to its bylaws and day-to-day operations.

Naturally, it would be unfortunate if the two initiatives were to remain separate as such situation would introduce unnecessary fragmentation of an already humbly-sized community. Nonetheless, provided that the “Made in Linux” program creates an adequately-sized following, it may become necessary at least in initial stages for the two programs to remain separate until the logistical and other concerns of their merging are ironed out.

7 Conclusion

It is undeniable that the Linux audio community is facing some tough decisions in the imminent future. These decisions will not only test the community's integrity, but will likely determine the very future of the Linux as a software DAW. Introducing new and improving existing software, while a quintessential factor for the success in the commercial market, unfortunately may not help solve some of the fundamental issues, such as the dubious state of the pro-audio hardware support. As such, this sole incentive will not ensure the long-term success of the Linux platform. Furthermore, whether one harbors interest in a joint effort towards promoting Linux also may not matter much in this case. After all, if Linux fails to attract professional audio hardware vendors, no matter how good the software offering becomes, it will be useless without the proper hardware support. Therefore, it is the formation of the foundation (or restructuring of the existing *Linux Audio Consortium*) and its relentless promotion of the Linux audio efforts that may very well be community's only chance to push Linux into the mainstream audio market where once again it will have a relatively secure future as a professional and competitive DAW solution.

8 Acknowledgements

My thanks, as always, go to my beloved family who through all these years of hardship and many sleepless nights troubleshooting unstable Linux kernels and modules, X session setups, and audio xrun's, stood by me. Big thanks also go to all the

members of the Linux audio community without whose generous efforts none of this would have been possible.

References

ALSA website. http://www.alsa-project.org (visited on January 10, 2005).

JACK website. http://jackit.sourceforge.net/ (visited on January 10, 2005).

Linux Audio Consortium website. http://www.linuxaudio.org (visited on January 10, 2005).

Linux-audio-developers (LAU) website. http://www.linuxdj.com/audio/lad/ (visited on January 10, 2005).

"Made in Linux" logo (GIMP format). http://meowing.ccm.uc.edu/~ico/Linux/log.xcf.

Slashdot website. http://www.slashdot.org (visited on January 10, 2005).

Steinberg/VST website. http://www.steinberg.net/ (visited on January 10, 2005).

WINE HQ website. http://www.winehq.com/ (visited on January 10, 2005).

Linux Audio Usability Issues

Introducing usability ideas based on Linux audio software

Christoph Eckert
Graf-Rhena-Straße 2
76137 Karlsruhe, Germany
mchristoph.eckert@t-online.de
February 2005

Abstract

The pool of audio software for Linux based operating systems has to offer very powerful tools to grant even average computer users the joy of making music in conjunction with free software. However, there are still some usability issues. Basic usability principles will be introduced, while examples will substantiate where Linux audio applications could benefit from usability ideas.

Keywords

Usability & application development

1. Introduction

Free audio software has become very powerful during the last years. It is now rich in features and mature enough to be used by hobbyists as well as by professionals.

In the real world, day by day we encounter some odds and ends which are able to make us unhappy or frustrated. May it be a screw driver which does not fit a screw, a door which does not close properly or a knife which simply is not sharp enough.

This is also valid for software usage. There often are lots of wee small things which sum up and prevent us from doing what we originally wanted to do. Some of these circumstances can be avoided by applying already existing usability rules to Linux audio software.

Usability usually is associated with graphical user interface design, mainly on Mac OS or even Microsoft Windows operating systems. Surprisingly, most of the rules apply to any software on any operating system, including command line interfaces.

A bunch of documents discovering this area of programming are available from various projects and companies, e.g. the GNU project, the Gnome desktop environment, the wxWidgets project[1], Apple computers or individuals.

One of the basic questions is how much a user needs to know before he is able to use a tool in order to perform a certain task. The amount of required knowledge can be reduced by clever application design which grants an average user an immediate start. Last but not least, clever software design reduces the amount of documentation needed; developers dislike writing documentation as well as users dislike reading it.

2. Usability Terms

Besides the classical paper »Mac OS 8 Human Interface Guidelines« by Apple Computer, Inc[2], the Gnome project has published an excellent paper discovering usability ideas[3].

Joel Spolsky has written a book called »User Interface Design for Programmers«. It is based on Mac and Windows development, but most ideas are also valid for Linux. Reduced in size, it is also available online[4].

To get familiar with the topic, some of the commonly found usability terms will be introduced. Included examples will concretize the ideas. Most of the examples do not only belong to one usability principle but even more.

2.1 Knowing the Audience

In order to write software which enables a user to perform a certain task, it is necessary to know the audience. It is a difference if the software system will teach children to read notes or enable a musician to create music. Different groups use a different vocabulary, want to perform different tasks and may have different computer knowledge.

Each user may have individual expectations on how the software will work. The expectations are derived from the task he wants to perform. The user has a model in mind, and the better the software model fits the user model, the more the user will benefit. This can be achieved by creating use cases, virtual users or asking users for feedback, so some applications include feedback agents.

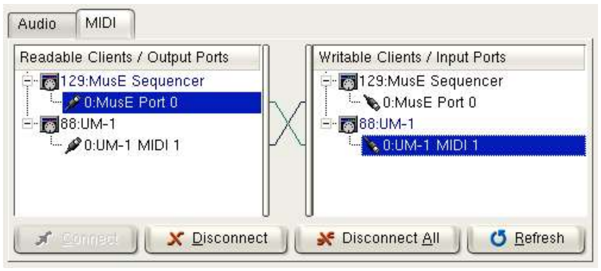
To fit the user's expectations is one of the most important and most difficult things. If the same question appears again and again in the user mailing lists, or even has been manifested in a list of frequently asked questions (known as FAQ), it is most likely that the software model does not fit the user model.

The target group of audio applications are musicians. They vary in computer skills, the music and instruments they play and finally the tasks they want to perform using the computer. Some want to produce electronic music, others want to do sequencing, hard disk recording or prepare scores for professional printing.

An audio application of course uses terms found in the world of musicians. On the other hand too specialized terms confuse the user. A piano teacher, for example, who gets interested in software sequencers, gets an easier start if the software uses terms he knows. A tooltip that reads »Insert damper pedal« can be understood more easily than »Create Controller 64 event«.

As soon as a user starts an audio software, he might expect that the software is able to output immediately sound to the soundcard. As we are on Linux, however, the user first needs to know how to set the MIDI and maybe the JACK connections.

An application therefore could make this easily accessible or at least remember the settings persistently until the next start and try to reconnect automatically. MusE for example is already doing so:

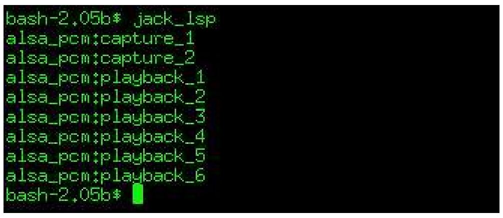


2.2 Metaphors

Metaphors are widely used to make working on computers more intuitive to the user, especially (but not only) in applications with graphical user interfaces (also known as GUI applications). Metaphors directly depend on the audience, because they often match things the user knows from the real world.

Instead of saving an URL, a bookmark gets created while surfing the web. On the other hand, choosing bad metaphors is worse than using no metaphor at all.

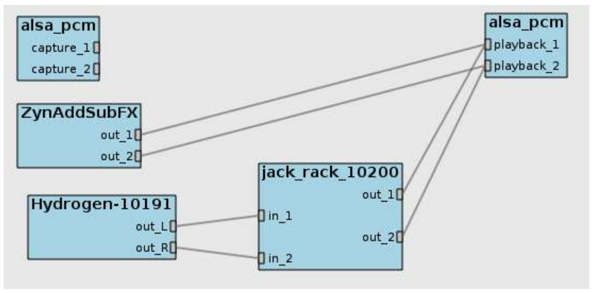
Well chosen metaphors reduce the need to understand the underlying system and therefore ensure that the user gets an immediate start. In the following example, a new user will be most probably confused by the port names:



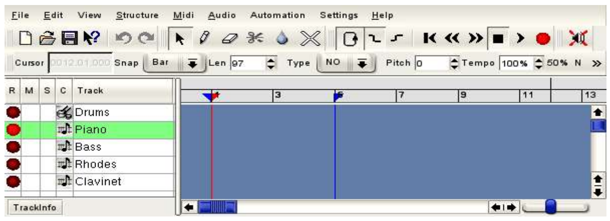
```
bash-2.05b$ jack_lsp
alsa_pcm:capture_1
alsa_pcm:capture_2
alsa_pcm:playback_1
alsa_pcm:playback_2
alsa_pcm:playback_3
alsa_pcm:playback_4
alsa_pcm:playback_5
alsa_pcm:playback_6
bash-2.05b$
```

A metaphor makes it easier to the user to understand what is meant, maybe Soundcard input 1 and 2 and Soundcard output 1 through 6 instead of alsa_pcm:capture or alsa_pcm:playback.

Patchage[6] is a nice attempt to use metaphors to visualize the data flow in a manner the audio user is familiar with, compared to connecting real world musical devices. It could become the right tool for intuitively making MIDI connections as well as JACK audio connections. If it contained a LADSPA and DSSI host, it could be a nice tool to easily control most aspects of a Linux audio environment:



An example for a misleading metaphor is an LED emulation used as a switch. An LED in real life displays a status or activity. A user will not understand that it is a control to switch something on and off. In MusE, LEDs are used to en- and disable tracks to be recorded, and this often is not clearly understood by users:

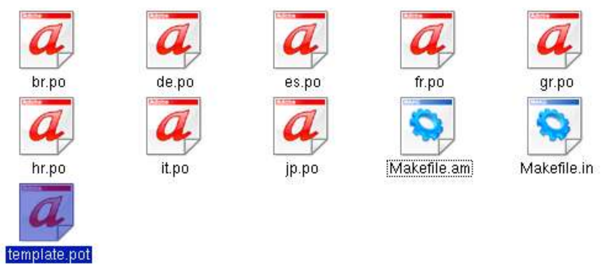


Replacing the LEDs by buttons which include an LED to clearly visualize the recording status of each track would make it easier to understand for the user.

2.3 Accessibility

All over the world, there are citizens with physical and cognitive limitations. There are blind people as well as people with hearing impairments or limited movement abilities. On a Linux system it is easily possible to design software which can be controlled using a command line interface. When designing GUI software it is still needed to include the most important options as command line options. This way even blind users are able to use a GUI software synthesizer by starting it with a certain patch file, connecting it to the MIDI input and playing it using an external keyboard controller.

Free software often gets used all over the world. It is desirable that software is prepared to easily get translated and localized for different regions. By including translation template files in the source tarball and the build process, users are able to contribute by doing translation work:



Besides Internationalization and Localization (both also known as i18n and l10n), accessibility includes paying attention to cultural and political aspects of software. Showing flags or parts of human beings causes unexpected results, maybe as soon as an icon with a hand of a western human will be seen by users in Central Africa.

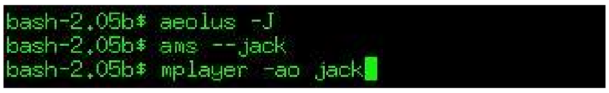
Keyboard shortcuts enable access to users who have problems using a mouse. The Alt key is usually used to navigate menus, while often needed actions are directly accessible by Ctrl-keycombos. The tabulator key is used to cycle through the controls of dialogs etc.

2.4 Consistency

Consistency is divided into several aspects of an application. It includes consistency with (even earlier versions of) itself, with the windowmanager used, with the use of metaphors and consistency with the user model.

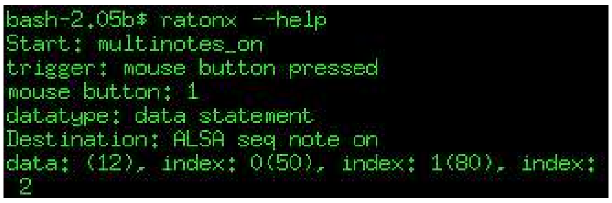
Most of the time, users do not only use one application to realize an idea. Instead, many applications are needed to perform a job, maybe to create a new song. The user tries to reapply knowledge about one application while using another. This also includes to make an application consistent with other applications even if something has been designed wrong in other applications. If an application is a Gnome application, the user expects the file open dialog to behave the same as in other Gnome applications. Writing a new file request dialog in one of the applications will certainly confuse the user even if it was better than the generic Gnome one.

Consistency does not only affect GUI programs but command line programs as well. Some Linux audio programs can be started with an option to use JACK for audio output using a command line parameter. As soon as applications behave differently, the user cannot transfer knowledge about one program to another one:



```
bash-2.05b$ aeolus -J
bash-2.05b$ ams --jack
bash-2.05b$ mplayer -ao jack
```

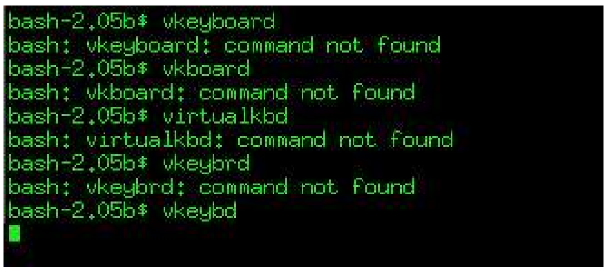
There are also programs which do not read any given command line parameters. Invoking such a program with the help parameter will simply cause it to start instead of printing some useful help on the screen:



```
bash-2.05b$ ratonx --help
Start: multinotes_on
trigger: mouse button pressed
mouse button: 1
datatype: data statement
Destination: ALSA seq note on
data: (12), index: 0(50), index: 1(80), index:
 2
```

The GNU coding standards[5] recommend to let a program at least understand certain options. To ask a program for version and help information should really included in any program. Even if there is no help available, it is helpful to clearly state this and point the user to a README file, the configuration file or an URL where he can get more information. If a GUI application contains a help menu, it is useful if it at least contains one entry. It is better to have a single help page clearly stating that there is no help at all and pointing to the project homepage or to a README file than having no help files installed at all.

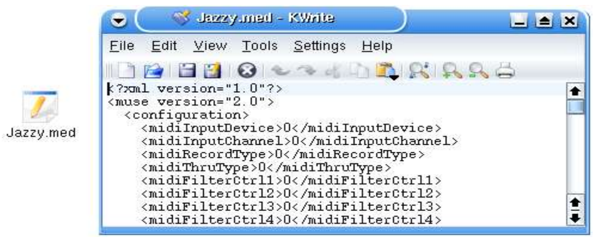
Programs containing uppercase characters in the name of the binary file confuse the user. The binary file of KhdRecord for example really reads as »KhdRecord« making it difficult for the user to start it from a command line, even if he remembers the name correctly. Another example are program's where the binary file name does not fit the application's name exactly, as found on the virtual keyboard. The user has to guess the binary representation, and this causes typing errors:



```
bash-2.05b$ vkeyboard
bash: vkeyboard: command not found
bash-2.05b$ vkboard
bash: vkboard: command not found
bash-2.05b$ virtualkbd
bash: virtualkbd: command not found
bash-2.05b$ vkeybrd
bash: vkeybrd: command not found
bash-2.05b$ vkeybd
```

GUI programs can add a menu entry to the system menu so the user is able to start the program the same way as other programs and doesn't need to remember the application's name. Therefore, GUI applications must not depend on command line options to be passed and need to print important error messages not only on the command line, but also via the graphical user interface.
For consistency reasons, such desktop integration shouldn't be left to the packagers. The user should always find an application on the same place in the menu system, regardless which distribution he is running.

Users are used to click on documents to open these in the corresponding application. So it is useful if an audio application registers the used filetypes. MusE for example saves its files as *.med files. Due to the lack of filetype registration, KDE recognizes these files as text files. Clicking on »Jazzy.med« will open it in a text editor instead of starting MusE and loading it:



```
<?xml version="1.0"?>
<muse version="2.0">
  <configuration>
    <midiInputDevice>0</midiInputDevice>
    <midiInputChannel>0</midiInputChannel>
    <midiRecordType>0</midiRecordType>
    <midiThruType>0</midiThruType>
    <midiFilterCtrl1>0</midiFilterCtrl1>
    <midiFilterCtrl2>0</midiFilterCtrl2>
    <midiFilterCtrl3>0</midiFilterCtrl3>
    <midiFilterCtrl4>0</midiFilterCtrl4>
```

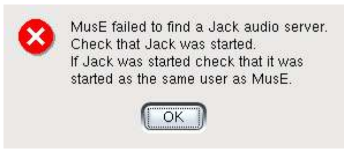
Consistency also includes the stability of an application, security aspects and its data compatibility with itself as well as other applications. Even in the world of free software it is often not simple for the user to exchange data between different applications.

2.5 Feedback

A computer is a tool to enter, process and output data. If a user has initiated a certain task, the system should keep him informed as long as there is work in progress or if something went wrong. This way the user doesn't have to guess the status of the system. The simpler a notification is, the better the user will be able to understand and to remember it after it is gone.

Providing information which enables the user to solve a problem and to avoid it in the future reduces disappointment and frustration. For the same reason message texts should be designed in a manner that the software instead of the user is responsible for errors.

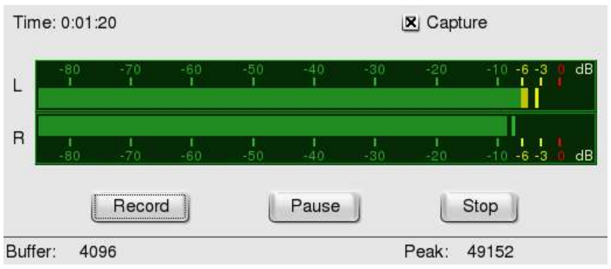
When starting MusE without a JACK soundserver already running, the user gets prompted by an alert clearly explaining the problem:



MusE failed to find a Jack audio server.
Check that Jack was started.
If Jack was started check that it was
started as the same user as MusE.

The user now has a good starting point and learns how to avoid this error in the future.

A further example for good feedback is found in qarecord[7]. It clearly shows the input gain using a nice level meter the user may know from real recording equipment:



Time: 0:01:20

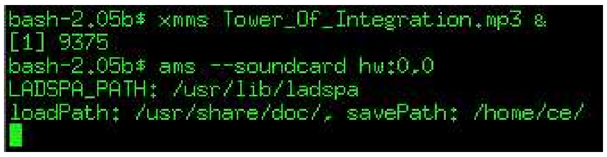
[✓] Capture

Buffer: 4096

Peak: 49152

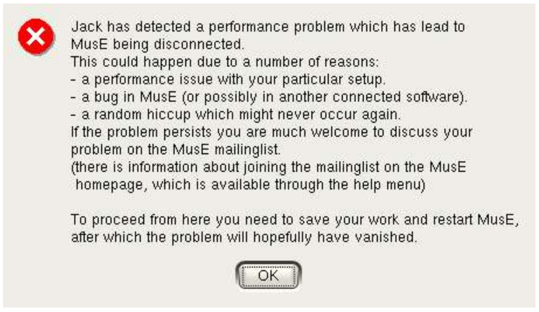
On the other hand, the user gets no feedback if currently audio is recorded or not. If no recording is in progress it makes no sense to press the pause and stop buttons, so both need to appear inactive. As soon as recording is in progress it makes no sense to press the record button again, so the record button needs to be set inactive. If the recording is paused, the pause or record button needs to be disabled.
Qarecord needs to be started in conjunction with some command line parameters defining the hardware which should be used to capture audio. If qarecord included a graphical soundcard and an input source selector as well as a graphical gain control, it would perfectly fulfill further usability ideas like direct access and user control. The user was able to do all settings directly in qarecord instead of using different applications for a simple job.

To offer feedback, it is necessary to add different checks to a software system. A user starting an audio application might expect it will immediately be able to output sound. On Linux based systems, there are some circumstances which prevent an application from outputting audio. This for example happens as soon as one application or soundserver blocks the audio device while another one also needs to access it. In this case, there are applications which simply seem to hang or even do not appear on the screen instead of giving feedback to the user. Actually, the application simply waits until the audio device gets freed.
In the following example, xmms gets started to play an audio file. After that, Alsa Modular Synth (AMS) gets started, also directly using the hardware device, while xmms still is blocking the device. Unfortunately, AMS does not give any feedback, neither on the command line nor in the graphical user interface:


```
bash-2.05b$ xmms Tower_Of_Integration.mp3 &
[1] 9375
bash-2.05b$ ams --soundcard hw:0,0
LADSPA_PATH: /usr/lib/ladspa
loadPath: /usr/share/doc/, savePath: /home/ce/
```

The user will only notice that AMS does not start. As soon xmms will be quit, even if it were hours later, AMS will suddenly appear on the screen. Some feedback on the command line and a graphical alert message helped the user to understand, to solve and to avoid this situation in the future.

Concerning JACK clients, it is an integral part of the philosophy that applications can be disconnected by jackd. As soon as this happens, some applications simply freeze, need to be killed and restarted.
MusE shows an informational dialog instead of simply freezing:


Jack has detected a performance problem which has lead to MusE being disconnected.
This could happen due to a number of reasons:
- a performance issue with your particular setup.
- a bug in MusE (or possibly in another connected software).
- a random hiccup which might never occur again.
If the problem persists you are much welcome to discuss your problem on the MusE mailinglist.
(there is information about joining the mailinglist on the MusE homepage, which is available through the help menu)

To proceed from here you need to save your work and restart MusE, after which the problem will hopefully have vanished.

Of course it would be better if jackified applications would not need to be restarted and could try to automatically reconnect to JACK without any user interaction as soon as a disconnect occures.

2.6 Straightforwardness

To perform a task, the user has to keep several things in mind, may it be a melody, a drum pattern etc. There are external influences like a ringing telephone or colleagues needing some attention. So, the user only spends a reduced amount of his attention to the computer. This is why applications need to behave as straightforward as possible.

One of the goals is that the computer remembers as many things as possible, including commands, settings etc. On Linux, the advantages of the command line interface are well known, but it is also known how badly commands and options are remembered if these are not used or needed often. This is why typed commands are replaced by menus and buttons in end user software whenever possible.

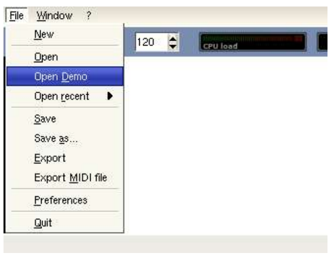
Users dislike reading manuals or dialog texts and developers dislike writing documentation. Keeping the interface and dialogs straightforward will reduce the need of documentation. It is also true that it is difficult to keep documentation in sync with the software releases.

It is also important to create a tidy user interface by aligning objects properly, grouping them together and giving equal objects the same size.

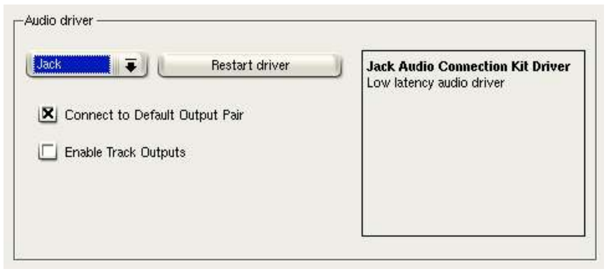
Disabling elements which do not fit the current situation helps the user to find the right tools at the right time. Ordering UI elements to fit the user's workflow reduces to write or to read documentation. Think about analogue synthesizers: Mostly, the elements are ordered to fit the signal flow from the oscillators through the mixer and filter sections to the outputs.

According to Dave Phillips, who asked the developers to write documentation, an additional thing is to reduce the amount of documentation needed. The best documentation is the one which does not need to be written and the best dialogs are the ones which do not need to appear. Both will reduce the time of project members spent on writing documentation and designing dialogs. The user will benefit as well as the project members.

After invoking an application, the user likes to start immediately to concentrate on the task to perform. He might expect a reasonable preconfigured application he immediately can start to use. When starting, Hydrogen opens a default template file. Further demo files are included, easily accessible via the file menu:



Hydrogen optionally remembers the last opened file to restore it during the next start and automatically reconnects to the last used JACK ports:

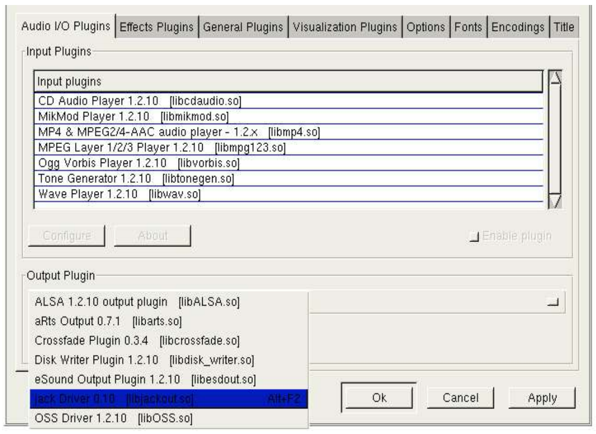


Jack Audio Connection Kit Driver
Low latency audio driver

[✓ Connect to Default Output Pair]
[ ] Enable Track Outputs

Alsa Modular Synth includes a lot of example files, too, but it does not load a template file when starting, and there is no easy access to the example files. The user needs to know that there are example files, and the user needs to know where these files are stored.

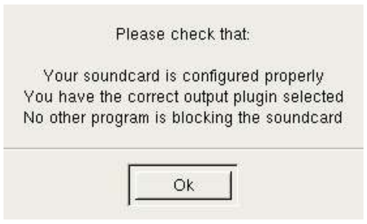
Another example is the fact that there are different audio subsystems on a Linux box. An audio application which wants to work straightforward included various output plugins. One application which already has a surprising amount of output plugins (including a JACK plugin) is xmms:



Input plugins
CD Audio Player 1.2.10 [libcdaudio.so]
MikMod Player 1.2.10 [libmikmod.so]
MP4 & MPEG2/4-AAC audio player - 1.2.x [libmp4.so]
MPEG Layer 1/2/3 Player 1.2.10 [libmpg123.so]
Ogg Vorbis Player 1.2.10 [libvorbis.so]
Tone Generator 1.2.10 [libtonegen.so]
Wave Player 1.2.10 [libwav.so]

ALSA 1.2.10 output plugin [libALSA.so]
aRts Output 0.7.1 [libarts.so]
Crossfade Plugin 0.3.4 [libcrossfade.so]
Disk Writer Plugin 1.2.10 [libdisk_writer.so]
eSound Output Plugin 1.2.10 [libesdout.so]
Jack Driver 0.10 [libjackout.so] Alt+F2
OSS Driver 1.2.10 [libOSS.so]

Furthermore, xmms offers feedback as soon as the configured output destination is not available during startup:


Please check that:

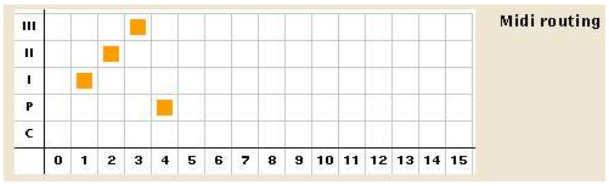
Your soundcard is configured properly
You have the correct output plugin selected
No other program is blocking the soundcard

A further idea to make it even more straightforward could be that xmms did some checks as soon as the preconfigured device is not available during startup and chooses an alternative plugin automatically. If doing these checks in an intelligent order (maybe JACK, esound, aRts, DMIX, ALSA, OSS), xmms will most probably match the user's expectations. If no audio system was found, an error message got printed.

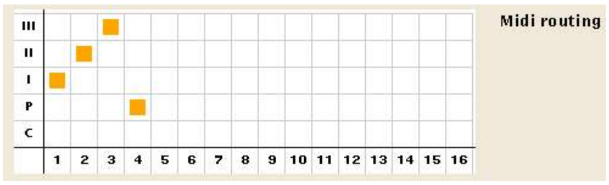
Introducing such a behaviour could improve other Linux audio applications, too. Some example code in the ALSA Wiki pages[8] could be a good starting point for future audio developers.

Average human beings tend to start counting at 1. In the software world, counting sometimes starts at zero for technical reasons. Software makes it possible to hide this in the user interface. This includes the numbering of sound cards (1, 2, 3 instead of 0, 1, 2) as well as program changes (1, 2, 3 ... 128 instead of 0, 1, 2 ... 127) or MIDI channels (1, 2, 3 ... 16 instead of 0, 1, 2 ... 15).

A musician configured his keyboards to send notes on the channels 1 through 4. If the software starts the numbering at zero, the user has to struggle with the setup:



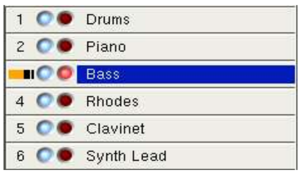
A more human readable numbering matched the user's expectations much more:



2.7 User Control

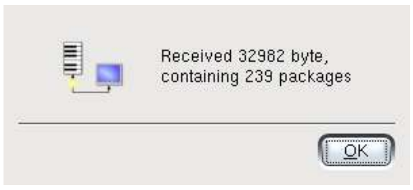
Every human wants to control his environment. Using software, this includes the ability to cancel long lasting operations as well as the possibility to configure the system, like preferred working directories and the preferred audio subsystem. An application that behaves the way the user expects makes him feel that he is controlling the environment. It also needs to balance contrary things, allowing the user to configure the system while preventing him from doing risky things.

As soon as a musician gets no audio in a real studio, he starts searching where the audio stream is broken. In a software based system, he also needs controls to do so. Such controls are not only needed for audio but also for MIDI connections. An LED in a MIDI sequencer indicating incoming data or a levelmeter in an audio recording program to indicate audio input are very helpful to debug a setup. On some hardware synthesizers corresponding controls exist. A Waldorf Microwave, for example, uses an LED to indicate that it is receiving MIDI data. Rosegarden displays a small bar clearly showing that there is incoming MIDI data on the corresponding track:



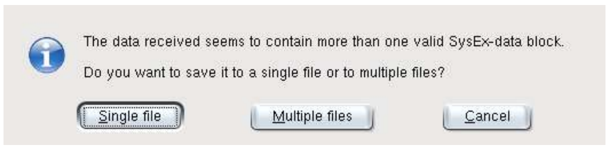
1	Drums
2	Piano
	Bass
4	Rhodes
5	Clavinet
6	Synth Lead

A further thing worth some attention is not to urge the user to make decisions. SysExxer is a small application to send and receive MIDI system exclusive data (also known as sysex). After SysExxer has received sysex data the user must decide that the transmission is finished by clicking the OK button:



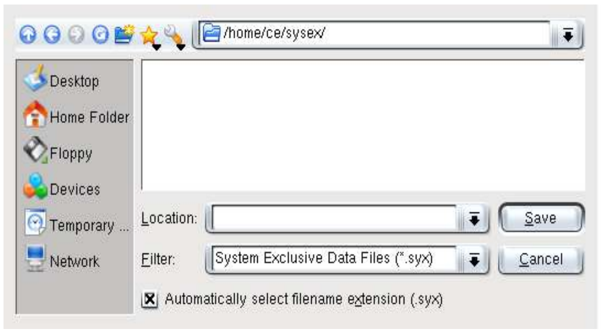
Received 32982 byte,
containing 239 packages

After that the user has to decide if the data gets saved as one single or as multiple split files:



The data received seems to contain more than one valid SysEx-data block.
Do you want to save it to a single file or to multiple files?

SysExxer then asks for a location to store the file:



/home/ce/sysex/

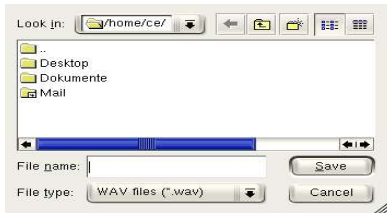
Location:

Filter: System Exclusive Data Files (*.syx)

[✓] Automatically select filename extension (.syx)

SysExxer urged the user to make three consecutive decisions and therefore the user does not feel like he is controlling the situation. Instead, the application controls the user.

Some Qt based audio programs often forget the last used document path. The user opens a file somewhere on the disk. As soon he wants to open another one, the application has forgotten the path the user has used just before and jumps back to the home directory:



Look in: /home/ce/

..
Desktop
Dokumente
Mail

File name:

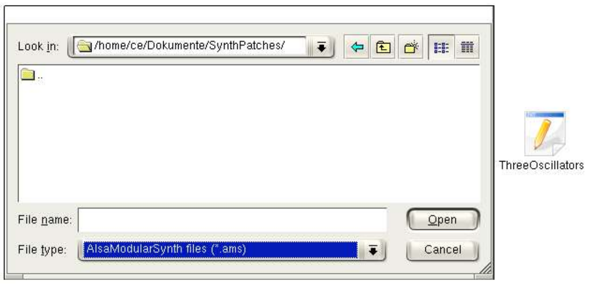
File type: WAV files (*.wav)

Most probably the user did expect to be put back to the last used directory. Applications can offer preferences for file open and file save paths or even better persistently remember the last used paths for the user.

The Linux operating system does not depend on file extensions. On the other hand, file extensions are widely used to assign a filetype to applications. If an application has chosen to use a certain file extension, it should ensure that the extension gets appended to saved files even if the user forgot to specify it.

A file open dialog can filter the directory contents so the user only gets prompted with files matching the application. On the other hand, a file open dialog also should make it possible to switch this filter off, so the user is able to open a file even if it is missing the correct extension.

If an application does not ensure that the extension gets appended when saving a file and does not enable the user to switch the filter off when trying to reopen the file, it will not appear in the file open dialog even if it resides in the chosen directory, so the user is unable to open the desired file:


Look in:
/home/ce/Dokumente/SynthPatches/
ThreeOscillators
File name:
Open
File type:
AlsaModularSynth files (*.ams)
Cancel

On Linux configuration options are usually stored in configuration files. A user normally doesn't care about configuration files because settings are done using GUI elements. On the other hand, sometimes things go wrong. Maybe a crashing application will trash its own configuration file or an update has problems reading the old one. Therefore, command line options to ask for the currently used configuration and data files are sometimes very helpful. The same information can be displayed on a tab in the about box. This enables the user to modify it if needed.

User control includes to enable the user to configure the base audio environment. Maybe he wants to rename soundcards and ports according to more realistic device names like »Onboard Soundcard Mic In« or similar. If a MIDI device which offers multiple MIDI ports is connected to the computer, it is useful to be able to name the ports according to the instruments connected to the ports. If there is more than one soundcard connected, the user may like to reorder them in a certain manner, perhaps to make the new USB 5.1 card the default device instead of the onboard soundchip. He wants to grant more than one application access to a soundcard, in order to listen to ogg files while a software telephone still is able to ring the bell.

In the last few years, alsaconf has done a really great job, but meanwhile it seems to be a little bit outdated because it is unable to configure more than one soundcard or to read an existing configuration. It is unable to configure the DMIX plugin or to put the cards in a certain user defined order. It still configures ISA but no USB cards.

A replacement seems to be a very desirable thing. Such a script, used by configuration front ends, will bring many of the existing but mostly unused features to more users.

A further script could be created to help configuring JACK, maybe by doing some tests based on the hardware used and automatically creating a base JACK configuration file.

2.8 Forgiveness

A software system which allows to easily undo more than one of the last actions performed will encourage the user to explore it. The classical undo and redo commands are even found on hardware synthesizers like the Access Virus[9].

Some applications allow to take snapshots so the user can easily restore a previous state of the system, for example on audio mixing consoles.

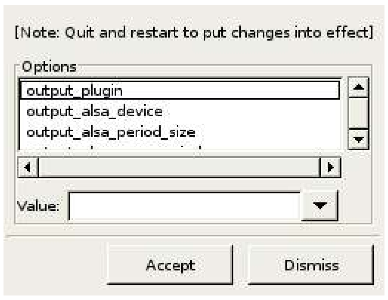
As soon the user wants to do an irreversible action an alert should ask for confirmation.

2.9 Direct Access

In graphical user interfaces tasks are performed by manipulating graphically represented objects. It is a usability design goal to make accessing the needed commands as easy as possible.

Options an application supports should be made dynamically accessible, even during runtime. AMS for example supports a monophonic as well as a polyphonic mode via a command line option. Unfortunately, it is not possible to enter the desired polyphony persistently via the preferences or to change the polyphony during runtime. As soon as the user forgets to pass the desired polyphony to AMS during startup, it needs to be quit and restarted with the correct polyphony applied. Then the user has to reopen the patch file and to redo all MIDI and audio connections. Therefore there is no direct access to the polyphony settings.

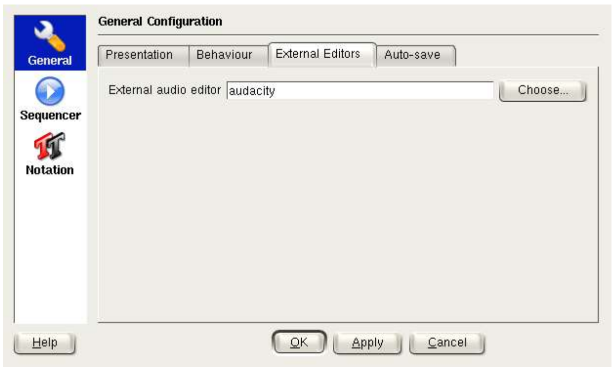
When a user changes the preference settings of an application, the program should change its behaviour immediately. It is also important to write the preferences file immediately after changes have been made. Otherwise, a crash will make the application forget the settings the user has made. In gAlan for example, preference changes make it necessary to restart the program:


[Note: Quit and restart to put changes into effect]

Tooltips and »What's this« help are very useful things in GUI applications. Both grant the user direct access to documentation on demand and reduce the need for writing documentation. On the other hand, if an application offers tooltips and

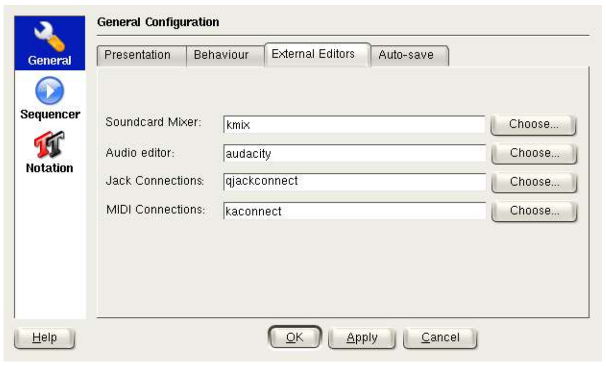
»What's this« help, both need to contain reasonable contents. Otherwise, the user may believe that similar controls will also be useless in other applications.

A further thing to grant users direct access is to give them needed options and controls where and when needed, even if the controls were external applications. A Linux audio user often is in need to access the soundcard's mixer as well as a MIDI or JACK connection tool. Therefore, a sequencer application offering a button to launch these tools from within its user interface grants the user direct access. There are different tools like kaconnect, qjackconnect or qjackctl, so preference settings make it possible that the user enters the programs he likes to use. Rosegarden for example allows the user to enter the preferred audio editor:



External audio editor: audacity

The same way external MIDI and JACK connection tools as well as a mixer could be made available:



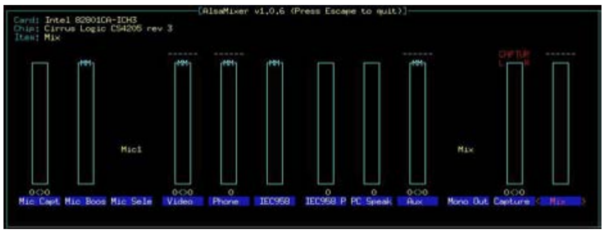
Soundcard Mixer:	kmix
Audio editor:	audacity
Jack Connections:	qjackconnect
MIDI Connections:	kaconnect

These fields need to be well preconfigured during the first application startup. One possibility is to set the application's defaults to reasonable values during development time. Unfortunately, this seems to be impossible because the configuration of the user's system is not known at this moment.

Therefore, Rosegarden could try to check for the existence of some well known tools like qjackctl, qjackconnect, kaconnect, kmix, kamix or qamix and enter them into the matching fields in case these are empty at application startup.

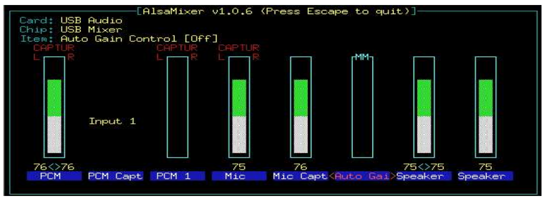
An application could even offer the possibility to make MIDI and audio connections directly from its interface with UI elements of its own. As long as the connections reappear in aconnect and jack_connect, it fulfills both the usability requirements of direct access and consistence.

A further issue concerning direct access is setting a soundcard's mixing controls. On a notebook with an AC '97 chip, alsamixer usually shows all controls the chip has to offer, regardless if all abilities of the chip are accessible from the outside of the computer or not:



```
[AlsaMixer v1.0.6 (Press Escape to quit)]
Card: Intel 82801CA-ICH3
Chip: Cirrus Logic CS4205 rev 3
Item: Mix
```

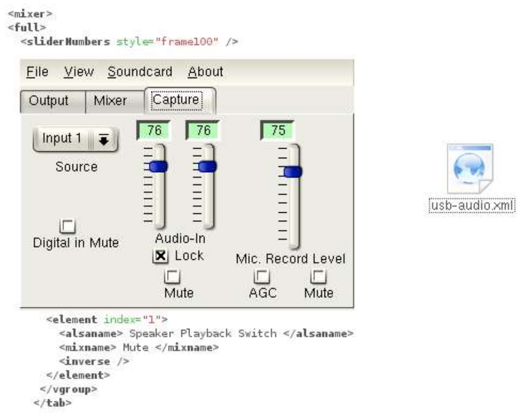
Of course ALSA cannot know which pins of the chip are connected to the chassis of the computer. The snd-usb-module really handles a huge amount of more and more hotpluggable devices. This simply makes it impossible for ALSA to offer a reasonable interface for each particular card. Currently, alsamixer for a Terratec Aureon 5.1 USB looks like this:



```
[AlsaMixer v1.0.6 (Press Escape to quit)]
Card: USB Audio
Chip: USB Mixer
Item: Auto Gain Control [Off]
```

PCM1 does not seem to do anything useful, while auto gain is not a switch (like the user might expect) but looks like a fader instead. It cannot be faded, of course, but muted or unmuted to switch it on and off. There are two controls called 'Speaker'. One of them controls the card's direct throughput, while the second one controls the main volume. The average user has no chance to understand this mixer until he plays around with the controls and checks what results are caused by different settings.

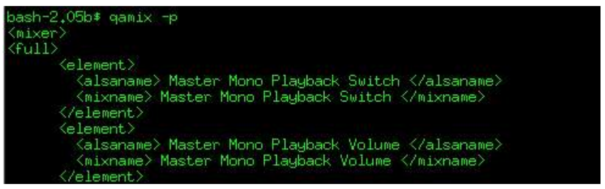
Qamix tries to solve this building the UI reading an XML file matching the card's name:



```
<mixer>
<full>
  <sliderNumbers style="frame100" />
```

```
      <element index="1">
        <alsaname> Speaker Playback Switch </alsaname>
        <mixname> Mute </mixname>
        <inverse />
      </element>
    </vgroup>
  </tab>
```

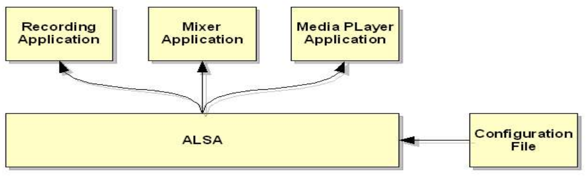
The user configures a card by adjusting this file. This is a nice attempt, but still qamix gets too less information from ALSA to make a really good UI:



```
bash-2.05b$ qamix -p
<mixer>
<full>
      <element>
        <alsaname> Master Mono Playback Switch </alsaname>
        <mixname> Master Mono Playback Switch </mixname>
      </element>
      <element>
        <alsaname> Master Mono Playback Volume </alsaname>
        <mixname> Master Mono Playback Volume </mixname>
      </element>
```

A further usability goal is consistency. All these points mentioned above require that ALSA needs to remain the central and master instance for all user applications.

One idea to realize this is to make ALSA able to give applications more and better information about the hardware. As soon a soundcard gets configured, a matching, human created configuration file got selected manually or via a card ID, so the controls of a particular card could be named more human readable. If the configuration file was in Unicode format, it even can contain names in different languages:



Introducing a similar system for ALSA device and MIDI port names also seems to be desirable. This needs writing some configuration files. Keeping the file format as simple unicode text or XML files which simply are stored at a certain loaction can make users easily contribute.

3. The Developer's Point of View

After having discussed a lot of usability issues from a user's point of view, it is also necessary to view it from a developer's point of view. There are several reasons why it is difficult to introduce usability into Linux audio software projects.

First of all, the question is if there is any reason why a developer of free software should respect usability ideas. If someone has written an application for his own needs and has given it away as free software, the motivation to apply usability ideas has a minor priority. Nobody demands that an application that someone submitted for free has to include anything.

In the beginning of a software project, it makes less sense to think about the user interface. At first, the program needs to include some useful functionality. As soon as a project grows and maybe several developers are working on it, the user base might increase. In this case, the ambition of the project members to make the application better and more user friendly usually increases.

Even in this case, it is often difficult to achieve a more user friendly design for technical reasons. Applications often are designed to start with command line parameters to make them behave as desired. If the design of the software is made to take options at application startup, it is likely that these cannot easily be changed during runtime. The same is valid for preference settings read from a configuration file during an application's startup. It often does not expect the values to be changed dynamically. Adjusting this behaviour at a later point in the development process is sometimes difficult and can cause consecutive bugs.

Keeping this in mind, it requires to design the software system to keep the different values as variable as possible from the very beginning of the development process. Changing this afterwards can be difficult.

A further point is the fact that developers tend to be technically interested. A developer might have been working on new features for a long time, having done research on natural phenomena and having created some well designed algorithms to simulate the result of the research in software. The developer now is proud having created a great piece of software and simply lacks the time to make this valuable work more accessible to the users. The developer has put a lot of time and knowledge in the software and then unfortunately limits the user base by saving time while creating the user interface.

Sometimes it is also caused by the lack of interest about user interface design. There have always been people who are more interested in backends and others more interested in frontends. Developers who are interested in both cannot be found often. It is important to notice that this is a given fact. But not each project has the luck that one of the members is an usability expert, so it is useful if all project members keep usability aspects in mind.

Of course, bugs need to be fixed, and even the existing user base tends to beg more for new features instead of asking to improve the existing user interface. This is understandable because the existing user base already knows how to use the program.

A software system will never be finished. So developers who want to introduce usability improvements need to clearly decide if they want to spent some time on it at a certain point.

Working on usability improvements needs time, and time is a strongly limited resource especially in free software projects which mainly are based on the work of volunteers.

Everything depends on the project members and if they are interested in spending some time on usability issues or not.

4. Summary

Linux is surely ready for the audio desktop. Most applications a musician needs are available, at least including the base functionality. Furthermore, there is free audio software which does things musicians have never heard about on other operating systems.

Due to this fact, keeping some usability ideas in mind will make more semi-skilled users enjoying free software. More users cause more acceptance of free software, and this will cause more people to participate.

Usability is not limited to graphical user interfaces. It also affects command line interfaces. Linux is known to be a properly designed operating system. Respecting some basic usability rules helps continuing the tradition.

On Linux there are many choices which environment to work in. There are different window managers and desktop environments as well as different toolkits for graphical user interface design.

Working and developing on a heterogeneous system like Linux does not mean that it is impossible or useless to apply usability rules. It simply means that it is a special challenge to address and work on usability ideas.

Paying attention to usability issues is not only important to make user's life easier or improve his impression. It also is important to broaden the user base in order to get more bug reports and project members. And finally it helps spreading Linux when surprising average computer skilled musicians what cool applications are available as free software.

5. License

6. Resources

- [1] The wxWidgets toolkit wxGuide: http://wxguide.sourceforge.net
- [2] The Apple human interface principles: http://developer.apple.com/documentation/mac/pdf/HIGuidelines.pdf
- [3] The Gnome user interface guidelines: http://developer.gnome.org/projects/gup/hig/
- [4] User Interface Design by Joel Spolsky: http://joelonsoftware.com /navLinks/fog0000000247.html
- [5] The GNU coding standards: http://www.gnu.org/prep/standards
- [6] Patchage: http://www.scs.carleton.ca/~drobilla/patchage/
- [7] Various software of M. Nagorni: http://alsamodular.sourceforge.net/
- [8] The ALSA wiki pages: http://alsa.opensrc.org
- [9] The Access Virus Synthesizers: http://www.virus.info/

Updates of the WONDER software interface for using Wave Field Synthesis

Marije A.J. BAALMAN
Communication Sciences, Technische Universität Berlin
Sekr. EN8, Einsteinufer 17
Berlin, Germany
baalman@kgw.tu-berlin.de

Abstract

WONDER is a software interface for using Wave Field Synthesis for audio spatialisation. Its user group is aimed to be composers of electronic music or sound artists. The program provides a graphical interface as well as the possibility to control it externally using the OpenSoundControl protocol. The paper describes improvements and updates to the program, since last year.

Keywords

Wave Field Synthesis, spatialisation

1 Introduction

Wave Field Synthesis (WFS) is a technique for sound spatialisation, that overcomes the main shortcoming of other spatialisation techniques, as there is a large listening area and no “sweet spot”. In recent years, WFS has become usable with commercially available hardware.

This paper describes the further development of the WONDER program, that was designed to make the WFS-technique available and usable for composers of electronic music and sound artists.

2 Short overview of WFS and WONDER

WFS is based on the principle of Huygens, which states that a wave front can be considered as an infinite number of point sources, that each emit waves; their wavefronts will add up to the next wavefronts. With Wave Field Synthesis, by using a discrete, linear array of loudspeakers, one can synthesize correct wavefronts in the horizontal plane (Berkhout e.a. 1993). See figure 1 for an illustration of the technique.

WONDER is an open source software program to control a WFS system. The program provides a graphical user interface and allows the user to

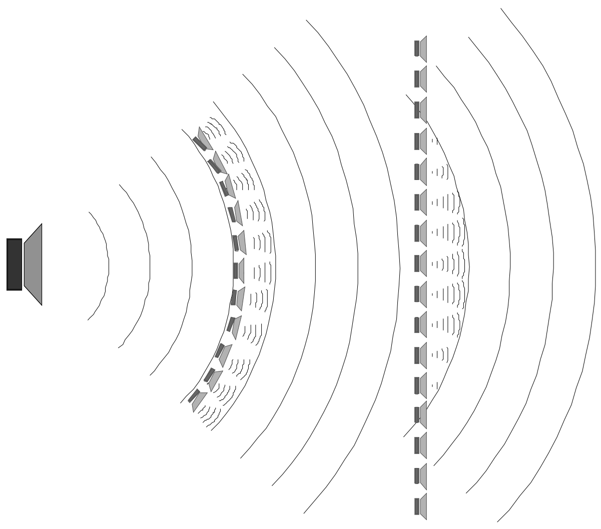


Figure 1. The Huygens' principle (left) and the Wave Field Synthesis principle (right).

think in terms of positions and movements, while the program takes care of the necessary calculations for the speaker driver functions. The program is built up in three parts: a grid definition tool, a composition tool and a play interface. Additional tools allow the user to manipulate grids or scores, or view filter data.

For the actual realtime convolution, WONDER relies on the program BruteFIR (Torger). This program is capable of doing the amount of filter convolutions that are necessary to use WFS in realtime. On the other hand, BruteFIR has the drawback, that all the filters need to be calculated beforehand and during runtime need to be stored in RAM. This limits the flexibility for realtime use. It is for this reason, that a grid of points needs to be defined in advance, so that the filters for these points can be calculated beforehand and stored in memory.

For a more complete description of the WONDER software and the WFS-technique in general, I refer back to previous papers (Baalman, 2003/2004).

3 Updates to WONDER

Since its initial release in July 2004, WONDER has gone through some changes and updates. New tools have been implemented and the functionality of some of the old tools have been improved. Also, some parts were reimplemented to allow for easier extension in the future and resulting in a cleaner and more modular design.

The graphical overview, which displays the spatial layout of the speakers and source positions have been made consistent with each other and now all provide the same functionality, such as the view point (a stage view or an audience view), whether or not to show the room, setting the limits and displaying a background image.

The room definition has been improved: it is now possible to define an absorption factor for each wall, instead of one for all walls.

3.1 Grid tools

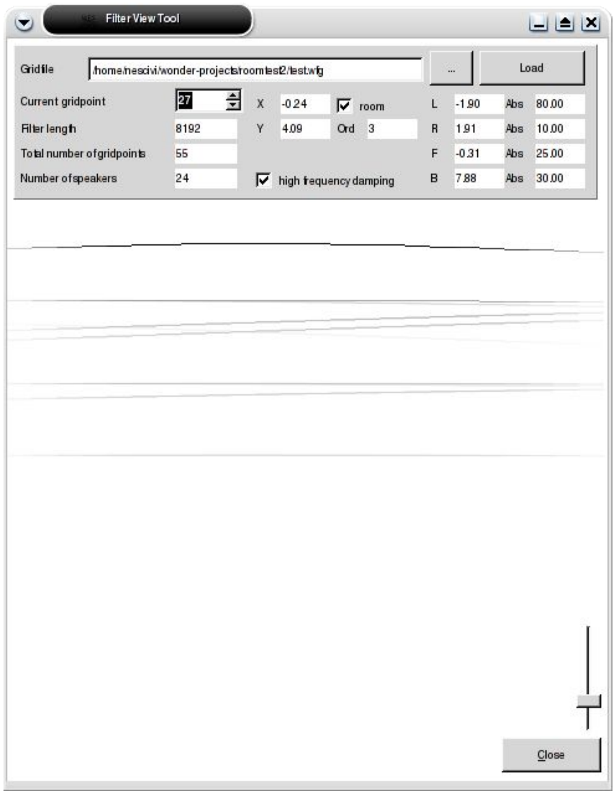
The Grid definition tool allows a user to define a grid consisting of various segments. Each segment can have a different spacing of points within its specified area and different characteristics, such as inclusion of high frequency damping and room parameters for reflections.

The menu “Tools” now provides two extra tools to manipulate Grids. It is possible to merge several grids to one grid and to transform a grid.

The Merge-tool puts the points of several grids into one grid, allowing the user to use more than one grid in an environment.

The Transform-tool applies several spatial transformations to the segments of a grid and then calculates the points resulting from these transformed segments. This transformation can be useful if a piece will be performed on another WFS system, which has another geometry of the speaker setup and the coordinates of the grid need to be transformed.

The filter view (fig. 2) is a way to verify a grid graphically. It shows you the coefficients of all the filters of a source position in a plot. The plot shows on the horizontal axis the loudspeakers, in the vertical direction the time (on the top is zero). The intensity indicates the value of the absolute value of the volume. Above the graph, the input parameters of the gridpoint are given, as well as some parameters of the grid as a whole. This filter overview can be useful for verification of calculations or for eductional purposes.



Gridfile	/home/nescivi/wonder-projects/roomtest2/test.wfg								
Current gridpoint	27	X	-0.24	☑ room		L	-1.90	Abs	80.00
Filter length	8192	Y	4.09	Ord	3	R	1.91	Abs	10.00
Total number of gridpoints	55					F	-0.31	Abs	25.00
Number of speakers	24	☑ high frequency damping				B	7.88	Abs	30.00

Figure 2. Filter view tool of WONDER. At the top, information about the grid and the current grid point are given. In the plot itself is in the horizontal direction the speakers, in the vertical direction the time. The intensity (contrast can be changed with the slider at the bottom right) is an indication of the strength of the pulse. The view clearly shows the reflection pattern of the impulse response.

Another way to verify a grid is using the grid test mode during playback with which you can step through the points of a grid and listen to each point separately.

3.2 Composition tools

With the composition tool the user can define a spatial composition of the sound source movements. For each source the movement can be divided in sections in time and the spatial parameters can be given segmentwise per section.

In the composition definition dialog it is also possible to transform the current composition. The user can define a set of spatial transformations that have to be applied to the sources and sections specified. After the transformations have been applied, the user can continue working on the composition. This tool is especially handy when one source has to make a similar movement as another: just make a copy of the one source and apply a transformation to the copy.

After a score has been created (either with the composition tool or by recording a score), there are four tools available in the "Tools"-menu to manipulate the scores.

"Clean score" is handy for a recorded score. This cleans up any double time information and rounds the times to the minimum time step (default: 25 ms).

"Merge scores" enables you to merge different scores into one. It allows a remapping of sources per score included.

"Transform score" allows you to make transformations to different sources in a score.

The last tool is the "timeline view", which shows the x- and y-component in a timeline (fig. 4); it shows the selected time section as a path in an x-y view. It is also possible to manipulate time points in this view. While playing it shows a playhead to indicate the current time. The concept of this timeline view is inspired by the program "Meloncillo" (Rutz). The timeline view allows for a different way of working on a composition: the user can directly manipulate the score.

3.3 Play interface

WONDER provides a graphical interface to move sound sources or to view the movement of the sound sources. The movement of sound sources can be controlled externally with the OSC-protocol (Wright, 2003).

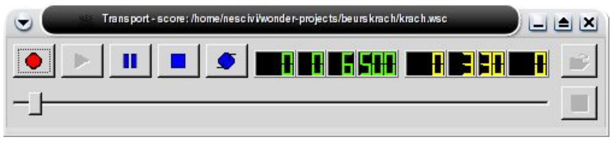

Transport - score: /home/nescivi/wonder-projects/beurskrach/krach.wsc

Figure 3. Transport control of WONDER. The slider enables the user to jump to a new time. The time in green (left) indicates the running time, the time in yellow (right) the total duration of the score. The caption of the window indicates the name of the score file.

Score control is possible by using the Transport controls (fig. 3), which have been re-implemented.

The sound in- and output can be chosen to be OSS, ALSA, JACK or a sound file. In the first three cases the input has to be equal to the output.

The program BruteFIR (Torger) is used as audio engine. The communication between BruteFIR and WONDER can be verified by using a logview, which displays the output of BruteFIR. Due to some changes in the command line interface of BruteFIR, the communication between WONDER and BruteFIR could be improved and there is no more a problem with writing and reading permissions for the local socket.

The logview, which records (in red) the messages that are shown in the statusbar of WONDER, shows the feedback from the play engine BruteFIR in black. It is possible to save the log to a file.

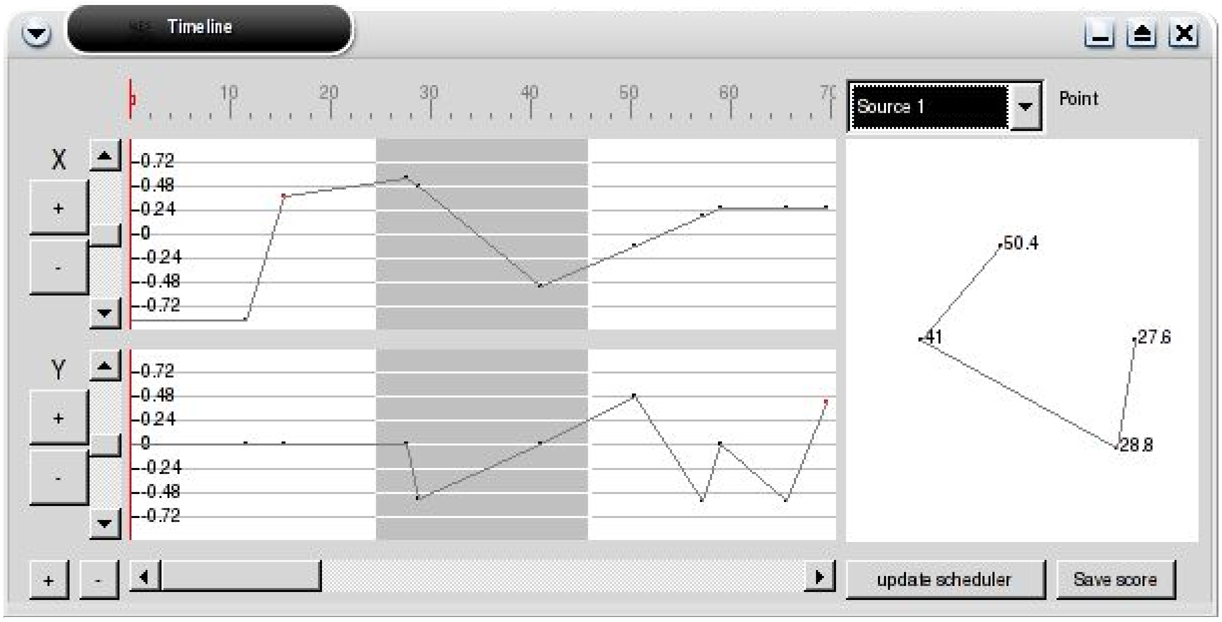


Figure 4. The timeline view of WONDER. The selected time section is shown as a path in the positional overview. The user can edit the score by moving the breakpoints or adding new ones.

3.4 Help functions

The manual of the program is accessible via the “Contents” item in the “Help”-menu. This displays a simple HTML-browser with which you can browse through the documentation. Alternately, you can use your own favourite browser to view the manual.

Additionally, in most views of WONDER, hovering above buttons, gives you a short explanation about the buttons' functionality.

4 External programs to control WONDER

There are two examples available which show how to control WONDER from another program: a SuperCollider Class available and an example MAX/MSP patch.

Another program that can be used for external control is the JAVA-program SpaceJockey (Kneppers), which was designed to enable the use of (customizable) movement patterns and to provide MIDI-control over the movements.

5 Conclusion

WONDER has improved in the last year and has become more stable and more usable. Several changes have been made to facilitate further development of the program.

Current work is to create an interface to EASE for more complex room simulation and to integrate the use of SuperCollider as an optional engine for the spatialisation. It is expected that SuperCollider can provide more flexibility, such as removing the necessity to load all filter files in RAM and the possibility to calculate filter coefficients during runtime.

Current research is focused on how to implement a more complex sound source definition.

Download

A download is available at:
http://gigant.kgw.tu-berlin.de/~baalman/

References

Baalman, M.A.J., 2003, *Application of Wave Field Synthesis in the composition of electronic music*, International Computer Music Conference 2003, Singapore

Baalman, M.A.J., 2004, *Application of Wave Field Synthesis in electronic music and sound installations*, Linux Audio Conference 2004, ZKM, Karlsruhe, Germany

Baalman, M.A.J. & Plewe, D., 2004, *WONDER - a software interface for the application of Wave Field Synthesis in electronic music and interactive sound installations*, International Computer Music Conference 2004, Miami, Fl., USA

Berkhout, A.J., Vries, D. de & Vogel, P. 1993, *Acoustic Control by Wave Field Synthesis*, Journal of the Acoustical Society of America, 93(5):2764-2778

Kneppers, M., & Graaff, B. van der, SpaceJockey, http://avdl1064.oli.tudelft.nl/WFS/

Rutz, H.H., Meloncillo, http://www.sciss.de/meloncillo/index.html

Torger, A., BruteFIR, http://www.ludd.luth.se/~torger/brutefir.html

Wright, M., Freed, A. & Momeni, A. 2003, “OpenSoundControl: State of the Art 2003”, *2003 International Conference on New Interfaces for Musical Expression*, McGill University, Montreal, Canada 22-24 May 2003, Proceedings, pp. 153-160

Development of a Composer's Sketchbook

Georg BÖNN
School of Electronics
University of Glamorgan
Pontypridd CF37 1DL
Wales, UK
gboenn@glam.ac.uk

Abstract

The goal of this paper is to present the development of an open source and cross-platform application written in C++, which serves as a sketchbook for composers. It describes how to make use of music analysis and object-oriented programming in order to model personal composition techniques. The first aim was to model main parts of my composition techniques for future projects in computer and instrumental music. Then I wanted to investigate them and to develop them towards their full potential.

Keywords

Music Analysis, Algorithmic Composition, Fractals, Notation, Object-Oriented Design

0 Introduction

Computer-Assisted Composition (CAC) plays an important role in computer music production. Many composers nowadays use CAC applications that are able to support and enhance their work. Applications for CAC help composers to manage the manifold of musical ideas, symbolic representations and musical structures that build the basis of their creative work. Maybe it is time to put a flashlight on CAC again and to look at examples where user-friendly interfaces meet efficient computation and interesting musical concepts.

What are the advantages of CAC? For a composer to be able to use the PC as an intelligent assistant that represents a new kind of sketchbook and a well of ideas.

Not only, that it is possible to quickly input and save musical data, the work in CAC results in a great freedom of choice between possible solutions of a given compositional problem. Maybe a network of different algorithms work together, then only little changes of initial parameters could trigger surprising twists within the final result. Moreover, you can take those outcomes and scrutinise their value by direct and immediate comparison. Thus, a composer can take the time and always look for alternatives. The manifold of structures and ideas is going to be manageable through CAC software. Also, one should take into account the thrill of surprise that well designed CAC algorithms might fuel into one's work-flow.

This paper wants to discuss one specific compositional problem that is the invention and modelling of melodic structures. The proposed software that resolves that particular probelm shall represent a germ for an open source (under the GNU Public License) and free CAC application that is planed to grow as the number of compositional algorithms will hopefully increase in the near future. It is intended that this software is easy to learn and to use and that it benefits from proven concepts of object-oriented design and programming. Therefore, the author hopes that the ideas presented will find the interest and also maybe the support of the Linux Audio Developer's community.

Of course, personal preferences and musical experience influence the work of a composer. Those together with intuition, imagination, phantasy and the joy to play, including the joy for intellectual challenges, they are, in my view, the driving forces behind musical creativity. Would it be possible to define a set of algorithms that would match that experience? Is finding an algorithm not also often a creative process?

1 Music Analysis

The fundamental idea in Composer's Sketchbook is the use of a user-defined database, that contains the building blocks for organic free-tonal musical stuctures. Those building blocks are three-note cells which stem from my personal examination of the major and minor third within their tonal contexts (harmonics, tuning systems, Schenker's Ursatz), as well as within atonal contexts (Schoenberg's 6 kleine Klavierstücke op.19, Ives' 4th violin sonata). Although in my system, tonality is hardly recognisable anymore because of constant modulations, i cannot deny the historic dimension of the third, nor can i neglect or avoid its tonal connotations. I suppose it helped me to create an equilibrium of free tonality where short zones of tonality balance out a constant stream of modulations.

Analysis of scores by Arnold Schoenberg and Charles Ives led me to a very special matrix of three-note cells. Further analysis revealed that it is possible to use a simple logic to concatenate those cells. Algorithms using this logic were created who are able to render an unprecedent variety of results based on a small table of basic musical material. In order to generate the table, a small set of generative rules is applied to a set of primordial note-cells. The general rule followed in this procedure is to start with very simple material, then apply combinatorics unfolding its inner complexity and finally generate a manifold of musical variations by using a combination of fractal and stochastic algorithms.

The first four cells forming the matrix are variations of the simple melodic figure e-d-c. The chromatic variations are e-flat-d-c, e-flat-d-flat-c and e-d-flat-c (see Figure 2, A-D).

One of the reasons why I chose these figures was, because they represent primordial, archetypical melodic material that can be found everywhere in music of all periods and ages. An exceptional example for the use of the Ursatz-melody is Charles Ives' slow movement of the 4th violin sonata. This movement quotes the chorale "Yes, Jesus loves me" and uses its three-note ending e-d-c ("he is strong") as a motive in manifold variations throughout the piece. Analysis reveals that Ives uses the major and minor versions of the motive and all its possible permutations (3-2-1, 2-1-3, 1-3-2).

2 The Matrix

The matrix I developed uses exactly the same techniques: Four different modes (A-D) of the 'Ur'-melody are submitted to all three possible permutations (see Figure 2). This process yields 12 primordial cells. Each one of those 12 cells is then submitted to a transformation called "partial inversion".

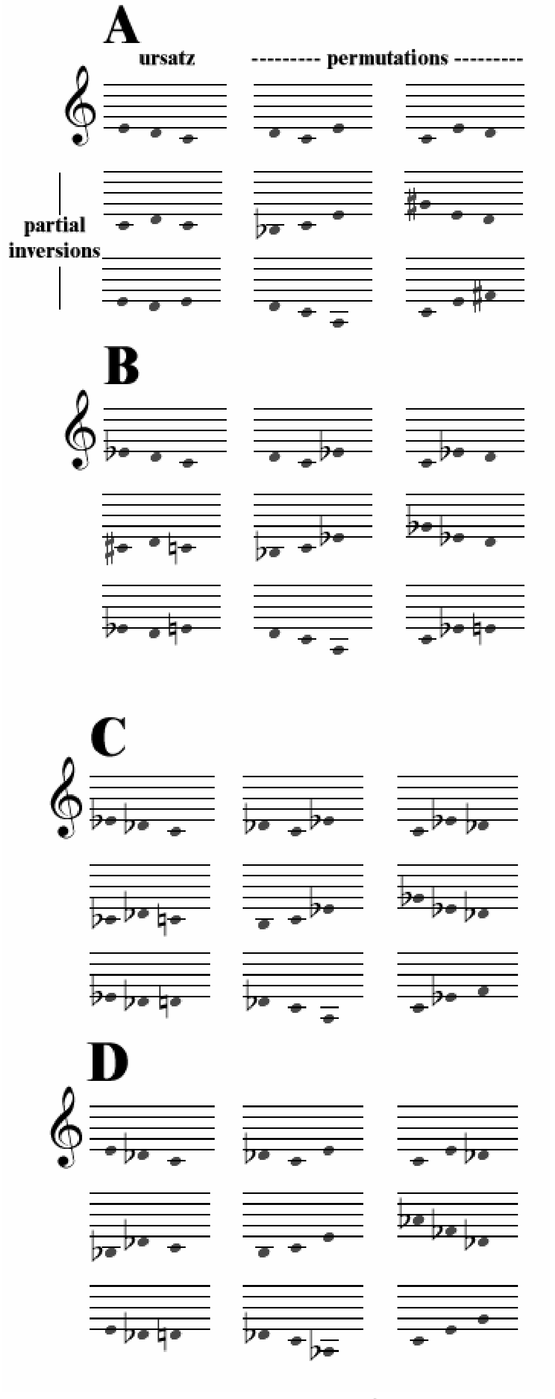


Figure 2: The matrix of 36 cells

Partial Inversion means: Invert the first interval but keep the second one untouched. Or, keep the first interval of the cell original and invert the second one. This process of partial inversion can be found extensively used by Arnold Schönberg in his period of free atonality. As a perfect example, have a look at his Klavierstück op.19/1, bar 6 with up-beat in bar 5.

The reason for using partial inversion is that it produces a greater variety of musical entities than the plain methods of inversion and retrograde. At the same time partial inversion guarantees consistency and coherence within the manifold of musical entities. Applying partial inversion to the 12 cells yields another 24 variants of the original e-d-c-melody.

The final matrix of 36 cells contains an extraordinary complex network of relationships. Almost every cell in the matrix has a partner, which is the very cell in a different form, the cell may be inverted, retrograde, permutated or inverse retrograde. Yet, each one of these is closely related to the original 'Ur'-melody.

Going back to Schönberg's op. 19/1, it came as a surprise that every single note in this piece is part of one or more groups of three notes, which can be identified as a member of the matrix of 36 cells.

The discovery of an element of my own language in the matrix gave me yet another good reason to further investigate its application. I call it "chromatic undermining", breaking into the blank space left behind by a local melodic movement. Cells which belong to that element are the partial inversions of Ursatz B and C, left column.

The matrix of cells forms the basis of all further work with algorithms. It can be regarded as a database of selected primordial music cells. Thus it is clear that a user-interface should make it possible to replace that database by any other set of entities where it is totally in the hands of the user to decide which types of entities should belong to the database. The user-interface should also allow to add, delete or edit the database at runtime and make the database persistent.

3 Algorithms

The algorithms that are used to concatenate cells from the database are as follows:

3.1 Fractal Chaining

Beginning with two notes, the fractal algorithm seeks to insert a new note between them. It detects the original interval between the two notes, then it queries the database whether there exists a cell, which contains that interval between its first and last note. If it is true, then the middle note of the cell is inserted between the two notes of our beginning (see Figure 3 a). We now have a sequence of three notes whose interval strucure is equal to the cell that was chosen from the database. The pitch-classes of our first two notes are preserved. This algorithm, can be applied recusively. Starting now with three notes from our result, the algorithm steps through the intervals and replaces each one of them by two other intervals that were obtained from another cell within our matrix database (see Figure 3 b). Of course, there are multiple solutions for the insertion of a note according to a given interval, because there are several cells within the matrix that would pass the check. The choice made by the program depends on a first-come-first-served basis. In order to make sure that each cell has an equal chance of getting selected, the order of the matrix has always been scrambled the moment before the query was sent to the database. The fractal algorithm needs a minimum input of two notes but it can work on lists of notes of any length. Fractal chains maintain the tendency of the original structure at the beginning. Therefore, this interval structure is the background structure of the final result after the algorithm has been called a few times.

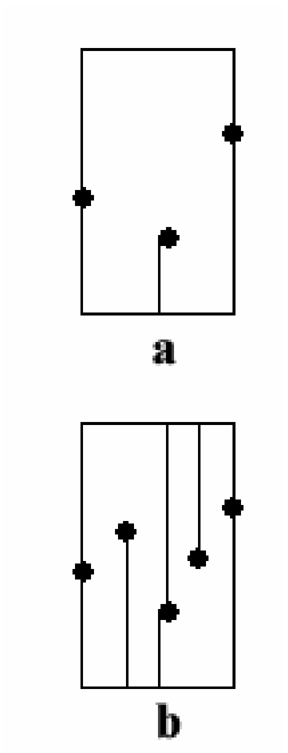


Figure 3: Sequence of Fractal Chaining

3.2 Chain overlapping 2 notes

The chaining algorithms differ from the fractal algorithm because their goal is to add notes to the tail of a given list rather then inserting them between every two notes. The chaining method overlapping two notes looks at the last interval of a melody. It then searches the matrix for a matching three-note cell whose first interval is equal to that last interval of the melody. If a match was found, then the second interval of the cell is added to the melody and so a new note is added, the melody expands (see Figure 4).

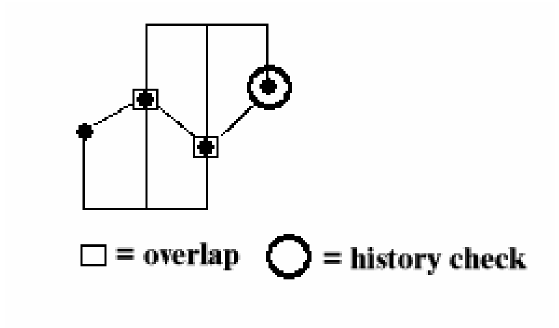


Figure 4: Scheme of Algorithm 3.2

The reason for using overlapping cells was a result of music analysis: One note may belong to more than just one cell, thus providing a high degree of coherence of the inner musical structure.

3.3 Chain overlapping 1 note

This method simply takes a random cell from the database and adds it to the end of the melody by taking its last note as the first note of the cell, thus adding two new notes to the melody (see Figure 5).

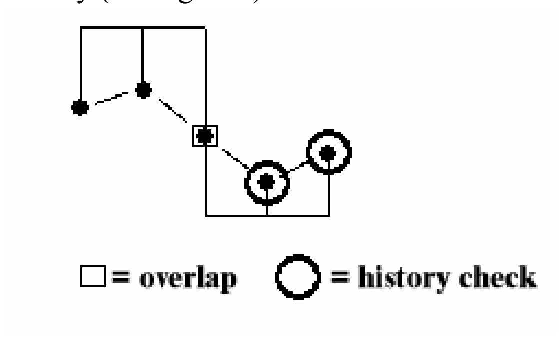


Figure 5: Scheme of Algorithm 3.3

3.4. Chain combining Algorithms 3.2 & 3.3

The algorithm takes the last interval of the melody, finds, if possible, a match with a cell from the database, adds the last note from the cell to the melody, then taking this note as the starting point of a randomly chosen cell from the database (see Figure 6).

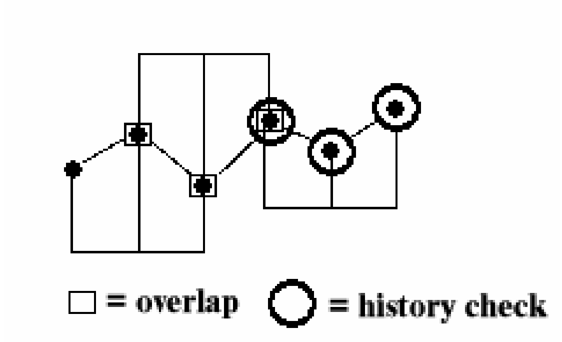


Figure 6: Scheme of Algorithm 3.4

3.5 Option: check history

This option can be switched on or off and triggers a statistical measurment of the pitch-class content of the melody. It ensures that only those cells are chosen from the database whose interval structure generates new pitch-classes when added to the melody. This option leads to the generation of totally chromatic fields and ensures that every pitch-class has an equal chance to occur within the melody. The history-check can be modified in order to meet other requirements, e.g. the algorithm can be told to chose only cells adding pitch-classes to the melody, which belong to a specific key or mode previously defined by the user.

3.6 Chain with no overlap

The algorithm choses the first interval of a randomly chosen cell and adds the interval to the melody. Then it takes another random cell and adds its content to the melody.

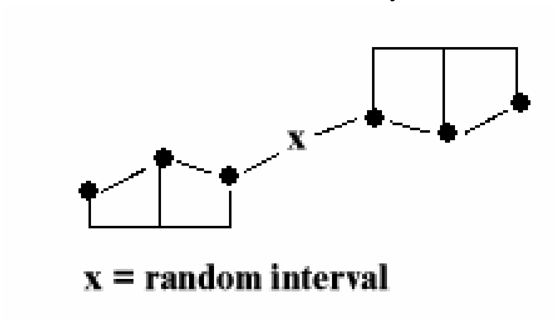


Figure 7: Scheme for Algorithm 3.5

3.7 Chain using different cell contours

The term contour means that each cell has a certain profile of up- or down-movements. These contours are automatically measured within the database. The chaining using contour comparisons takes the last three-note group of the melody and mesures the contour. Then it looks-up the database in order to find a cell which has one of the four possible contours: up/down, down/up, up/up or down/down. The user decides which one of those criteria has to be met and the matching cell is added to the melody.

3.8 Using and extending the program

By using different chaining methods, possibly in a sequence of processes, the user has an enormous freedom of choice and variety in modelling a melodic line and the structure of a sequence.

It will be possible to use within the program an editor for context-free grammar in order to let the user define a set of rules by which musical entites are chosen from the database.

It is easy to imagine, that the matrix of 36 cells can be extended and more variations of the e-d-c-melody could join the matrix. For instance, i added 10 more modes of the 'Ur'-melody resulting in a database of 126 cells.

It is also possible to use the chaining algorithms in order to build-up chord structures. When using a database containig major and minor thirds (or other intervals) it is easy to imagine that the algorithm "Chain overlapping 1 note" with history-check "on" returns all possible chords containing 3 up to 12 notes using nothing but major and minor thirds (or other intervals).

We are also not restricted to the well-tempered 12-note scale. The software is open to more notes per octave as well as it is possible to use micro-intervals.

4 Program Development

The software development uses the C++ language and started as an application for Windows using MFC. In order to port the program to Linux and MacOS X the development switched to the wxWidgets[1] framework.

4.1 User-Interface

The user can input notes via his computer keyboard that automatically mimics the keys of a piano keyboard: 'Q'=c, '2'=c#, 'W'=d, '3'=d#, 'E'=e, 'R'=f, '5'=f# and so on; the lower octave starts with 'Y'=c, 'S'=c# etc.. By using the up- and down-arrow keys the user can switch to the next octave up or down. The notes are being played instantly via MIDI interface and printed directly on screen in western music notation. After a melody has been entered, it is possible to transform the input in a number of ways. For example, rubber-band-style selections can be made, notes can be transposed or deleted just like with a text editor. Series of notes can be inverted and reversed and the aforementioned fractal and chaining algorithms applied. These commands work on both selections or the entire score. A "Refresh" command exists for the fractal algorithms. It automatically reverts the transformation and gives simply "another shot", stepping through all possible alternatives. Of course, selections or the entire score can be played and stopped.

In future versions, all commands described here shall run in their own worker-

[1] Formerly known as wxWindows. The name has been changed to wxWidgets. See also www.wxwindows.org

thread, not in the user-interface thread, so the user-interface will not be blocked by any calculations. The program supports export of standard MIDI files. It is planned for future versions to support MIDI, XML and CSV file import in order to give more choices for the replacement of the cell database.

4.2 Classes

Following the above description of the algorithms and the user-interface, it is evident that we had to implement the classic Model-View-Controller paradigm. Frameworks like MFC or wxWidgets are built around this paradigm, so it makes sense to use them. Since wxWidgets supports all popular platforms it was chosen as framework for my development.

The representaion of music data as objects makes it necessary to design fundamental data structures and base classes. The software uses mainly two different container classes: An Array class, which is used to organise and save the note-cell data of our database. The Arrays save notes as midi-note numbers (ints) and they generate automatically information about the intervals they contain, which is a list of delta values. The whole database is kept as an Array of Arrays in memory. In order to facilitate ordering of the cells and random permutations of the database, pointers to the note-cell Arrays are kept in a Double-Linked List. The Double-Linked List is a template-based class which manages pointers to object instances. The melody imput from the user is also kept in an instance of the class Double-Linked List, where the elements consist of pointers to a note-table, that can be shared by other objects as well. The Algorithms described in this paper work on a copy of the melody note-list. Since the note-list is of type Double-Linked List, only pointers are being copied. The Algorithm class intitalises its own Array of notes, which itself creates a delta-list of intervals. Pointers to the delta-list elements are the stored inside a Double-Linked List object, so the Algorithms can easily work on the interval-list from the user input. This is done because there is a lot of comparison of interval sizes going on. The history-check that was described as an option is implemented as a Visitor of the Algorithm. This object calculates a history table of all pitch-classes that have been used so far. It uses the Double-Linked List containing the intervals from the Algorithm object it is visiting. All Events that are sent to the MIDI interface are also managed by a Linked-List Container.

In order to build the editor for note display and to implement user-interaction, the Composite Design-Pattern will be used. All graphic elements will be either Component or Composite instances. For instance a Staff would be a Composite that contains other Components like Notes, Clefs, Barlines, etc..

5 Conclusion

The use of design-patterns like composite and vistor allows us to achieve a very robust code that is both easy to maintain and to extend. The paper also showed that it is possible to model composition techniques using object-oriented design of musical data. An application has been created that has the flexibility to extend knowledge gained from music analysis and personal experience. The initial goal of creating a sketchbook for composers has been achieved.

6 References

J. Dunsby and A. Whittall. 1988. Music Analysis in Theory and Practice. Faber, London

E. Gamma, R. Helm, R. Johnson and J. Vlissides. 1995. Design Patterns : Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading, MA

Bruno R. Preiss. 1999. Data Structures and Algorithms with Object-Oriented Design Patterns in C++. Wiley, New York

M. Neifer. 2002. Porting MFC applications to Linux. http://www-106.ibm.com/developerworks/library/l-mfc/

SoundPaint – Painting Music

Jürgen Reuter
Karlsruhe
Germany
reuter@ipd.uka.de

Abstract

We present a paradigm for synthesizing electronic music by graphical composing. The problem of mapping colors to sounds is studied in detail from a mathematical as well as a pragmatic point of view. We show how to map colors to sounds in a user-definable, topology preserving manner. We demonstrate the usefulness of our approach on our prototype implementation of a graphical composing tool.

Keywords

electronic music, sound collages, graphical composing, color-to-sound mapping

1 Introduction

Before the advent of electronic music, the western music production process was clearly divided into three stages: Instrument craftsmen designed musical instruments, thereby playing a key role in sound engineering. Composers provided music in notational form. Performers realized the music by applying the notational form on instruments. The diatonic or chromatic scale served as commonly agreed interface between all participants. The separation of the production process into smaller stages clearly has the advantage of reducing the overall complexity of music creation. Having a standard set of instruments also enhances efficiency of composing, since experience from previous compositions can be reused.

The introduction of electro-acoustic instruments widened the spectrum of available instruments and sounds, but in principle did not change the production process. With the introduction of electronic music in the middle of the 20th century however, the process changed fundamentally. Emphasis shifted from note-level composing and harmonics towards sound engineering and creating sound collages. As a result, composers started becoming sound engineers, taking over the instrument crafts men's job. Often, a composition could not be notated with traditional notation, or, even worse, the composition was strongly bound to a very particular technical setup of electronic devices. Consequently, the composer easily became the only person capable of performing the composition, thereby often eliminating the traditional distinction of production stages. At least, new notational concepts were developed to alleviate the problem of notating electronic music.

The introduction of MIDI in the early 80s was in some sense a step back to electro-acoustic, keyed instruments music, since MIDI is based on a chromatic scale and a simple note on/off paradigm. Basically, MIDI supports any instrument that can produce a stream of note on/off events on a chromatic scale, like keyed instruments, wind instruments, and others. Also, it supports many expressive features of non-keyed instruments like vibrato, portamento or breath control. Still, in practice, mostly keyboards with their limited expressive capabilities are used for note entry.

The idea of our work is to break these limitations in expressivity and tonality. With our approach, the composer creates sound collages by visually arranging graphical components to an image, closely following basic principles of graphical notation. While the graphical shapes in the image determine the musical content of the sound collage, the sound itself is controlled by color. Since in our approach the mapping from colors to actual sounds is user-definable for each image, the sound engineering process is independent from the musical content of the collage. Thus, we resurrect the traditional separation of sound engineering and composing. The performance itself is done mechanically by computation, though. Still, the expressive power of graphics is straightly translated into musical expression.

The remainder of this paper is organized as follows: Section 2 gives a short sketch of image-to-audio transformation. To understand the role of colors in a musical environment, Section

3 presents a short survey on the traditional use of color in music history. Next, we present and discuss in detail our approach of mapping colors to sounds (Section 4). Then, we extend our mapping to aspects beyond pure sound creation (Section 5). A prototype implementation of our approach is presented in Section 6. We already gained first experience with our prototype, as described in Section 7. Our generic approach is open to multiple extensions and enhancements, as discussed in Section 8. In Section 9, we compare our approach with recent work in related fields and finally summarize the results of our work (Section 10).

2 Graphical Notation Framework

In order to respect the experience of traditionally trained musicians, our approach tries to stick to traditional notation as far as possible. This means, when interpreting an image as sound collage, the horizontal axis represents time, running from the left edge of the image to the right, while the vertical axis denotes the pitch (frequency) of sounds, with the highest pitch located at the top of the image. The vertical pitch ordinate is exponential with respect to the frequency, such that equidistant pitches result in equidistant musical intervals. Each pixel row represents a (generally changing) sound of a particular frequency. Both axes can be scaled by the user with a positive linear factor. The color of each pixel is used to select a sound. The problem of how to map colors to sounds is discussed later on.

3 Color in Musical Notation History

The use of color in musical notation has a long tradition. We give a short historical survey in order to show the manifold applications of color and provide a sense for the effect of using colors.

Color was perhaps first applied as a purely notational feature by Guido von Arezzo, who invented colored staff lines in the 11th century, using yellow and red colors for the do and fa lines, respectively. During the *Ars Nova* period (14th century), note heads were printed with black and red color to indicate changes between binary and ternary meters(Apel, 1962). While in medieval manuscripts color had been widely applied in complex, colored ornaments, with the new printing techniques rising up in the early 16th century (most notably Petrucci's *Odhecaton* in 1501), extensive use of colors in printed music was hardly feasible or just too expensive and thus became seldom. Mozart wrote a manuscript of his horn concert K495 with colored note heads, serving as a joke to irritate the hornist Leutgeb – a good friend of him(Wiese, 2002). In liturgical music, red color as contrasted to black color remained playing an extraordinary role by marking sections performed by the priest as contrasted to those performed by the community or just as a means of readability (black notes on red staff lines). Slightly more deliberate application of color in music printings emerged in the 20th century with technological advances in printing techniques: The advent of electronic music stimulated the development of graphical notation (cp. e.g. Stockhausen's *Studie II*(Stockhausen, 1956) for the first electronic music to be published(Simeone, 2001)), and Wehinger uses colors in an aural score(Wehinger, 1970) for Ligeti's *Articulation* to differentiate between several classes of sounds. For educational purposes, some authors use colored note heads in introductory courses into musical notation(Neuhäuser et al., 1974). There is even a method for training absolute hearing based on colored notes(Taneda and Taneda, 1993). Only very recently, the use of computer graphics in conjunction with electronic music has led to efforts in formally mapping colors to sounds (for a more detailed discussion, see the Related Work Section 9).

While Wehinger's aural score is one of the very few notational examples of mapping colors to sounds, music researchers started much earlier to study relationships between musical and aural content. Especially with the upcoming psychological research in the late 19th century, the synesthetic relationship between hearing and viewing was studied more extensively. Wellek gives a comprehensive overview over this field of research(Wellek, 1954), including systems of mapping colors to keys and pitches. Painters started trying to embed musical structures into their work (e.g. Klee's *Fugue in Red*). Similarly, composers tried to paint images, as in Mussorgsky's *Pictures at an Exhibition.* In Jazz music, synesthesis is represented by coinciding emotional mood from acoustic and visual stimuli, known as the blue notes in blues music.

4 Mapping Colors to Sounds

We now discuss how colors are mapped to sounds in our approach.

For the remainder of this discussion, we define a *sound* to be a 2π-periodic, continuous function $s : \mathbf{R} \mapsto \mathbf{R}, t \to s(t)$. This definition meets the real-world characteristic of oscillators as the most usual natural generators of sounds and the fact that our ear is trained to recognize periodic signals. Non-periodic natural sources of sounds such as bells are out of scope of this discussion. We assume normalization of the periodic function to 2π periodicity in order to abstract from a particular frequency. According to this definition, the set of all possible sounds – the *sound space* – is represented by the set of all 2π-periodic functions.

Next, we define the *color space* $\mathbf{C}$ following the standard RGB (red, green, blue) model: the set of colors is defined by a three-dimensional real vector space $\mathbf{R}^3$, or, more precisely, a subset thereof: assuming, that the valid range of the red, green and blue color components is $[0.0, 1.0]$, the color space is the subset of $\mathbf{R}^3$ that is defined by the cube with the edges $(0, 0, 0)$, $(1, 0, 0)$, $(0, 1, 0)$, and $(0, 0, 1)$. Note that the color space is not a vector space since it is not closed with respect to addition and multiplication by scalar. However, this is not an issue as long as we do not apply operations that result in vectors outside of the cube. Also note that there are other possibilities to model the color space, such as the HSB (hue, saturation, brightness) model, which we will discuss later.

Ideally, for a useful mapping of colors to sounds, we would like to fulfill the following constraints:

- **Injectivity.** Different colors should map to different sounds in order to utilize the color space as much as possible.
- **Surjectivity.** With a painting, we want to be able to address as many different sounds as possible – ideally, all sounds.
- **Topology preservation.** Most important, similar colors should map to similar sounds. For example, when there is a color gradation in the painting, it should result in a sound gradation. There should be no discontinuity effect in the mapping. Also, we want to avoid noticeable hysteresis effects in order to preserve reproducibility of the mapping across the painting.
- **User-definable mapping.** The actual mapping should be user-definable, as research has shown that there is no general mapping that applies uniquely well to all individual humans.

Unfortunately, there is no mapping between the function space of 2π-periodic functions and $\mathbf{R}^3$ that fulfills all of the three constraints. Pragmatically, we drop surjectivity in order to find a mapping that fulfills the other constraints. Indeed, dropping the surjectivity constraint does not hurt too much, if we assume that the mapping is user-definable individually for each painting and that a *single* painting does not need to address *all* possible sounds: rather than mapping colors to the full sound space, we let the user select a three-dimensional subspace $\mathbf{S}$ of the full sound space. This approach also leverages the limitation of our mapping not being surjective: since for each painting, a different sound subspace can be defined by the composer, effectively, the whole space of sounds is still addressable, thus retaining surjectivity in a limited sense.

Dropping the surjectivity constraint, we now focus on finding a proper mapping from color space to a three-dimensional subset of the sound space. Since we do not want to bother the composer with mathematics, we just require the basis of a three-dimensional sound space to be defined. This can be achieved by the user simply defining three different sounds, that span a three-dimensional sound space. Given the three-dimensional color space $\mathbf{C}$ and a three-dimensional subspace $\mathbf{S}$ of the full sound space, a bijective, topology preserving mapping can be easily achieved by a linear mapping via a matrix multiplication,

$$M : \mathbf{C} \mapsto \mathbf{S}, x \to y = Ax, x \in \mathbf{C}, y \in \mathbf{S} \qquad (1)$$

with A being a 3×3 matrix specifying the actual mapping. In practice, the composer would not need to specify this vector space homomorphism M by explicitly entering some matrix A. Rather, given the three basis vectors of the color space $\mathbf{C}$, i.e. the colors red, green, and blue, the composer just defines a sound individually for each of these three basis colors. Since each other color can be expressed as a linear combination of the three basis colors, the scalars of this linear combination can be used to linearly combine the three basis sounds that the user has defined.

5 Generalizing the Mapping

As excitingly this approach may sound at first, as disillusioning we are thrown back to reality: pure linear combination of sounds results in nothing else but cross-fading waveforms, which quickly turns out to be too limited for serious composing. However, what we can still do is to extend the linear combination of sounds onto further parameters that influence the sound in a non-linear manner. Most notably, we can apply non-linear features on sounds such as vibrato, noise content, resonance, reverb, echo, hall, detune, disharmonic content, and others. Still, also linear aspects as panning or frequency-dependent filtering may improve the overall capabilities of the color-to-sound mapping. In general, any scalar parameter, that represents some operation which is applicable on arbitrary sounds, can be used for adding new capabilities. Of course, with respect to our topology preservation constraint, all such parameters should respect continuity of their effect, i.e. there should no remarkable discontinuity arise when slowly changing such a parameter.

Again, we do not want to burden the composer with explicitly defining a mapping function. Instead, we extend the possibilities of defining the three basis sounds by adding scalar parameters, e.g. in a graphical user interface by providing sliders in a widget for sound definition.

So far, we assumed colors red, green and blue to serve as basis vectors for our color space. More generally, one could allow to accept any three colors, as long as they form a basis of the color space. Changing the basis of the color space can be compensated by adding a basis change matrix to our mapping M:

$$M' : \mathbf{C'} \mapsto \mathbf{S}, x \to y = A\phi_{C' \to C}x = A'x, \quad (2)$$

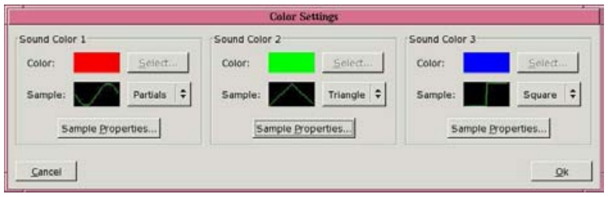
assuming that $\phi_{C' \to C}$ is the basis change matrix that converts x from space $\mathbf{C'}$ to space $\mathbf{C}$.

Specifically, composers may want to prefer the HSB model over the RGB model: traditionally, music is notated with black or colored notes on white paper. An empty, white paper is therefore naturally associated with silence, while a sheet of paper heavily filled with numerous musical symbols typically reminds of terse music. Probably more important, when mixing colors, most people think in terms of subtractive rather than additive mixing. Conversion between HSB and RGB is just another basis change of the color space.

When changing the basis of the color space, care must be taken with respect to the range of the vector components. As previously mentioned, the subset of the $\mathbf{R}^3$, that forms the color space, is not a vector space, since the subset is not closed with respect to addition and multiplication by scalar. By changing the basis in $\mathbf{R}^3$, the cubic shape of the RGB color space in the first octant generally transforms into a different shape that possibly covers different octants, thereby changing the valid range of the vector components. Therefore, when operating with a different basis, vectors must be carefully checked for correct range.

6 SoundPaint Prototype Implementation

In order to demonstrate that our approach works, a prototype has been implemented in C++. The code currently runs under Linux, using wxWidgets(Roebling et al., 2005) as GUI library. The GUI of the current implementation mainly focuses on providing a graphical front-end for specifying an image, and parameterizing and running the transformation process, which synthesizes an audio file from the image file. An integrated simple audio file player can be used to perform the sound collage after transformation.


Color Settings

Sound Color 1
Color: 🟥
Sample: Partials

Sound Color 2
Color: 🟩
Sample: Triangle

Sound Color 3
Color: 🟦
Sample: Square

Figure 1: Mapping Colors to Sounds

Currently, only the RGB color space is supported with the three basis vectors red, green, and blue. The user defines a color-to-sound mapping by simply defining three sounds to be associated with the three basis colors. Figure 1 shows the color-to-sound mapping dialog. A generic type of wave form can be selected from a list of predefined choices and further parameterized, as shown in Figure 2 for the type of triangle waves. All parameters that go beyond manipulating the core wave form – namely pan, vibrato depth and rate, and noise content – are common to all types of wave forms, such that they can be linearly interpolated between different types. Parameters such as the duty cycle however only affect a particular wave form and thus need not be present for other types of wave forms.

Some more details of the transformation are worth mentioning. When applying the core transformation as described in Section 2, the

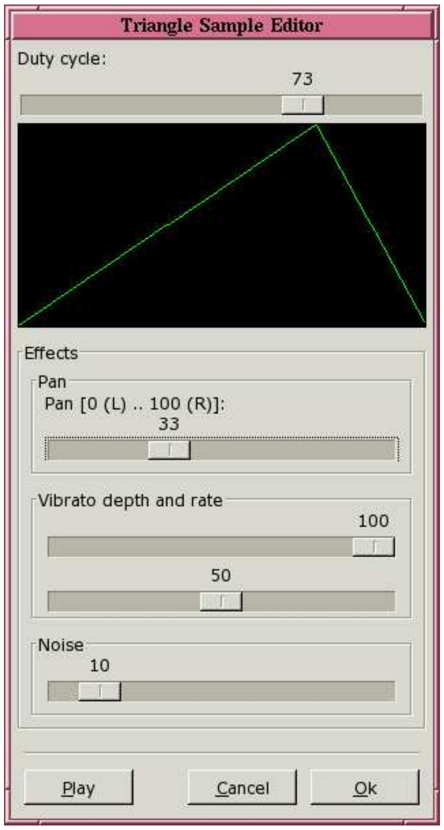


Figure 2: Parameterizing a Triangle Wave

resulting audio file will typically contain many crackling sounds. These annoying noises arise from sudden color or brightness changes at pixel borders: a sudden change in sound produces high-frequency peaks. To alleviate these noises, pixel borders have to be smoothened along the time axis. As a very simple method of anti-aliasing, SoundPaint horizontally divides each image pixel into sub-pixels down to audio resolution and applies a deep path filter along the sub-pixels. The filter characteristics can be controlled by the user via the `Synthesize Options` widget, ranging from a plain overall sound with clearly noticeable clicks to a smoothened, almost reverb-like sound.

Best results are achieved when painting only a few colored structures onto the image and leaving the keeping the remaining pixels in the color that will produce silence (i.e., in the RGB model, black). For performance optimization, it is therefore useful to handle these silent pixels separately, rather than computing a complex sound with an amplitude of 0. Since, as an effect of the before mentioned pixel smoothing, often only very few pixels are exactly 0, SoundPaint simply assumes an amplitude of 0, if the amplitude level falls below a threshold value. This threshold value can be controlled via the `gate` parameter in the `Synthesize Options` widget.

7 Preliminary Experience

SoundPaint was first publically presented in a workshop during the last *Stadtgeburtstag* (city's birthday celebrations) of the city *Karlsruhe*(Sta, 2004). Roughly 30 random visitors were given the chance to use SoundPaint for a 30 minutes slot. A short introduction was presented to them with emphasis on the basic concepts from a composer's point of view and basic use of the program. They were instructed to paint on black background and keep the painting structurally simple for achieving best results. For the actual process of painting, XPaint (as default) and Gimp (for advanced users) were provided as external programs.

Almost all users were immediately able to produce sound collages, some of them with very interesting results. What turned out to be most irritating for many users is the additive interpretation of mixed colors. Also, some users started with a dark gray rather than black image background, such that SoundPaint's optimization code for silence regions could not be applied, resulting in much slower conversion. These observations strongly suggest to introduce HSB color space in SoundPaint.

8 Future Work

Originally stemming from a command-line tool, SoundPaint still focuses on converting image files into audio files. SoundPaint's GUI mostly serves as a convenient interface for specifying conversion parameters. This approach is, from a software engineering point of view, a good basis for a clean software architecture, and can be easily extended e.g. with scripting purposes in mind. A composer, however, may prefer a sound collage in a more interactive way rather than creating a painting in an external application and repeatedly converting it into an audio file in a batch-style manner. Hence, SoundPaint undoubtedly would benefit from integrating painting facilities into the application itself.

Going a step further, with embedded painting facilities, SoundPaint could be extended to support live performances. The performer would simply paint objects *ahead* of the cursor of SoundPaint's built-in player, assuming that the image-to-audio conversion can be performed in real-time. For Minimal Music like performances, the player could be extended to play in loop mode, with integrated painting facilities allowing for modifying the painting for the next loop. Inserting or deleting multiple objects following predefined rhythmical patterns with a single action could be a challenging feature.

Assembling audio files generated from multiple images files into a single sound collage is desired when the surjectivity of our mapping is an issue. Adding this feature to SoundPaint would ultimately turn the software into a multi-track composing tool. Having a multi-track tool, integration with other notation approaches seems nearby. For example, recent development of LilyPond's(Nienhuys and Nieuwenhuizen, 2005) GNOME back-end suggests to integrate traditional notation in separate tracks into SoundPaint. The overall user interface of such a multi-track tool finally could look similar to the arrange view of standard sequencer software, but augmented by graphical notation tracks.

9 Related Work

Graphical notation of music has a rather long history. While the idea of graphical composing as the reverse process is near at hand, practically usable tools for off-the-shelf computers emerged only recently. The most notably tools are presented below.

Maybe Iannis Xenakis was the first one who started designing a system for converting images into sounds in the 1950's, but it took him decades to present the first implementation of his UPIC system in 1978(Xenakis, 1978). Like SoundPaint, Xenakis uses the coordinate axes following the metaphor of scores. While SoundPaint uses a pixel-based conversion that can be applied on any image data, the UPIC system assumes line drawings with each graphical line being converted into a melody line.

Makesound(Burrell, 2001) uses the following mapping for a sinusoidal synthesis with noise content and optional phase shift:

x position	phase
y position	temporal position
hue	frequency
saturation	clarity (inv. noise content)
luminosity	intensity (amplitude)

In Makesound, each pixel represents a section of a sine wave, thereby somewhat following the idea of a spectrogram rather than graphical notation. Color has no effect on the wave shape itself.

EE/CS 107b(Suen, 2004) uses a 2D FFT of each of the RGB layers of the image as basis for a transformation. Unfortunately, the relation between the image and the resulting sound is not at all obvious.

Coagula(Ekman, 2003) uses a synthesis method that can be viewed as a special case of SoundPaint's synthesis with a particular set of color to sound mappings. Coagula uses a sinusoidal synthesis, using x and y coordinates as time and frequency axis, respectively. Noise content is controlled by the image's blue color layer. Red and green control stereo sound panning. Following Coagula's documentation, SoundPaint should show a very similar behavior when assigning 100% noise to blue, and pure sine waves to colors red and green, with setting red color's pan to left and green color's pan to right.

Just like Coagula, MetaSynth(Wenger and Spiegel, 2005) maps red and green to stereo panning, while blue is ignored.

Small Fish(Furukawa et al., 1999), presented by the ZKM(ZKM, 2005), is an illustrated booklet and a CD with 15 art games for controlling animated objects on the computer screen. Interaction of the objects creates polytonal sequences of tones in real-time. Each game defines its own particular rules for creating the tone sequences from object interaction. The tone sequences are created as MIDI events and can be played on any MIDI compliant tone generator. Small Fish focuses on the conversion of movements of objects into polytonal sequences of tones rather than on graphical notation; still, shape and color of the animated objects in some of the games map to particular sounds, thereby translating basic concepts of graphical notation into an animated real-time environment.

The PDP(Schouten, 2004) extension for the Pure Data(Puckette, 2005) real-time system follows a different approach in that it provides a framework for general image or video data processing and producing data streams by serialization of visual data. The resulting data stream can be used as input source for audio processing.

Finally, it is worth mentioning that the visualization of acoustic signals, i.e. the opposite conversion from audio to image or video, is frequently used in many systems, among them Winamp(Nullsoft, 2004) and Max/MSP/Jitter(Cycling '74, 2005). Still, these species of visualization, which are often implemented as real-time systems, typically work on the audio signal level rather than on the level of musical structures.

10 Conclusions

We presented SoundPaint, a tool for creating sound collages based on transforming image data into audio data. The transformation follows to some extent the idea of graphical notation, using x and y axis for time and pitch, respectively. We showed how to deal with the color-to-sound mapping problem by introducing a vector space homomorphism between color space and sound subspace. Our tool mostly hides mathematical details of the transformation from the user without imposing restrictions in the choice of parameterizing the transformation. First experience with random users during the city's birthday celebrations demonstrated the usefulness of our tool. The result of our work is available as open source at `http://www.ipd.uka.de/~reuter/soundpaint/`.

11 Acknowledgments

The author would like to thank the Faculty of Computer Science of the University of Karlsruhe for providing the infrastructure for developing the SoundPaint software, and the department for technical infrastructure (ATIS) and Tatjana Rauch for their valuable help in organizing and conducting the workshop at the city's birthday celebrations.

References

Willi Apel. 1962. *Die Notation der polyphonen Musik 900-1600.* Breitkopf & Härtel, Wiesbaden.

Michael Burrell. 2001. Makesound, June. URL: ftp://mikpos.dyndns.org/pub/src/.

Cycling '74. 2005. Max/MSP/Jitter. URL: http://www.cycling74.com/.

Rasmus Ekman. 2003. Coagula. URL: http://hem.passagen.se/rasmuse/Coagula.htm.

Kiyoshi Furukawa, Masaki Fujihata, and Wolfgang Münch. 1999. *Small fish: Kammermusik mit Bildern für Computer und Spieler*, volume 3 of *Digital arts edition.* Cantz, Ostfildern, Germany. 56 S. : Ill. + CD-ROM.

Meinolf Neuhäuser, Hans Sabel, and Richard Rudolf Klein. 1974. *Bunte Zaubernoten. Schulwerk für den ganzheitlichen Musikunterricht in der Grundschule.* Diesterweg, Frankfurt am Main, Germany.

Han-Wen Nienhuys and Jan Nieuwenhuizen. 2005. LilyPond, music notation for everyone. URL: http://lilypond.org/.

Nullsoft. 2004. Winamp. URL: http://www.winamp.com/.

Miller Puckette. 2005. Pure Data. URL: http://www.puredata.org/.

Robert Roebling, Vadim Zeitlin, Stefan Csomor, Julian Smart, Vaclav Slavik, and Robin Dunn. 2005. wxwidgets. URL: http://www.wxwidgets.org/.

Tom Schouten. 2004. Pure Data Packet. URL: http://zwizwa.fartit.com/pd/pdp/overview.html.

Nigel Simeone. 2001. Universal edition history.

2004. Stadtgeburtstag Karlsruhe, June. URL: http://www.stadtgeburtstag.de/.

Karl-Heinz Stockhausen. 1956. Studie II.

Jessie Suen. 2004. EE/CS 107b. URL: http://www.its.caltech.edu/~chia/EE107/.

Naoyuki Taneda and Ruth Taneda. 1993. *Erziehung zum absoluten Gehör. Ein neuer Weg am Klavier.* Edition Schott, 7894. B. Schott's Söhne, Mainz, Germany.

Rainer Wehinger. 1970. *Ligeti, Gyorgy: Articulation. An aural score by Rainer Wehinger.* Edition Schott, 6378. B. Schott's Söhne, Mainz, Germany.

Albert Wellek. 1954. Farbenhören. *MGG – Musik in Geschichte und Gegenwart*, 4:1804–1811.

Eric Wenger and Edward Spiegel. 2005. Methasynth 4, January. URL: http://www.uisoftware.com/DOCS_PUBLIC/MS4_Tutorials.pdf.

Henrik Wiese. 2002. *Preface to Concert for Horn and Orchestra No. 4, E flat major, K495.* Edition Henle, HN 704. G. Henle Verlag, München, Germany. URL: http://www.henle.de/katalog/Vorwort/0704.pdf.

Iannis Xenakis. 1978. The UPIC system. URL: http://membres.lycos.fr/musicand/INSTRUMENT/DIGITAL/UPIC/UPIC.htm.

2005. Zentrum für Kunst und Medientechnologie. URL: http://www.zkm.de/.

System design for audio record and playback with a computer using FireWire

Michael SCHÜEPP
BridgeCo AG
michael.schuepp@bridgeco.net

Rolf "Day" KOCH
BridgeCo AG
rolf.koch@bridgeco.net

Rene Widtmann
BridgeCo AG
rene.widtmann@bridgeco.net

Klaus Buchheim
BridgeCo AG
klaus.buchheim@bridgeco.net

Abstract

This paper describes the problems and solutions to enable a solid and high-quality audio transfer to/from a computer with external audio interfaces and takes a look at the different elements that need to come together to allow high-quality recording and playback of audio from a computer.

Keywords

Recording, playback, IEEE1394

1 Introduction

Computers, together with the respective digital audio workstation (DAW) software, have become powerful tools for music creation, music production, post-production, editing. More and more musicians turn to the computer as a tool to explore and express their creative ideas. This tendency is observed for both, professional and hobby musicians. Within the computer music market a trend towards portable computers can be observed as well. Laptops are increasingly used for live recordings outside a studio as well as mobile recording platforms. And, with more and more reliable system architectures, laptops/computers are also increasingly used for live performances.

However making music on a computer faces the general requirement to convert the digital music into analogue signals as well as to digitize analogue music to be processed on a computer.

Therefore the need for external, meaning located outside of the computer housing, audio interfaces is increasing.

The paper describes a system architecture for IEEE1394 based audio interfaces including the computer driver software as well as the audio interface device.

1.1 System Overview

When discussing the requirements for an audio interfaces it is important to understand the overall system architecture, to identify the required elements and the environment in which those elements have to fit in.

The overall system architecture can be seen in the following figure:

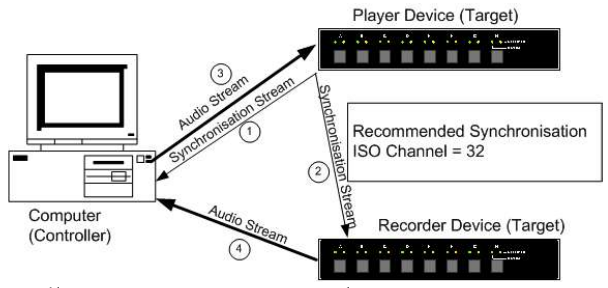


Illustration 1: Computer audio system overview

The overall system design is based on the following assumptions:

- A player device receives m audio channels (connection 3), from the computer, and plays them out. In addition it plays out data to i MIDI Ports. The data (audio and MIDI) sent from the computer are a compound stream.
- A recorder device records n audio channels and sends the data to the computer (connection 4). In addition it records data from j MIDI Ports and sends their data to the computer. The data (audio and MIDI) sent from the computer are a compound stream.
- A device can send or receive a synchronisation stream (connection 1 and 2). Typically one of the Mac/PC attached audio devices is the clock master for the synchronisation stream.

The player and recorder functions can be stand-alone devices or integrated into the same device.

1.2 What is there?

In the set-up above the following elements already exist and are widely used:

On computers:

- Digital audio workstation software such as Cubase and Logic with their respective audio APIs (ASIO and CoreAudio)
- Operating system (here Windows XP and Apple Mac OS X)
- Computer hardware such as graphic cards, OHCI controllers, PCI bus etc.

On audio interface:

- Analogue/Digital converters with I2S interfaces

All of the above elements are well accepted in the market and any solution to be accepted in a market place needs to work with those elements.

1.3 What is missing?

The key elements that are missing in the above system are the following:

1. The driver software on the computer that takes the audio samples to/from a hardware interface and transmits/receives them to/from the audio APIs of the music software.
2. The interface chip in the audio interface that transmits/receives the audio samples to/from the computer and converts them to the respective digital format for the converter chips.

The paper will now focus on these two elements and shows, what is required for both sides to allow for a high-quality audio interface. In a first step we will look at the different problems we face and then at the proposed solutions.

2 Issues to resolve

To allow audio streaming to/from a computer the following items have to be addressed:

2.1 Signal Transport

It has to be defined how the audio samples get to/from the music software tools to the audio interface. The transfer protocol has to be defined as well the transfer mode.

Additionally precautions to reduce the clock jitter during the signal transport have to be taken. Also the latency in the overall system has to be addressed.

2.2 Synchronization

In a typical audio application there are many different clock sources for the audio interface. Therefore we have the requirement to be able to synchronize to all of those clock sources and to have means to select the desired clock source.

2.3 Signal Processing

For low latency requirements and specific recording set-up, it is required to provide the capability for additional audio processing in the audio interface itself. An example would be a direct monitor mixer that mixes recorded audio onto the audio samples from the computer.

2.4 Device Management

Since we have the requirement to sell our product to various different customers as well as for various different products in a short time-to-market, it is necessary to provide a generic approach that reduces the customization efforts on the firmware and driver. Therefore it was necessary to establish a discovery process that allows the driver at least to determine the device audio channels and formats on-the-fly. This would reduce the customization efforts significantly. Therefore means to represent the device capabilities within the firmware and to parse this information by the driver have to be found.

2.5 User Interface

It must be possible to define a user interface on the device as well as the computer or a mix of both. Therefore it is required to provide means to supply control information from both ends of the system

2.6 Multi-device Setup

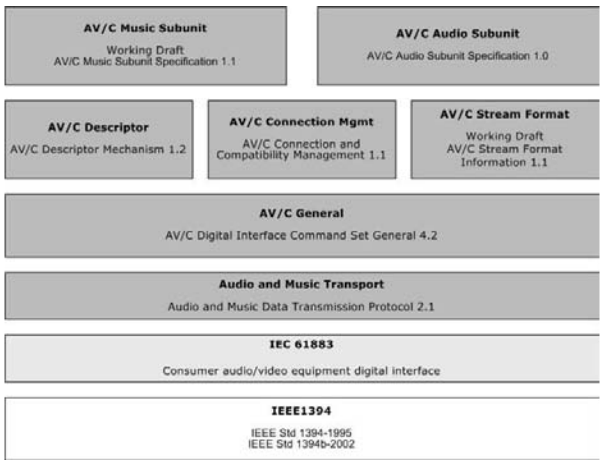
It is believed that it must be possible to use several audio interfaces at the same time to provide more flexibility to end-users. This puts additional requirements on all above issues. To avoid sample rate conversion in a multi-device setup it is mandatory to allow only a single clock source within the network. This requirement means to select the master clock within the multi-device setup as well as to propagate the clock information within the network so that all devices are slaved to the same clock.

3 Resolution

Very early in the design process it was decided to use the IEEE1394 (also called FireWire) standard

[6] as the base for the application. The IEEE1394 standard brings all means to allow isochronous data streaming, it is designed as a peer-to-peer network and respective standards are in place to transport audio samples across the network. It was also decided to base any solution on existing standards to profit from already defined solutions. However it was also clear that the higher layers of the desired standards were not good enough to solve all of our problems. Therefore efforts have been undertaken to bring our solutions back to the standardization organisations.

Overall the following standards are applied to define the system:


AV/C Music Subunit
Working Draft
AV/C Music Subunit Specification 1.1
AV/C Audio Subunit
AV/C Audio Subunit Specification 1.0
AV/C Descriptor
AV/C Descriptor Mechanism 1.2
AV/C Connection Mgmt
AV/C Connection and Compatibility Management 1.1
AV/C Stream Format
Working Draft
AV/C Stream Format Information 1.1
AV/C General
AV/C Digital Interface Command Set General 4.2
Audio and Music Transport
Audio and Music Data Transmission Protocol 2.1
IEC 61883
Consumer audio/video equipment digital interface
IEEE1394
IEEE Std 1394-1995
IEEE Std 1394b-2002
Illustration 2: Applied standards

As we can see, a variety of standards on different layers and from different organisations are used.

3.1 Signal Transport

The signal transport between the audio interfaces and the computer is based on the isochronous packets defined in the IEEE1394 standard. The format and structures of audio packets is defined in IEC61883-6 standard [6], which is used here as well. However a complex audio interface requires transmitting several audio and music formats at the same time. This could e.g. be PCM samples, SPDIF framed data and MIDI packets. An additional requirement is synchronicity between the different formats. Therefore it was decided to define a single isochronous packet, based on an IEC 61338-6 structure that contains audio and music data of different formats. Such a packet is called a compound packet. The samples in such a packet are synchronized since the time stamp for the packet applies to all the audio data within the packet.

IEC 61883-6 packets that contain data blocks with several audio formats are called compound packets. Isochronous streams containing compound packets are called compound streams. Compound streams are used within the whole system to transfer audio and music data to/from the audio interface.

The IEC 61883-6 standard defines the structure of an audio packet being sent over the IEEE1394 bus. The exact IEC 61883-6 packet structure can be found in [6].

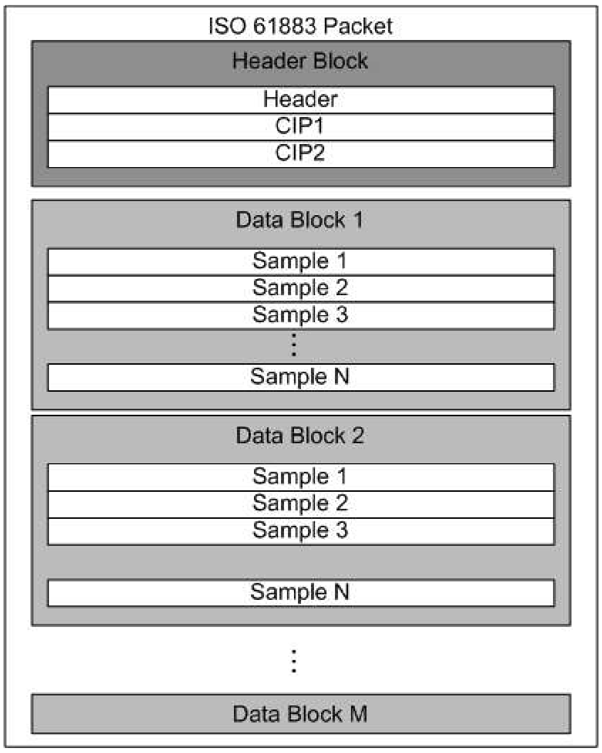


Illustration 3: IEC 61883 packet structure

The blocking mode is our preferred mode for data transmission on the IEEE1394 bus. In case data is missing to complete a full packet on the source side empty-packets are being sent. An empty-packet consists of only the header block and does not contain data. The SYT field in the CIP1 header is set to 0xffff.

The following rules to create an IEC 61883 compliant packet are applied:

- A packet always begins with a header block consisting of a header and two CIP header quadlets.
- (M) data blocks follow the header block.
  Table 1 defines M and its dependency on the stream transfer mode.
- In blocking mode, the number of data blocks is constant. If insufficient samples are available to fill

all the data blocks in a packet, an empty packet will be sent.

- In non-blocking mode, all the available samples are placed in their data blocks and sent immediately. The number of data blocks is not constant.

Sampling Frequency (FDF) [kHz]	Blocking Mode	Non-Blocking Mode
32	8	5-7
44.1	8	5-7
48	8	5-7
88.2	16	11-13
96	16	11-13
176.4	32	23-25
196	32	23-25

Table 1: Number of data blocks depending on the sampling frequency

The header information and structure for an isochronous IEC61883 packet is defined as follows:

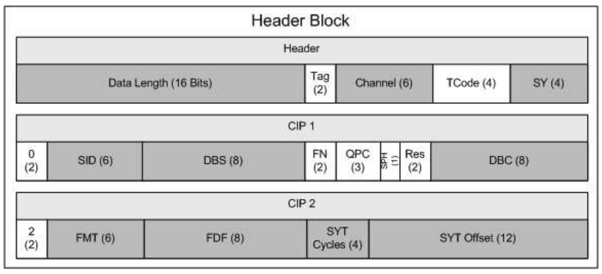


Illustration 4: IEC 61883 packet header

Table 2 describes the different elements and their definition within the packet header:

Field	Description
Data Length	Length in bytes of the packet data, including CIP1 and CIP2 header.
Channel	Isochronous channel to which the packet belongs.
SY	"System" Can be of interest if DTCP (streaming encryption) is used.
SID	"System Identification" Contains the IEEE1394 bus node id of the stream source.
DBS	"Data Block Size" Contains information about the number of samples belonging to a data block.
DBC	"Data Block Count" Is a counter for the number of data blocks that have already been sent. It can be used to detect multiply sent packets or to define the MIDI port to which the sample belongs.
FMT	"Format" The format of the stream. For an audio stream this field is always 0x10.
FDF	The nominal sampling frequency of the stream. See [3.1] for value definition.
SYT Cycles	This field, in combination with the SYT Offset field, defines the point in time when the packet should be played out. Value range: 0 – 15
SYT Offset	This field, in combination with the SYT Cycles field, defines the point in time when the packet should be played out. Value range: 0 – 0xBFF

Table 2: IEC 61883 packet header fields

Within an IEC 61883 packet, the data blocks follow the header. For the data block structure we applied the AM824 standard as defined in 6.

An audio channel is assigned to a slot within the data block:

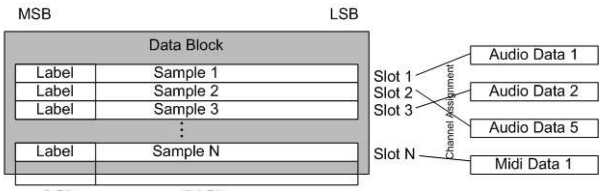
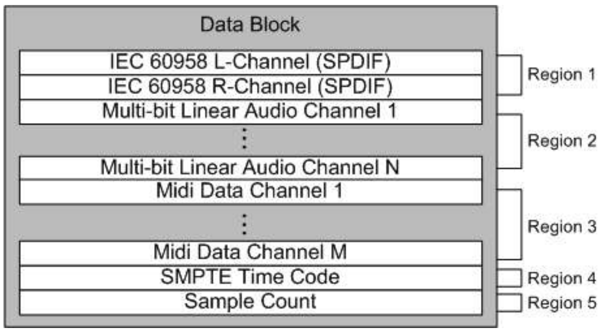


Illustration 5: Data block structure

The following rules apply to assemble the data blocks:

1. The number of samples (N) within a data block of a stream is constant.
2. The number of samples should be even (padding with ancillary no-data samples see 6)
3. The label is 8 bits and defines the sample data type
4. The sample data are MSB aligned
5. The channel to slot assignment is constant

The channel to data block slot assignment is user defined. To create a compound packet, a data structure had to be defined to place the different audio and music formats within the data blocks. No current standard defines the order in which such user data must be placed within a data block. The current standard 6 simply provides a recommended ordering. We applied this recommendation for our application and made the following data block structure mandatory to stream audio and music information:


Data Block
IEC 60958 L-Channel (SPDIF)
IEC 60958 R-Channel (SPDIF)
Multi-bit Linear Audio Channel 1
Multi-bit Linear Audio Channel N
Midi Data Channel 1
Midi Data Channel M
SMPTE Time Code
Sample Count
Region 1
Region 2
Region 3
Region 4
Region 5
Illustration 6: User data order

The following rules are applied to create the data blocks of a compound packet:

1. A region within a data block always contains data with the same data type
2. Not every region type must exist in a packet

The following region order is used:

1. SPDIF: IEC 60958 (2 Channels)
2. Raw Audio: Multi-Bit Linear Audio (N Channels)
3. MIDI: MIDI Conformant Data
4. SMTPE Time Code
5. Sample Count

MIDI data is transferred, like audio data, within channels of a compound data block. Because of the low transfer rate of one MIDI port, data of 8 MIDI ports, instead of just one, can be transferred in one channel. As shown in Illustration 7, one data part of a MIDI port will be transferred per data block and channel. This method of splitting data is called multiplexing.

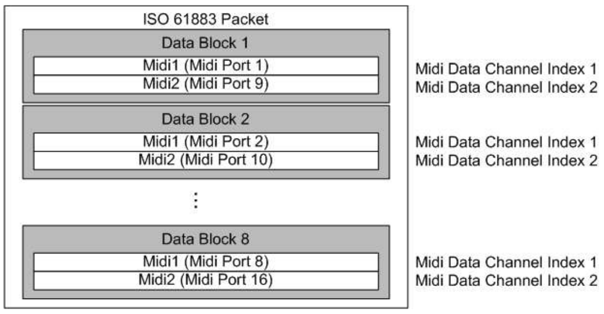


Illustration 7: MIDI data multiplexing

For the two main elements in this system, the driver and the interface processor, it is required to assemble the data packet correctly when sending data as well as to receive and disassmble the packets. Based on the dynamics of the system with different channel counts and formats the final packet structure has to be derived from the configuration from the interface such as number of channels per format and sample rate. Overall in the system it is required to keep the latency low so the framing and deframing processes have to be done as efficiently as possible.

3.2 Synchronization

The system synchronization clock for an IEEE1394 based audio interface can normally be retrieved from four different sources:

1. The time information in the SYT header field of an incoming isochronous stream.
2. The 8KHz IEEE1394 bus Cycle Start Packet (CSP).
3. The time information from an external source like Word Clock or SPDIF.
4. A clock generated by a XO or VCXO in the device.

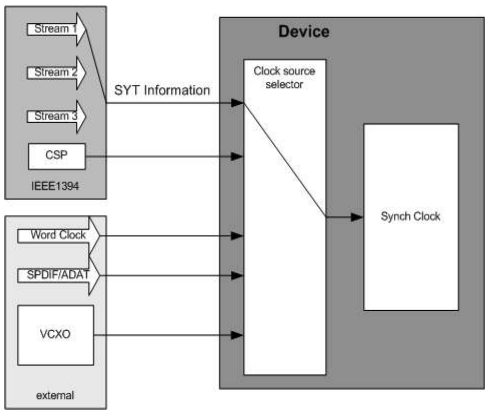


Illustration 8: Possible synchronization sources for an IEEE1394 based audio interface

3.3 Signal Processing

The specific architecture of the BridgeCo DM1x00 series is designed to enable signal processing of the audio samples once they have been deframed:

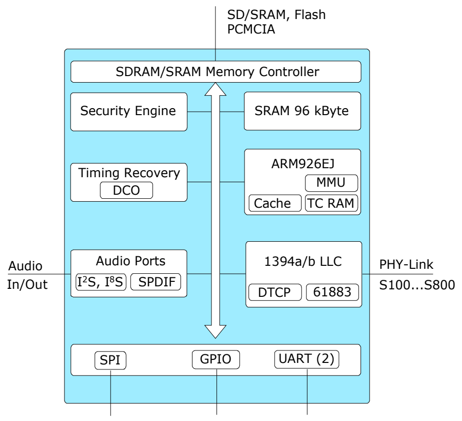


Illustration 9: Architecture of the BridgeCo DM1000 processor

Since the on-board ARM processor core can access every audio sample before it is either sent to the audio ports or sent to the IEEE1394 link layer, it is possible to enable signal processing on the audio interface. Typical applications used in this field are direct monitor mixers, which allow mixing the inputs from the audio ports with the samples from the IEEE1394 bus before they are played out over the audio ports:

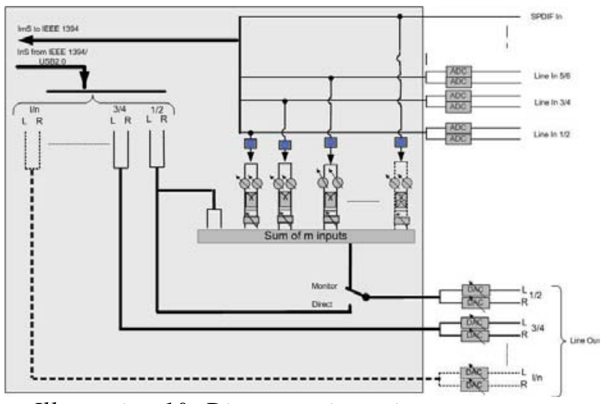


Illustration 10: Direct monitor mixer

3.4 Device Management

The device management plays a key role within the overall concept. To implement a generic approach it is absolutely necessary for the driver software to determine the device capabilities such as number of audio channels, formats and possible sample rates on-the-fly. Based on that information, the driver can expose the respective interfaces to the audio software APIs. The device management and device discovery is normally defined in the AV/C standards from the 1394TA. To achieve our goals, several audio and music related AV/C standards have been used:

- AV/C Music Subunit Specification V1.0
- AV/C Stream Format Information Specification V1.0
- AV/C Audio subunit 1.0

However the standards did not provide all means to describe and control devices as intended. Therefore two of above standards, AV/C Music Subunit and AV/C Stream Format Information are currently updated within the 1394TA organization, based on the experience and implementations from Apple Computer and BridgeCo.

Using AV/C, the driver has the task to determine and query the device for its capabilities whereas the device (meaning the software running on the device) needs to provide all the information requested by the driver to allow to stream audio between the driver and the audio interface. As soon as the device is connected to the driver via IEEE1394, a device discovery process is started. The discovery process is based on a sequence of AV/C commands exchanged between the driver and the device. Once this sequence is executed and the device is recognized as an AV/C device, the driver starts to query the device for the specific device information. This can either be done using proprietary AV/C commands or by parsing an AV/C descriptor from the device.

Within the device, the control flow and different signal formats are described with an AV/C model. The AV/C model is a representation of the internal control flow and audio routing:

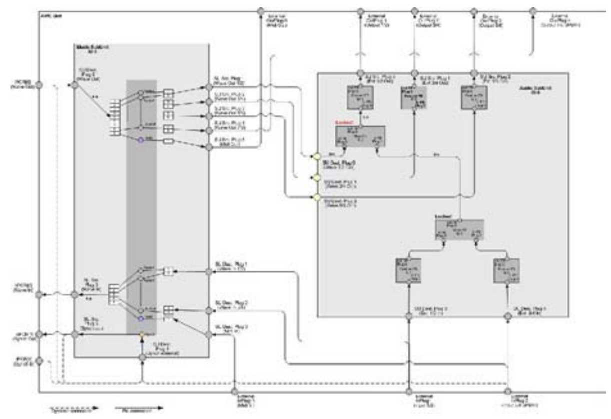


Illustration 11: Typical AV/C model

The following rules are applied to an AV/C model:

1 Fixed connections must not be changed. Every control command requesting such a change must be rejected.
2 Every unclear requested connection (like Ext. IPlug0 → Destination Plug) must be rejected.

In the AV/C model we also see the control mechanism for the direct monitor mixer which can be controlled over AV/C e.g. to determine the levels of the different inputs into the mixer.

Based on this information, the driver software can now determine the number of audio channels to be sent to the device, the number of audio channels received from the device, the different formats and expose this information to the audio streaming APIs such as ASIO or CoreAudio and expose all available sample rates to a control API.

3.5 User Interface

In the described system of an audio interface connected to a computer there are two natural points to implement a user interface:

- A control panel on the computer
- Control elements on the audio interface

Depending on customer demands and branding, different OEMs have different solutions/ideas of a control interface. In our overall system architecture we understand that it is impossible to provide a generic approach to all possible configurations and demands. Therefore we decided to provide APIs that can easily be programmed. Those APIs have to be present on both sides, on the driver as well as in the device software:

- On the driver side we expose a control API that allows direct controlling the device as well as to sent/use specific bus commands.
- On the device we have several APIs that allow to link in LEDs, knobs, buttons and rotaries.

The commands from the control API of the driver are send as AV/C commands or vendor specific AV/C commands to the device. The control API provides the correct framing of those commands whereas the application utilizing the control API needs to fill in the command content.

On the device side, those AV/C commands are received and decoded to perform the desired action. This could e.g. be different parameters for the mixer routines or being translated into SPI sequences to control and set the external components.

Next to the UI information, the device might need to send additional information to a control panel, e.g. peak level information of different audio samples, rotary information for programmable functions etc. For those high-speed data transfers, the AV/C protocol can be too slow since it e.g. allows a timeout of about 10 msec before sending retries. Within that time frame, useful and important information might already be lost. Therefore we deployed a high-speed control interface (HSCI) that allows the device to efficiently provide information to the driver. With the HSCI, the device writes the desired information into a reserved space of the IEEE1394 address space of the device. This allows the application on the computer doing highly efficient asynchronous read requests to this address space to obtain the information. Since the information structure is such, that no information gets lost, the PC application can pull the information when needed.

3.6 Multi-device Setup

A multi-device setup is normally needed when users like to use multiple device to gain a higher channel count or use different formats that are not all available in a single audio interface:

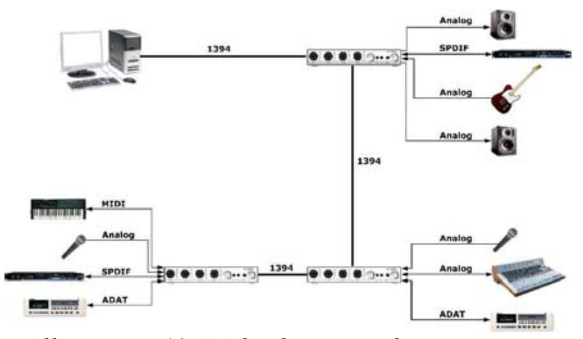


Illustration 12: Multi-device configuration

If multiple audio interfaces are connected to a computer, we face certain limitations, either imposed by the operating system, the audio software on a computer, APIs etc. :

1. ASIO and CoreAudio based audio software (e.g. Cubase) can only work with a single device.
2. To avoid sample rate conversion, only a single clock source can be allowed for all devices.
3. All devices need to be synchronised over the network

To overcome those limitations the driver software has to provide the following capabilities:

1. Concatenate multiple IEEE1394 devices into a single ASIO or CoreAudio device for the audio software application.
2. Allow selecting the clock source for each device.
3. Ability to transmit and send several isochronous streams.
4. Ability to supply the same SYT values to all transmitted isochronous streams

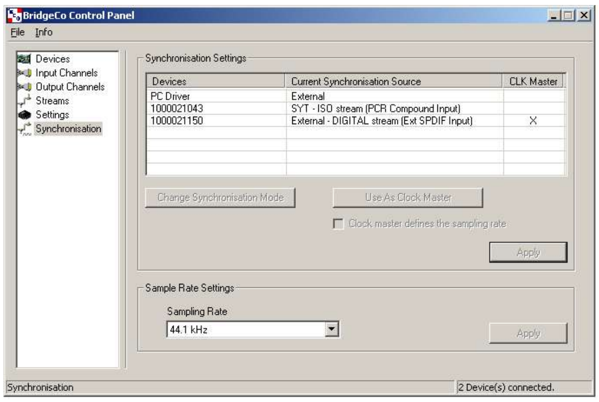
The device itself needs to provide the following functions:

1. Expose available clock sources to the driver software
2. Generate correct SYT values for outgoing isochronous streams

To synchronise multiple devices on a single clock source, which might be an external clock source for one of the devices, the following clocking scheme is used:

1. A device must be selected as clock master. This can be the computer as well.
2. If an external device is the clock master, the driver software synchronizes to the SYT time stamps within the isochronous stream from the clock master device.
3. The driver copies the received SYT time stamps from the clock master stream to its outgoing stream for all other devices.

4. All external devices expect the clock master use the SYT time stamps of their incoming isochronous stream as a clock source.



Synchronisation Settings

Devices	Current Synchronisation Source	CLK Master
PC Driver	External	
1000021043	SYT - ISO stream (PCR Compound Input)	
1000021150	External - DIGITAL stream (Ext SPDIF Input)	X

☐ Clock master defines the sampling rate

Sample Rate Settings

Sampling Rate: 44.1 kHz

Synchronisation

2 Device(s) connected.

Illustration 13: Example for a multi-device clock setup

Now, all devices are slaved across the IEEE1394 network to a single clock source. This avoids to use word clock or similar schemes to synchronize multiple devices.

Based on above configurations, each device needs to be able to synchronize on the SYT time stamps of the isochronous packets to work in such an environment.

Therefore the following features are required for the driver software:

1. Concatenate multiple devices into a single device on the audio API (ASIO or CoreAudio)
2. Allow synchronizing on the SYT time stamps from a selected isochronous stream
3. Generate correct SYT time stamps for all isochronous streams based on the received SYT time stamps
4. Parse AV/C model to determine all available clock sources on a device
5. Allow to set the clock source for each device

For the chip/firmware combination on the audio interface the following requirements must be met:

1. Allow to synchronize to SYT time stamps
2. Expose all available clock sources on the device in the AV/C Model
3. Allow external control of the clock sources via AV/C

4 FreeBob Project

Currently, there only exists drivers for the Windows and MacOS X platform, which are of course not free. The FreeBob project is trying to implement a complete free and generic driver for Linux based systems. The project is still in early stages, though first (hardcoded) prototypes are working. For further informatin please visit the website of the project (http://freebob.sf.net).

5 Conclusion

Due to the wide spectrum of interpretation within the available standards a very tight cooperation between all elements in the system is necessary. In developing such a system, it is not enough just to concentrate on and develop one element within the system. Instead it is rather required to start from a system perspective, to design an overall system concept that is initially independent of the different elements. Then, in a second step the individual tasks for each element can be defined and implemented. BridgeCo has chosen this approach and with over 20 different music products shipping today has proven that the concept and the system design approach leads to a success story. BridgeCo also likes to express its gratitude to Apple Computer, which has been a great partner throughout the design and implementation process and has provided valuable input into the whole system design concept.

6 Reference

[1] IEEE Standard 1394-1995, IEEE Standard for a High Performance Serial Bus, IEEE, July 22 1996

[2] IEEE Standard 1394a-2000, IEEE Standard for a High Performance Serial Bus—Amendment 1, IEEE, March 30 2000

[3] IEEE Standard 1394b-2002, IEEE Standard for a High-Performance Serial Bus—Amendment 2, IEEE, December 14 2002

[4] TA Document 2001024, “Audio and Music Data Transmission Protocol” V2.1, 1394TA, May 24 2002

[5] TA Document 2001012, “AV/C Digital Interface Command Set General Specification”, Version 4.1, 1394TA, December 11, 2001

[6] TA Document 1999031, “AV/C Connection and Compatibility Management Specification”, Version 1.0, 1394TA, July 10, 2000

[7] TA Document 1999025, “AV/C Descriptor Mechanism Specification”, Version 1.0, 1394TA, April 24 2001

[8] TA Document 2001007, “AV/C Music Subunit”, Version 1.0, 1394TA, April 8 2001

[9] TA Document 1999008, “AV/C Audio Subunit Specification”, Version 1.0, 1394TA, October 24 2000

[10] IEC 61883-6, Consumer audio/video equipment - Digital interface - Part 6: Audio and music data transmission protocol, IEC, October 14 2002

Recording all Output from a Student Radio Station

John ffitch
Department of Computer Science
University of Bath
Bath BA2 7AY,
UK,
jpff@cs.bath.ac.uk

Tom Natt
Chief Engineer, URB
University of Bath
Bath BA2 7AY,
UK,
ma1twn@bath.ac.uk

Abstract

Legal requirements for small radio stations in the UK mean, *inter alia*, that the student station at Bath (University Radio Bath or URB) must retain 50 days of the station's output. In addition, as it has recently become easier to transfer data using disposable media, and general technical savvy amongst presenters has improved, there is now some interest in producing personal archives of radio shows. Because existing techniques, using audio videos, were inadequate for this task, a modern, reliable system which would allow the simple extraction of any audio was needed. Reality dictated that the solution had to be cheap. We describe the simple Linux solution implemented, including the design, sizing and some surprising aspects.

Keywords

Audio archive, Audio logging, Radio Station, Portaudio.

1 Introduction

The University of Bath Student's Union has been running a radio station(URB, 2004) for many years, and it has a respectable tradition of quality, regularly winning prizes for its programmes(SRA, 2004). Unfortunately the improved requirements for logging of output from the station coincided with the liquidation of a major sponsor and hence a significant reduction in the station's income, so purchasing a commercial logging device was not an option.

Following a chance conversation the authors decided that the task was not difficult, and a software solution should be possible. This paper describes the system we planned and how it turned out. It should be borne in mind that during development, cost was the overriding factor, in particular influencing hardware choices.

2 The Problem

The critical paragraph of the regulations on Radio Restricted Service Licences, which control such activities as student broadcasting in the UK read

> **You are required to make a recording of all broadcast output, including advertisements and sustaining services. You must retain these recordings ('logging tapes') for a period of 42 days after broadcast, and make them readily available to us or to any other body authorised to deal with complaints about broadcast programmes. Failure to provide logging tapes on request will be treated seriously, and may result in a sanction being imposed.**

where the bold is in the original (OffComm, 2003). In the previous state the logging was undertaken using a video player and a pile of video tapes. These tapes were cycled manually so there was a single continuous recording of all output. This system suffered from the following problems.

The quality was largely unknown. In at least the last 3 years no one has actually listening to anything recorded there; indeed it is not known if it actually works! The system required someone to physically change the tape. Hence, there were large gaps in logging where people simply forgot to do this, which would now expose the station to legal problems.

Assuming that the tapes actually worked, recovering audio would be a painstaking process requiring copying it from the tapes to somewhere else before it could be manipulated in any way. Hence this was only really useful for the legal purposes, rather than for people taking home copies of their shows. Also, as far as could be determined, whilst recovering audio, the logger itself had to be taken offline.

Put simply, the system was out of date. Over the last two years there has been a move to modernise URB by shifting to computers where possible, so it seemed logical to log audio in such a way that it would be easy to transmit onto the network, and be secure regarding the regulations.

3 Requirements

The basic requirement for the system is that it should run reliably for many days, or even years, with little or no manual intervention. It should log all audio output from the radio station, and maintain at least 50 days of material. Secondary requirements include the ability to recover any particular section of audio by time (and by a non-technical user). Any extracted audio is for archiving or rebroadcast so it must be of sufficient quality to serve this purpose. There is another, non functional, requirement, that it should cost as close to zero as possible!

Quick calculations show that if we were to record at CD quality (44.1KHz, 16bit stereo) then we would need $44100 \times 2 \times 2$ bytes a second, or $44100 \times 2 \times 2 \times 60 \times 24 = 14$Gb each day, which translates to over 700Gb in a 50 day period. While disks are much cheaper than in earlier times, this is significantly beyond our budget. Clearly the sound needs to be compressed, and lossy compression beckons. This reduces the audio quality but, depending on compression rates, not so much that it violates our requirements.

We sized the disk requirements on a conservative assumption of 1:8 compression, which suggests at least an 80Gb disk. Quick experiments suggested about a 400MHz Intel processor; the decision to use Linux was axiomatic. Given sufficient resource, a system to record DJ training sessions and demo tapes was suggested as a simple extension.

We assumed software would be custom-written C, supported by shell scripts, cron jobs and the like. A simple user recovery system from a web interface would be attractive to the non-technical user, and it seemed that PERL would be a natural base for this.

There are commercial logging computers, but the simple cost equation rules them out.

4 Hardware

A search for suitable hardware allowed the creation of a 550MHz Celeron machine with 128Mb of memory, ethernet, and two old SoundBlasters retrieved from a discard pile. SuSE9.1(Novell, 2004) was installed with borrowed screen, keyboard and mouse. The only cash expenditure was a new 120Gb disk; we decided that 80Gb was a little too close to the edge and the additional space would allow a little leeway for any extensions, such as the DJ training.

There were two unfortunate incidents with the hardware; the disk was faulty and had to be replaced, and following the detection of large amounts of smoke coming from the motherboard we had to replace the main system; the best we could find was a 433MHz Celeron system. Fortunately the disk, soundcards and other equipment were not damaged and in the process of acquiring a new motherboard and processor combination we were lucky enough to find another stick of RAM. Most important, what we lost was time for development and testing as we needed to hit the deadline for going live at the beginning of the 2004 academic year.

Hardware	Features
433MHz Celeron	slower than our design
120Gb disk	New!
2 × SoundBlaster 16	old but working
256Mb main memory	
10 Mbit ether	

Table 1: Summary of Hardware Base

5 Implementation

The main program is that the suite needs to perform two main tasks: read a continuous audio stream and write compressed audio to the disk. The reading of the audio feed must not be paused or otherwise lose a sample. The current design was derived from a number of alternative attempts. We use a threaded program, with a number of threads each performing a small task, coordinated by a main loop. A considerable simplification was achieved by using PortAudio(Por, 2004) to read the input, using a call-back mechanism. We shamelessly cannibalised the test program `patest_record` written by Phil Burk to transfer the audio into an array in large sections. The main program then writes the raw audio in 30 second sections onto the disk. It works on a basic 5 period cycle, with specific tasks started on periods 0, 3 and 4.

On 0 a new file is used to write the raw audio, and a message is written to the syslog to indicate the time at which the file starts. On period 3 a subthread is signalled to start the compression of a raw audio file, and on period 4 the next raw audio file is named and created. By sharing out the tasks we avoid bursts of activity which could lead to audio over-runs. This is shown in figure 1.

The compression is actually performed by a lower priority sub task which is spawned by a call to system. There is no time critical aspect of the compression as long as it averages to compressing faster than the realtime input. Any local load on the machine may lead to local variation but eventually it must catch up. There is a dilemma in the compression phase. The obvious format is OGG, for which free unencumbered software exists, but the student community is more used to MP3 format. We have experimented with `oggenc`(ogg, 2004), which takes 80% of elapsed time on our hardware and compresses in a ratio of 1:10, and `notlame`(Not, 2004), where compression is 1:11 and 74% of elapsed time. Our sources have both methods built in with conditional compilation.

We have varied the period, and have decided on a minute, so each audio file represents five minutes of the station's output; this is a good compromise between overheads and ease of recovery.

The result of this program is a collection of 5 minute compressed audio files. Every day, just after midnight, these files are moved to a directory named after the day, and renamed to include the time of the start of the interval of time when the recording started. This is achieved with a small C program which reads the syslog files to get the times. This program could have been written in PERL but one of us is very familiar with C. A snapshot of part of the logging directory is shown in figure 2, where compressed audio, raw PCM files, unused PCM files and a compression log can be seen.

The decision to rename actual files was taken to facilitate convenience during soak testing. We were running the system over this time as if it were live, and hence were using it to extract logs when they were requested by presenters. Cross-referencing files with the system log was a tedious task so an automatic renaming seemed the obvious choice. Using this opportunity to refile the logs in directories corresponding to the date of logging also assisted greatly in retrieval. A more long-term consideration was that renamed files would be easier to extract via a web interface and hence this work could probably be used in the final version also.

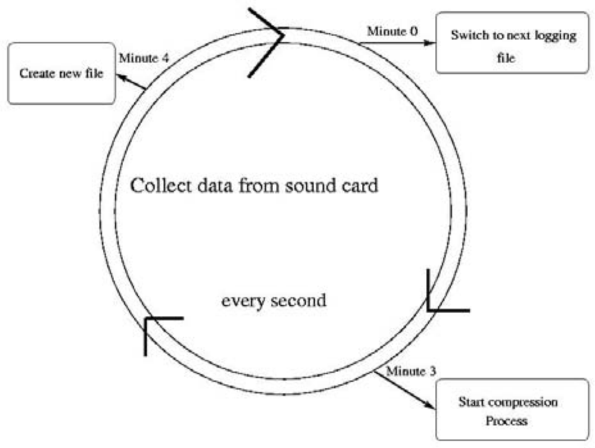

Minute 0
Switch to next logging file
Minute 4
Create new file
Collect data from sound card
every second
Minute 3
Start compression Process

Figure 1: Overview of Software Cycle

6 Experience

The program has now been running for a significant time. Two problems with the design have emerged.

The first was the change to winter time which happened a few days after going live. Our logs, and hence times, where based on local time as that seemed to be closest to what the users would require. But with the clock being put backwards, times repeat. Clearly we need to maintain both times in the logs, and append the time zone to the ultimate file name, or some similar solution. But how did we manage this shift backwards without loss of data? The answer is in the second problem.

We are capturing the raw station output in 44.1KHz 16bit stereo. Every five minutes a new file is started. Actually we are not starting files by time but by sample count (13230000 frames). As was predicted, the sound card was not sampling at exactly CD rate, but faster, and as a result we are drifting by 19 seconds a day. In itself this is not a problem, and indeed rescued the possible data loss from the introduction of winter time, but it is less convenient for the student DJs who want a copy of their program. The suggestion is that the files should be aligned on five minute boundaries by the clock. This entails monitoring the clock to decide on a change of file, which would be a considerable departure from the simplicity of design. Exactness is not important, but we would like to be less than a minute adrift. Our revised code, not yet in service, makes the switch of files after reading the clock, and additional care is needed to avoid clock drift.

```
-rw-r--r--   1 root root   4801096 Mar  7 10:59 Arc0041022.mp3
-rw-r--r--   1 root root   4801096 Mar  7 11:04 Arc0041023.mp3
-rw-r--r--   1 root root   4801096 Mar  7 11:09 Arc0041024.mp3
-rw-r--r--   1 root root   4801096 Mar  7 11:14 Arc0041025.mp3
-rw-r--r--   1 root root   4801096 Mar  7 11:19 Arc0041026.mp3
-rw-r--r--   1 root root   4801096 Mar  7 11:24 Arc0041027.mp3
-rw-r--r--   1 root root   4801096 Mar  7 11:29 Arc0041028.mp3
-rw-r--r--   1 root root   4801096 Mar  7 11:34 Arc0041029.mp3
-rw-r--r--   1 root root   4801096 Mar  7 11:39 Arc0041030.mp3
-rw-r--r--   1 root root  52920000 Mar  7 11:44 Arc0041032
-rw-r--r--   1 root root   4801096 Mar  7 11:44 Arc0041031.mp3
-rw-r--r--   1 root root         0 Mar  7 11:47 log5Pwl4o
-rw-r--r--   1 root root  42332160 Mar  7 11:48 Arc0041033
-rw-r--r--   1 root root         0 Mar  7 11:48 Arc0041034
-rw-r--r--   1 root root   3145728 Mar  7 11:48 Arc0041032.mp3
```

Figure 2: Part of Directory for Running System

It was this clock drift, which can be seen in figure 3, that saved the situation when we changed from summer time to winter time. If we had implemented the time alignment method then the file names would have repeated for the one hour overlap (1am to 2am is repeated in the UK time scheme), but as the soundcard had read faster, the second hour was fortuitously aligned to a different second, and so we could rescue the system manually.

The zone change from winter to summer involves the non-existence of an hour and so raises no problems. Before next autumn we need to have deployed a revised system. It has been suggested that using the Linux linking mechanisms we could maintain all logging in universal time, and create separate directories for users to view.

There was one further problem. When the syslog file got large the usual logrotate mechanism started a new file. But as our renaming system reads the syslog, it subsequently missed transfer and rename of some output. This was fixed by hand intervention, but at present we do not have a good solution to this; nasty solutions do occur to us though!

Another minor problem encountered during initial testing was with the hardware: it seems under Linux older versions of the SoundBlaster chipset could not handle both recording from an input stream and outputting one simultaneously. The output stream took priority so unless we specifically muted the output channels on the card, no sound was captured. This is only mentioned here in case an attempt is made to duplicate this work, and so to avoid the hours of frustration endured during our initial tests. We expect that similar minor problems will appear later as we develop the system, but the main data collection cycle seems most satisfactorily robust. Most importantly, despite being forced to downgrade our hardware, the system performs within its new limitations without loss of data during compression phases — even during periods of additional load from users (*i.e.* when logs are being extracted). There is sufficient slack for us to consider adding additional services.

7 Conclusions

Tests have demonstrated that our original aim, of a cheap data logging system, has been easily achieved — the whole system cost only £60 in new purchased materials. What is also clear is that the whole structure of the Linux and Open Source movements made this much more satisfactory than we feared. The efficiency of Linux over, say, Windows meant that we could use what was in effect cast-off hardware. The ability to separate the data collection from the compression and filing allowed a great simplification in the design, and so we were able to start the logging process days before we had thought about the disposal process, but before the start of the university term. The `crontab` mechanism enables us to delete whole directories containing a single day after 60 days have passed. We still need to implement a web-interface to extracting of programs, but the availability of PERL, Apache, and all the related mechanisms suggests that this is not a major task.

```
-rw-r--r--   1 root root 4801096 Mar  4 22:55 22:45:08.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:00 22:50:08.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:05 22:55:08.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:10 23:00:08.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:15 23:05:08.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:20 23:10:07.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:25 23:15:07.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:30 23:20:07.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:35 23:25:07.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:40 23:30:07.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:45 23:35:07.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:50 23:40:07.mp3
-rw-r--r--   1 root root 4801096 Mar  4 23:55 23:45:07.mp3
-rw-r--r--   1 root root 4801096 Mar  5 00:00 23:50:07.mp3
-rw-r--r--   1 root root 4801096 Mar  5 00:05 23:55:07.mp3
-rw-r--r--   1 root root    6912 Mar  5 01:10 index
```

Figure 3: Part of an Archive Directory

Although it is not a major problem to write, the extraction web page for the system will be the only part most users see and hence design for ease of use must be key. Currently, the idea is to incorporate this into the current URB online presence(URB, 2004) which allows members of the station to log into a members area. We will add a logging page, which presents users with a very simple interface specifying only the beginning and end points of the period required. With their user-names tied to a download destination, presenters will always find their logs in the same place, limiting the possibility of confusion.

Being based on such reliable software packages, we are sure that if we ever have sufficient funds for an upgrade, for example to a digital input feed, this can easily be accommodated. We are aware that the current system lacks redundancy, and a secondary system is high on our wish-list. More importantly we have not yet completed a physically distributed back-up in case the next machine fire *does* destroy the disk.

We are confident that as the radio station continues to be the sound-track of the University of Bath, in the background we are listening to all the sounds, logging them and making them available for inspection. With this infrastructure in place we might even consider a "play it again" facility, if the legal obstacles can be overcome.

Naturally as the program is based on open source code we are willing to provide our system to anyone else who has this or a similar problem.

8 Acknowledgements

Our thanks go to Simon Lee, the instructor of Bath University Students' Union T'ai Chi Club, for tolerating our discussions before (and sometimes during) training sessions.

References

2004. Notlame mp3 encoder. `http://users.rsise.anu.edu.au/~conrad/not_lame/`.

Novell. 2004. `http://www.novell.com/de-de/linux/suse`.

OffComm. 2003. Office of Communications Document: Long-Term Restricted Service Licences. `http://www.ofcom.org.uk/codes_guidelines/broadcasting/radio/guidance/lo%ng_term_rsl_notes.pdf`, January.

2004. Ogg front end. `http://freshmeat.net/projects/oggenc/`.

2004. PortAudio — portable cross-platform Audio API. `http://www.portaudio.com/`.

2004. SRA: Student Radio Association. `http://www.studentradio.org.uk/awards`.

2004. URB: University Radio Bath. `http://www.bath.ac.uk/~su9urb`.

AGNULA/DeMuDi - Towards GNU/Linux audio and music

Nicola Bernardini, Damien Cirotteau, Free Ekanayaka, Andrea Glorioso
Media Innovation Unit - Firenze Tecnologia
Borgo degli Albizi 15
50122 Firenze
Italy

Abstract

AGNULA (acronym for "A GNU/Linux Audio distribution", pronounced with a strong g) is the name of a project which has been funded until April 2004 by the European Commission (number of contract: IST-2001-34879; key action IV.3.3, Free Software: towards the critical mass). After the end of the funded period, AGNULA is continuing as an international, mixed volunteer/funded project, aiming to spread Free Software in the professional audio/video arena. The AGNULA team is working on a tool to reach this goal: AGNULA/DeMuDi, a GNU/Linux distribution based on Debian, entirely composed of Free Software, dedicated to professional audio research and work. This paper[1] describes the current status of AGNULA/DeMuDi and how the AGNULA team envisions future work in this area.

Keywords

AGNULA, audio, Debian

1 The AGNULA project - a bit of history

In 1998 the situation of sound/music Free Software applications had already reached what could be considered well beyond initial pioneeristic stage. A website, maintained by musician and GNU/Linux[2] enthusiast Dave Phillips, was already collecting all possible sound and music software running on GNU/Linux architectures. At that time, the biggest problem was that all these applications were dispersed over the Internet: there was no common operational framework and each and every application was a case-study by itself.

A natural development followed shortly after, when musician/composer/programmer Marco Trevisani proposed a to a small group of friends (Nicola Bernardini, Maurizio De Cecco, Davide Rocchesso and Roberto Bresin) to create LAOS (the acronym of *Linux Audio Open Sourcing*), a binary distribution of all essential sound/music tools available at the time including website diffusion and support. LAOS came up too early, and it did not go very far.

But in 2000, when Marco Trevisani proposed (this time to Nicola Bernardini, Günter Geiger, Dave Phillips and Maurizio De Cecco) to build DeMuDi (*Debian Multimedia Distribution*) an unofficial Debian-based binary distribution of sound/music Free Software, times were riper.

Nicola Bernardini organized a workshop in Firenze, Italy at the beginning of June 2001, inviting an ever–growing group of supporters and contributors (including: Marco Trevisani, Günter Geiger, Dave Phillips, Paul Davis, François Déchelle, Georg Greve, Stanko Juzbasic, Giampiero Salvi, Maurizio Umberto Puxeddu and Gabriel Maldonado). That was the occasion to start the first concrete DeMuDi distribution, the venerable *0.0 alpha* which was then quickly assembled by Günter Geiger with help from Marco Trevisani. A bootable CD-version was then burned just in time for the ICMC 2001 held in La Habana, Cuba, where Günter Geiger and Nicola Bernardini held a tutorial workshop showing features, uses and advantages of DeMuDi(Déchelle et al., 2001).

On November 26, 2001 the European Commission awarded the AGNULA Consortium — composed by the Centro Tempo Reale, IRCAM, the IUA-MTG at the Universitat Pompeu Fabra, the Free Software Foundation Europe, KTH and Red Hat France — with consistent funding for an accompanying measure lasting 24 months (IST-2001-34879). This accompanying measure, which was terminated on March 31st 2004, gave considerable thrust to

[1]This paper is Copyright © 2004 Bernardini, Cirotteau, Ekanayaka, Glorioso and Copyright © 2004 Firenze Tecnologia. It is licensed under a Creative Commons BY-SA 2.0 License (see `http://creativecommons.org/licenses/by-sa/2.0/legalcode`).

[2]Throughout the document, the term GNU/Linux will be used when referring to a whole operating system using Linux as its base kernel, and Linux when referring to the kernel alone.

the AGNULA/DeMuDi project providing scientific applications previously unreleased in binary form and the possibility to pay professional personnel to work on the distribution.

After the funded period, Media Innovation Unit, a component of Firenze Tecnologia (itself a technological agency of the Chamber of Commerce of Firenze) has decided to partly fund further AGNULA/DeMuDi developments.

AGNULA has constituted a major step in the direction of creating a full-blown Free Software infrastructure devoted to audio, sound and music, but there's much more to it: it is the first example of a European-funded project to clearly specify the complete adherence of its results to the Free Software paradigm in the project contract, thus becoming an important precedent for similar projects in the future.

2 Free Software and its applications in the "pro" audio domain

When describing the AGNULA project, and the AGNULA/DeMuDi distribution specifically, a natural question arises - why is it necessary or desiderable to have a completely Free Software based distribution (whether based on the Linux kernel or not is not the point here) for audio professionals and research in the sound domain?

Free Software[3] is the set of all computer programs whose usage and distribution licenses (think about the "EULA" or "End User Licensing Agreements", that so many users have come to know throughout the years) guarantee a precise set of freedoms:

- The freedom to run the program, for any purpose (freedom 0);
- The freedom to study how the program works, and adapt it to your needs (freedom 1). Access to the source code is a precondition for this;
- The freedom to redistribute copies so you can help your neighbor (freedom 2);
- The freedom to improve the program, and release your improvements to the public, so that the whole community benefits (freedom 3). Access to the source code is a precondition for this;

The most famous of such licenses is probably the GNU General Public License, which is the founding stone of the Free Software Foundation effort to build a completely free operating system, GNU (GNU's Not Unix).

This is not the right place to describe the concepts and the history of the Free Software movement as it would deserve.

Suffice it to say that the possibility to use, study, modify and share computer programs with other people is of paramount importance to the everyday life of creators (i.e. composers), professional users (i.e. sound engineers, performers) and researchers. This distinction is of course artificial, since all of us can be creators, professional users and researchers in specific moments of our life. But this taxonomy can work as a simple tool to better understand the pros of using Free Software in everyday life and work:

- **Creators** can use tools which don't dictate them what they *should* do, instead being easily modifiable into something that does what they *want* them to do. The non-physical nature of software makes for a very convenient material to build with; even though the creator might not have the needed technical skills and knowledge to modify the program to best suit his/her needs, s/he can always ask someone else to do it; on a related note, this kind of requests make for a potentially (and in some key areas, factually) very thriving marketplace for consultants and small businesses;
- **Professional users** have at their disposal a series of tools which were often thought and designed by other professional users; they can interact more easily with the software writers, asking features they might need or reporting bugs so that they are corrected faster (some would say "at all"). They can base their professional life not on the whim of a single company whose strategies are not necessarily compatible with the professional user's own

[3]We tend to prefer this term, rather than "Libre Software", even if the former term is inherently ambiguous because of the english term "free" — which can mean "free as in free beer" or "free as in free speech". Free Software is, of course, free as in free speech (and secondarily, but not necessarily, as in free beer). Usage of the term "Libre Software" arose in the european context trying to overcome this ambiguity with a term, libre, which is correct in the french and spanish languages and is understandable in italian and other european languages. However, it is not universally accepted as an equivalent of "Free Software" and its usage can induce confusion in readers and listeners — we therefore prefer to stick to the traditional, albeit somewhat confusing, terminology.

plans, but on a shared ecosystem of software which won't easily disappear — if the original software authors stop maintaining the software, someone else can always replace them;

- **Researchers** can truly bend the tool they have at their disposal to its maximum extent, something which is often very hard to do with proprietary software (even with well designed proprietary software, as it is basically impossible to understand all users' requirements in advance). They can count on computer programs which have been deeply scrutinized by a peer-review process which finds its counterpart only in the scientific community tradition[4] as opposed to the habit of proprietary software to completely obscure the "source code" of a program, and all the bugs with it. Last, not least, for all those researchers who use software programs not as simple tools but as bricks in software development (as often happens today in computer–assisted composition and more generally in sound research) the possibility to draw from an immense database of freely available, usable and distributable computer programs can prove an incredible advantage, especially when considering the cost of proprietary computer programs and the financial situations of most research institutions nowadays.[5]

In the end, one might ask whether creativity is truly possible without control on the tools being used — a control which Free Software guarantees and proprietary software sometimes grants, but more often than not manipulates for purely economical reasons.

This is not an easy question to answer at all — there are many subtle issues involved, which span in the field of economics, psychology, engineering, sociology, etc, etc. The AGNULA project actually believes that creativity is very difficult without such control,[6] but it's unquestionable that the subject would deserve a fairer treatise, through cross-subject studies able to span the whole range of fields outlined above.

[4]This is not a coincidence, as the GNU project was basically born in the Artificial Intelligence Laboratories at the M.I.T.

[5]It should be noted, however, that whilst monetary costs are of course a strong variable of all the equation, the central importance of Free Software in research is **not** related to money itself. Having free (i.e. gratis) software which is not free (i.e. not libre) can be an ephemeral panacea, but on the long run it simply means tying oneself and one's own research strategy to somebody else's decisions.

3 The AGNULA/DeMuDi framework

The framework of AGNULA/DeMuDi is the "classical" environment one can expect from a GNU/Linux system to run audio applications. The first component is the Linux kernel patched to turn it into an efficient platform for real time applications such as audio applications. Then the ALSA drivers allow the usage of a wide range of soundcards from consumer grade to professional quality. On top of the drivers runs the Jack server which allows lowlatency, synchronicity and inter-application communication. Last, not least, the LADSPA plugins format is the standard for audio plugins on the GNU/Linux platform.

3.1 The Linux kernel

3.1.1 Is the Linux kernel suitable for audio applications?

The heart of the AGNULA/DeMuDi distribution is the Linux kernel. However, since Linux was originally written for general purpose operating systems (mainly for servers and desktop applications) as a non preemptive kernel, it was not really useful for real-time applications. Truely showing the power of Free Software, several improvements of the kernel scheduler turned it into a good platform for a Digital Audio Workstation (DAW).

To face this limitation two strategies have been adopted: the preemption patch and the lowlatency patch.

3.1.2 Preemption patch

Originally created by MontaVista and now maintained by Robert M. Love,[7] this patch redesigns the kernel scheduler and redefines the spinlocks from their SMP specific implementation to preemption locks. This patch allows the Linux scheduler to be preemptive – when an interruption of higher priority occurs the kernel preempts the current task and runs the higher priority task – except for specific critical sections (such as spinlocks or when the scheduler

[6]This belief has become a sort of mantra, as is stated on our t-shirts: "There is no free expression without control on the tools you use".

[7]`http://www.tech9.net/rml/linux`

is running). This strategy has proven its efficiency and reliability and has been included in the new stable releases of the kernel (2.6.x).

3.1.3 Lowlatency patch

Introduced by Ingo Molnar and improved by Andrew Morton, the lowlatency[8] patch introduces some specific conditional rescheduling points in some blocks of the kernel. Even if the concept of this patch is quite simple, it imposes a very high maintenance burden because the conditional rescheduling points are spread all over the kernel code without any centralization.

3.1.4 Which patch is the best?

We test the kernel 2.4.24 with the methodology of (Williams, 2002).[9] We used *realfeel*[10] while running the Cerberus Test Control System [11] to stress the kernel. 5.000.000 interrupts were generated with a frequency of 2048 interrupt per second and the scheduling latency is measured for each interrupt on a Intel Centrino 1.4 MHz with 512 Mb of RAM .

The result for the non–patched kernel (see Figure 1) with a maximum latency of 48,1 ms makes this kernel not suitable for real–time application. The patches greatly improve the situation. The lowlatency patch provides better results – better maximum latency and highest percentage of lowlatency interrupts. The optimal choice seems to be the combination of both. The combination of the patches has also proven to be more reliable after a long uptime (see (Williams, 2002))

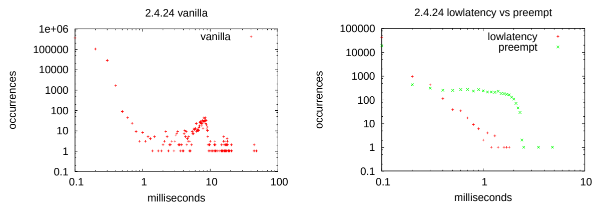


Figure 1: *Vanilla vs Lowlatency and preempt 2.4.24 scheduler latency*

Even if AGNULA/DeMuDi still provides a 2.4.x kernel some preliminary tests show that

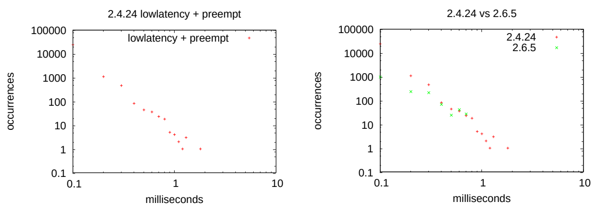


Figure 2: *Lowlatency + Preempt 2.4.24 and preempt 2.6.5 scheduler latency*

the new stable kernel (2.6.x) provides better scheduler and will therefore be very suitable for an audio platform. The preempt patch is now directly shipped with the vanilla kernel. The maximum latency measured for the 2.6.5 kernel is $0.7ms$, and the percentage of interrupts being served within 0.1 ms is significantly higher than for any version of the 2.4.24 kernel.

3.1.5 Capability patch

The third patch applied to the kernel does not improve the performance of the system but allows a non–root users to use the real time *capability* of Linux. It is particularly useful to run the Jack (see 3.3) audio server as a normal user.

3.2 ALSA

ALSA (Advanced Linux Sound Architecture) is a modular software framework which supports a wide range of soundcards[12] from consumer grade to professional quality. The ALSA drivers also provide an OSS/Free emulation to allow compatibility with legacy applications. *ALSA* is now the standard audio subsytem of the 2.6.x Linux kernels (replacing OSS, which was the standard throughout the 2.4.x series). *ALSA* also provides an API and a user space library (libasound).

3.3 The Jack Audio Connection Kit

The Jack Audio Connection Kit (*Jack*) can be considered as the real user–space skeleton of AGNULA/DeMuDi. This audio server runs on top of the audio driver (ALSA , OSS or Portaudio) and allows the different audio applications to communicate with each other. While other audio servers exist (aRts and esd among others), Jack is the only one which has been designed from the grounds up for professional audio usage: it guarantees low latency operations and synchronicity between different client

[8]`http://www.zip.com.au/~akpm/linux/schedlat.html`

[9]We invite the reader to consult this paper for a more detailed explanation of how the kernel scheduler works and of the two patches.

[10]`http://www.zip.com.au/~akpm/linux/schedlat.html#amlat`

[11]`http://sourceforge.net/projects/va-ctcs/`

[12]See the – not-so-up-to-date – soundcards matrix on the ALSA web pages to have an idea of the number of soudcards supported.

	2.4.24 vanilla	2.4.24 lowlatency	2.4.24 preempt	2.4.24 both	2.6.5 preempt
$max(L)(ms.)$	48.1	1.8	4.8	1.8	0.7
$L < 0.1ms(\%)$	90.2182	99.1168	99.5404	99.4831	99.9685
$L < 0.2ms(\%)$	97.3432	99.9679	99.9115	99.9643	99.9878
$L < 0.5ms(\%)$	99.9768	99.9976	99.9311	99.9973	99.9982
$L < 1ms(\%)$	99.9801	99.9997	99.9567	99.9998	100
$L < 10ms(\%)$	99.9983	100	100	100	100
$L < 50ms(\%)$	100	100	100	100	100

Table 1: Distribution of the latency measurements for the different kernels

applications. Therefore it has become a *de facto* standard for professional audio on GNU/Linux systems and the majority of the applications included in the AGNULA/DeMuDi distribution are Jack–compliant ("jackified", to use the relevant jargon). Another reason for Jack's success is the simple, high–level but powerful API that it provides, which has greatly facilitated the *jackification* of audio applications.

Last, not least, *Jack* also provides a master transport which allows for simultaneous control of different applications (start/pause/stop).

3.4 The LADPSA plugins

LADSPA, which stands for Linux Audio Developers Simple Plugins Architecture, is the VST equivalent on GNU/Linux systems. It provides a standard way to write audio plugins. The majority of the applications included in AGNULA/DeMuDi supports this format; since a number of high qualities plugins are available and non–compliant applications are "changing their mind", it's apparent how LADSPA is the "de facto" standard as far as audio plugins are concerned.

4 Applications

AGNULA/DeMuDi doesn't provide all the music/audio programs available for the GNU/Linux platform; the goal is to provide a thought–out selection of the "best" ones, allowing every kind of user to choose from consumer–grade to professional–grade applications. Even after the reduction process the original list underwent, the number of applications included in AGNULA/DeMuDi (100+) obliges us to present a restricted but representative overview. A complete list of the available applications is included either in the distribution itself or online[13].

[13]`http://www.agnula.org/Members/damien/List/view`

Sound Editors The choice of the sound editors included in AGNULA/DeMuDi illustrate the versatility of the distribution: it goes from the complex but extremely powerful *Snd* to the user friendly and straightforward *audacity* for the time domain. Frequency domain edition is possible with *ceres3*

Multitracker Considered as one of the major audio applications for GNU/Linux, *Ardour* is not only an excellent multitrack recorder but it also "caused" the development of *Jack*, as the author of these two programs, Paul Davis, originally developed *Jack* to fulfil a need he had for *Ardour*. *Ecasound* is a robust non-GUI alternative for multitrack recording.

Interactive Graphical Building Environments Free Software is very strong in this field with two well developed applications which have been enjoying a tremendous success for years: *jMax* and *Pure Data* (better known as *Pd*).

Sequencers Two sequencers amongst others are worth mentioning: *Rosegarden* and *Muse*. While originally they were pure midi–sequencers, now they both have some audio capabilities which turn them into complete musical production tools.

Sound Processing Languages A wide choice of compositional languages like *CSound*, *SuperCollider*, *Common Lisp Music* are available. It may be noticed that the first two were re–licensed under respectively the GNU LGPL (GNU Lesser General Public License) and the GNU GPL during the funded lifetime of the AGNULA project.

Software synthesizers A good range of software synthesizer is provided, including tools for modular synthesis (*AlsaModularSynth*, *SpiralSynthModular*); for additive and subtractive synthesis (*ZynAddSubFX*); and dedicated synthesis/compositional languages, such as *Csound* and *SuperCollider*.

Last, not least, *fluidsynth* and *TiMidity++* allow sample-based synthesis. In the attempt to distribute only Free Software, a Free GUS patch, *freepat* is also provided with *TiMidity++*. The patch is not complete (it still misses some instruments to cover the General Midi map) and this raised our perception that free content (like free samples or free loops) are a crucial need in order to provide a totally Free audio platform.

Notation The last category is particularly well represented with the professional–grade automated engraving system *Lilypond*. While *Lilypond* provides a descriptive language of the musical scores, it is also a back-end for different applications such as *Rosegarden* or the dedicated notation interface *NoteEdit*.

5 Prosecution after the ending of the funded phase

AGNULA/DeMuDi gave rise to a fairly large interest. Even after the ending of the funded phase, users' feedback has constantly increased as well as the requests for further enhancements. Being AGNULA/DeMuDi a Free Software project, these conditions naturally favoured its continuation.

As a matter of fact over the past months the distribution kept improving, and has now achieved most of its fundamental goals along with a certain degree of maturity and stability.

Nevertheless the project is probably encountering growth limits. At the moment the improvement of the distribution almost exclusively depends on the "centralised" effort of the AGNULA team.

As computer based audio and multimedia processing are very wide fields, a distribution aiming to embrace them all from different perspectives needs to actively involve different communities.

Time has come to exit the prototype and experimental phase and put A/DeMuDi in a wider picture.

Here follow some steps that the AGNULA team is going to undertake during the next months in order to entight the connection with all those projects/communites whose goals, spirit and people are closely related with A/DeMuDi.

5.1 Development infrastructure

The development infrastructure, currently hosted on the AGNULA server[14] will be moved to Alioth[15] and A/DeMuDi source packages will be tracked using the official Debian Subversion server.

Every major Custom Debian Distribution is already registered on Alioth[16]

, and having them all in a single place helps exchanging code, automate common services, spawn new CDDs.

Moreover all Debian developers are members of Alioth, and having a Debian-related project registered on Alioth makes it easier for the Debian community to notice and possibly join it.

5.2 Mailing lists

All user level discussions shall be carried directly on official Debian mailing list.

Probably due to historical reasons AGNULA/DeMuDi is now often perceived as a different distribution from or a derivation of Debian, as other popular projects [17]

This attitude somehow conflicts with the concept of Custom Debian Distribution and its advantages.

One of the goals of the AGNULA project was and is to improve the quality of Debian as far as audio and multimedia are concerned, and this effort will be carried directly inside Debian, with the work of Debian maintainers.

AGNULA/DeMuDi releases shall be considered as a comfortable way to install Debian for audio and multimedia work.

Every issue concerning AGNULA/DeMuDi is actually concerning Debian and it makes sense discuss it on Debian mailing lists, where one can get in touch and receive support from a much larger community than AGNULA.

These lists are of particular interest for the AGNULA community:

[14]`http://devel.agnula.org`

[15]`http://alioth.debian.org/projects/demudi`

[16]`https://alioth.debian.org/projects/debian-edu/`
`https://alioth.debian.org/projects/debian-np/`
`https://alioth.debian.org/projects/debian-br-cdd/`
`https://alioth.debian.org/projects/cdd/`

[17]`http://www.knoppix.net`
`http://www.morphix.org/modules/news/`
`http://www.progeny.com/`
`http://www.ubuntulinux.org/`

Debian-Multimedia [18] For audio and multimedia Debian-specific issues

Debian-User [19] For generic user support in English. Furthermore other dedicated mailing lists, as Debian-French, Debian-Italian, Debian-Spanish, Debian-Russian, Debian-Japanese, etc. [20] offer user support in various languages.

Moreover we encourage joining the Linux Audio Mailing list [21] for all discussions on using Linux for audio purposes.

5.3 Quality assurance

The AGNULA team is going to promote the birth of a quality assurance group dealing with audio and multimedia Debian packages.

While AGNULA/DeMuDi had a fairly large success among the users, creating an active community around the project, it is remarkable that, beside a few cases, the same thing did not happen with respect to the developers, who generally preferred to stick to Debian and coordinate themselves through the Debian-Multimedia group.

Debian-Multimedia is an official Debian sub-project started by Marco Trevisani (former technical manager of AGNULA/DeMuDi) whose goals are virtually identical to AGNULA/DeMuDi. The activity of the group is not as intense as AGNULA/DeMuDi, but it is constant in time, and has achieved some high quality results (e.g. good packaging for the JACK Audio Connection Kit). Currently Debian-Multimedia is simply a mailing list, and no web page has been yet published to describe the project, as it happened for other Debian groups [22]

The Debian-Multimedia sub-project not only represents the ideal door for AGNULA/DeMuDi to enter Debian, but can also be considered a reference point for other Debian based distributions dealing with audio and multimedia (e.g. Medialinux), and it would allow to gather the various efforts under the same hat.

Beside the tasks which Debian-Multimedia is already successfully carrying on, the group would:

- be a reference point for the audio/multimedia subset of Debian, assuring coherence and usability
- deal with a well defined set of packages
- provide bleeding-edge applications
- test packages and look for possible bugs
- discuss design and interface issues
- maintain a FAQ of the Debian-Multimedia mailing list

6 Conclusions

The AGNULA project, originally funded by the European Commission, is now continuing to pursue its goal of making Free Software the best choice for audio/video professionals on a volunteer/paid basis. The history of the AGNULA project, AGNULA/DeMuDi current status and its foreseeable future have been shown, as well as the general philosophy and technical beliefs that are behind the AGNULA team choices.

The AGNULA team does believe that a positive feedback loop has been spawned between Debian and the fast evolving domain of GNU/Linux audio applications. As a matter of fact a previously weak ring in the chain between audio professionals, musicians and composers on one side and Free Software developers on the other has been significantly strengthened.

This result can be considered the basis of a future adoption of Free Software tools by people who formerly had no alternative to proprietary software, along with all the implications of such a process in the educational, social, artistic and scientific fields.

7 Acknowledgements

As the reader may expect, projects such as AGNULA/DeMuDi are the result of the common effort of a very large pool of motivated people. And indeed, giving credit to any deserving individual that contributed to these projects would probably fill completely the space allotted for this paper. Therefore, we decided to make an arbitrarily small selection of those without whose help AGNULA/DeMuDi would not probably exist. We would like to thank,

[20] `http://lists.debian.org/debian-french/`
`http://lists.debian.org/debian-italian/`
`http://lists.debian.org/debian-user-spanish/`
`http://lists.debian.org/debian-user-portuguese/`
`http://lists.debian.org/debian-user-russian/`
`http://lists.debian.org/debian-japanese/`

[21] `http://www.linuxdj.com/audio/lad/subscribelau.php`

[22] `http://www.debian.org/devel/debian-desktop/`
`http://www.debian.org/devel/debian-installer/`
`http://www.debian.org/doc/ddp`
`http://www.debian.org/security/audit/`
`http://people.debian.org/ csmall/ipv6/`

Marco Trevisani, who has been pushing the envelope of a Free audio/music system for years, Dave Phillips, Günter Geiger, Fernando Lopez-Lezcano, François Déchelle and Davide Rocchesso: all these people have been working (and still work) on these concepts and ideas since the early days. Other people that deserve our gratitude are: Philippe Aigrain and Jean-François Junger, the European Commission officials that have been promoting the idea that AGNULA was a viable project against all odds inside the Commission itself; Luca Mantellassi and Giovanni Nebiolo, respectively President of Firenze's Chamber of Commerce and CEO of Firenze Tecnologia, for their support: they have understood the innovative potential of Free Software much better than many so-called open-source evangelists. Finally we wish to thank Roberto Bresin and the rest of the Department of Speech Music and Hearing (KTH, Stockholm), for kindly hosting the AGNULA server.

References

François Déchelle, Günter Geiger, and Dave Phillips. 2001. Demudi: The Debian Multimedia Distribution. In *Proceedings of the 2001 International Computer Music Conference*, San Francisco USA. ICMA.

Clark Williams. 2002. Linux scheduler latency. Technical report, Red Hat Inc.

SURVIVING ON PLANET CCRMA, TWO YEARS LATER AND STILL ALIVE

Fernando Lopez-Lezcano, nando@ccrma.stanford.edu
CCRMA
Stanford University
http://ccrma.stanford.edu/planetccrma/software/

ABSTRACT

Planet CCRMA at Home [2] is a collection of packages that you can add to a computer running RedHat 9 or Fedora Core 1, 2 or 3 to transform it into an audio workstation with a low-latency kernel, current ALSA audio drivers and a nice set of music, midi, audio and video applications. This presentation will outline the changes that have happened in the Planet over the past two years, focusing on the evolution of the linux kernel that is part of Planet CCRMA.

1. INTRODUCTION

Creating worlds is not an easy task, and Planet CCRMA is no exception. The last two years have seen a phenomenal expansion of the project. The history of it will reflect, I hope, part of the recent history of Linux Audio projects and kernel patching.

2. A BIT OF HISTORY

For those of you that are not familiar with Planet CCRMA [2] a bit of history is in order. At CCRMA (the Center for Computer Research in Music and Acoustics at Stanford University) we have been using Linux as a platform for research and music production since the end of 1996 or so. Besides the software available in the plain distribution I installed at the time, I started building and installing custom music software in our main server (disk space was not what it is today, and there were not that many Linux machines at that time, we were dual booting some PCs between Linux and NEXTSTEP, which was the main computing platform at CCRMA). I don't need to say that sound support for Linux in 1997 was a bit primitive. Not many sound cards were supported, and very few existed that had decent sound quality at all. Low latency was not a concern as just getting reliable sound output at all times was a bit of a challenge. Eventually the sound drivers evolved (we went through many transitions, OSS, ALSA 0.5 and then 0.9), and patches became available for the Linux kernel that enabled it to start working at low latencies suitable for realtime reliable audio work, so I started building custom monolithic kernels that incorporated those patches and all the drivers I needed for the hardware included in our machines (building monolithic kernels was much easier than trying to learn the details of loadable kernel modules :-).

But over time hard disks became bigger so that there was now more free space in the local disks, and the number of Linux machines kept growing, so the server installed software was going to become a network bottleneck.

Also, some adventurous CCRMA users started to install and try Linux in their home machines, and wanted an easy way to install the custom software available in all CCRMA workstations.

I was installing RedHat so I started to use RPM (the RedHat Package Manager) to package a few key applications that were used in teaching and research (for example the Snd sound editor, the CM - CLM - CMN Common Lisp based composition and synthesis environment, Pd and so on and so forth).

At first I just stored those packages in a network accessible directory and told potential users, "there you are, copy the packages from that directory and install them in your machine". A simple Web site with links to the packages was the next step, and installation instructions were added as I got feedback from users on problems they faced when trying to install the packages. Finally the project was made "public" with a post announcing it in the Cmdist mailing list - an email list for users of Snd and CM/CLM/CMN (although I later learned that some users had discovered the existence of the packages through search engines, and were already using them). The announcement happened on September 14th 2001. Time flies.

This changed the nature of the project. As more people outside of CCRMA started using the packages I started to get requests for packaging music software that I would not have thought of installing at CCRMA. The number of packages started to grow and this growth benefited both CCRMAlites and external Planet CCRMA users alike.

As the project (and this was never an "official" project, it was a side effect of me packaging software to install at CCRMA) grew bigger the need for a higher level package management solution became self-evident. The dreaded "dependency hell" of any package based distribution was a problem. More and more packages had external dependencies that had to be satisfied before installing them and that needed to be automatic for Planet CCRMA to be really usable. At the beginning of 2002 apt for rpm (a port of the Debian apt tool by Conectiva) was incorporated into Planet CCRMA, and used for all package installation and management. For the first time Planet CCRMA was reasonably easy to install by mere mortals (oh well, mere geek mortals).

Fast forward to today: there are more than 600 individual packages spanning many open source projects in each of the supported branches of RedHat/Fedora Core. You can follow the external manifestation of these changes over time by reading the online ChangeLog that I have maintained as part of the project (a boring read, to say the least).

3. AT THE CORE OF THE PLANET

Since the announcement of the project outside CCRMA on September 2001, the base distribution on which it was based (RedHat) has seen significant changes. In July 2003 RedHat stopped releasing commercial consumer products, and the last RedHat consumer version was 9, released on March 2003. The Fedora Project was created, with the aim of being a community driven distribution with a fast release cycle that would also serve as a testbed for new technologies for the enterprise line of RedHat products. Fedora Core 1 was the first release, followed by Fedora Core 2 and 3, at approximately 6 month intervals. The rapid release cycle plus the introduction of new technologies in the releases have made my life more “interesting”.

In particular, Fedora Core 2 saw the introduction of the 2.6 kernel, which created a big problem for rebuilding the Planet CCRMA package collection on top of it. The problem: a good, reliable low latency kernel did not exist. At that point in time 2.6 did not have an adequate low latency performance, despite the assurances heard during the 2.5 development cycle that new infrastructure in the kernel was going to make it possible to use a stock kernel for low latency tasks. Alas, that was not possible when Fedora Core 2 was released (May 2004).

4. THE KERNELS

Up to Fedora Core 1 the base distribution used a 2.4 kernel, and Planet CCRMA provided custom kernel packages patched with the well known low latency (by A. Morton) [6] and preemptible kernel (by R. Love) [5] patches (the last originally created by Monta Vista [4]), in addition to the tiny capabilities patch that enabled to run the Jack Audio Connection Kit server [15] and his friends with realtime privileges as non-root users.

Fedora Core 2 changed the equation with the introduction of the 2.6 kernel. Running a 2.4 kernel on top of the basic distribution presented enough (small) compatibility problems that I discarded the idea very early in my testing cycle. And 2.6 had a very poor latency behavior, at least in my tests. As a consequence until quite recently I still recommended using Fedora Core 1 for new Planet CCRMA installs.

For the first 2.6 kernels I tested (March 2004) I used a few additional patches by Takashi Iwai [7] that solved some of the worst latency problems. But the results were not very usable.

Ingo Molnar and Andrew Morton again attacked the problem and a very good solution evolved that is now available and widely used. Ingo started writing a series of patches for realtime preemption of the 2.6 kernel [8] (named at the beginning the “voluntary preemption” patchset). This set of patches evolved on top of the “mm” patches by Andrew Morton [9], the current equivalent of the old unstable kernel series (there is no 2.7 yet!, experimental kernel features first appear in the “mm” patches and then slowly migrate - the successful ones, that is - to the official release candidates and finally to the stable releases of the Linux kernel). Ingo did very aggressive things in his patches and the voluntary preemption patches (later renamed realtime preemption patches) were not the most stable thing to run in your computer, if it booted at all (while tracking successive releases I must have compiled and tried out more than 40 fully packaged kernels, for details just look at the changelog in the spec files of the Planet CCRMA 2.6 kernels). I finally released a preliminary set of kernel packages on December 24 2004, using version 0.7.33-04 of Ingo’s patches, one of the first releases that managed to boot in all my test machines :-)

What proved out to be interesting and effective in Ingo’s patches gradually percolated to the not so bleeding edge “mm” patches by Andrew Morton, and bits and pieces of “mm” gradually made it upstream to the release candidates and then to the stable kernel tree.

So, little by little the latency performace of the stock kernel improved. By the time of the release of 2.6.10 (December 24 2004 again just a coincidence) it was pretty good, although perhaps not as good as a fully patched 2.4 kernel. But keep in mind that this is the stock kernel with no additional patches, so the situation in that respect is much much better than it was in the old stock 2.4 kernel.

The end result for Planet CCRMA dwellers at the time of this writing are two sets of kernels, currently available on both Fedora Core 2 and 3.

4.1. The “stable” kernel

The current version is 2.6.10-2.1.ll. 2.6.10 turned out to be an unexpected (at least by me) milestone in terms of good low latency behavior. Finally, a stock kernel that has good low latency performance, out of the box. I would say it is close to what a fully patched 2.4 kernel could do before. The package also adds the realtime lsm kernel module, more on that later.

4.2. The “edge” kernel

Currently 2.6.10-0.6.rdt based on Ingo Molnar’s realtime preempt patch version 0.7.39-02. This is a more bleeding edge kernel, with significantly better low latency performance and based on Ingo Molnar’s realtime preemption patches. The downside of trying to run this kernel is that it still (at the time of this writing) does not work perfectly in all hardware configurations. But when it works, it works very well, and users have reported good performance with no xruns running with two buffers of 64 or even 32 samples! Amazing performance.

I'm still being a bit conservative in how I configure and build this kernel, as I'm not currently using the REALTIME_RT configuration option, but rather the REALTIME_DESKTOP option (thus the rdt in the release). The penalty in low latency behavior is worth the extra stability (at this time). I hope that the RT option (which gets the linux kernel close to being a "hard realtime" system) will evolve and become as stable as the REALTIME_DESKTOP configuration.

These packages also include the realtime lsm module.

4.3. Small details that matter

But a kernel with good low latency is not nearly enough. You have to be able to run, for example, Jack, from a normal non-root account. Enter Jack O'Quinn [10] and Torben Hohn. Their efforts created a kernel module, part of the kernel security infrastructure, that enables applications run sgid to a certain group, or run by users belonging to a group, or run by any user (all of this configurable, even at runtime), to have access to realtime privileges without having to be root. This is more restrictive and secure than the old capabilities patch, and at the time of this writing and after a very long discussion in the Linux Kernel mailing list (see [11] and [12]), has been incorporated into the "mm" kernel patches. Hopefully it will eventually percolate down to the stable kernel tree at some point in the future. It was a tough sell advocating for it in the Linux Kernel mailing list, many thanks to Jack O'Quinn, Lee Revell and others for leading that effort and to Ingo Molnar and Con Kolivas for proposing workable alternatives (that were later discarded). When the realtime patch becomes part of the standard kernel tree, a stock kernel will not only have decent low latency performance but will also work with software that needs realtime privileges like Jack does (including the ability of applications to run with elevated SCHED_FIFO scheduling privileges and to lock down memory so that it is not paged to disk).

But this was not enough for a Planet CCRMA release. Ingo Molnar's realtime preemption patch changed the behavior of interrupt requests, the lower half of the interrupt processes (if I understand correctly) are now individual processes with their own scheduling class and priorities, and a vital part of tuning a system for good low latency behavior is to give them, and Jack itself, the proper realtime priorities so that the soundcard and its associated processes have more priority than other processes and peripherals. I was trying to find a solution to this that did not involve users looking around /proc and tuning things by hand, when Rui Nuno Capela sent me a neat startup service script called rtirq that does just that, it sorts all interrupt service routines and assigns them decent priorities. Together with another small startup script I wrote that loads and configures the realtime lsm module, they make it possible to package an easy to install turn-key solution to a low latency 2.6 based kernel.

4.4. The core packages

The end result in Planet CCRMA are two sets of meta packages that reduce the installation and configuration of a 2.6 kernel to two apt-get invocations (installing planetccrma-core for the safer kernel and planetccrma-core-edge for the more risky one that offers better low latency performance).

This, coupled to the fact that due to 2.6 both Fedora Core 2 and 3 use ALSA by default, made installing Planet CCRMA is a much easier process when compared to Fedora Core 1 or RedHat 9 and their 2.4 kernels.

5. CONTINENTS AND ISLANDS

A large number of applications and supporting libraries have been added and updated over time to Planet CCRMA since 2003. Although not as many as I would like (just take a look at the "Pipeline" web page for packages waiting to be added to the repository). The list is almost too long but here it goes: seq24, filmgimp (later renamed to cinepaint), fluidsynth (formerly iiwusynth), the mcp ladspa plugins, hydrogen, rezound, cinelerra, mammut, csound, qarecord, qamix, qjackctl, gmorgan, ceres, pmidi, denemo, jackeq, cheesetracker, the rev ladspa plugins, qsynth, xmms-jack, jamin, vco ladspa plugins, pd externals (percolate, creb, cxc, chaos, flext, syncgrain, idelay, fluid, fftease, dyn), tap ladspa plugins, timemachine, caps ladspa plugins, xmms-ladspa, specimen, simsam, pvoc, brutefir, aeolus, fil ladspa plugins, pd vasp externals, jaaa, tap reverb editor, jackmix, coriander, liblo, jack bitscope, dvtitler, the soundtouch library, beast, phat, sooperlooper, qmidiarp, dssi. Sigh, and that's only new packages. Many many significant updates as well. Go to the Planet CCRMA web page for links to all these (and many other) fine software packages.

6. OTHER WORLDS

Planet CCRMA is one of many package repositories for the RPM based RedHat / Fedora family of distributions. Freshrpms [13], Dag [14], Atrpms, Dries and many others constitute a galaxy of web sites that provide easy to install software. Planet CCRMA is in the process of integrating with several of them (the so called RpmForge project) with the goal of being able to share spec files (the building blocks of RPM packages) between repositories. That will make my work, and that of the other packagers, easier, will reduce the inevitable redundancy of separate projects and will increase compatibility between repositories.

Another world in which I also want to integrate parts of Planet CCRMA is the Fedora Extras repository. This Fedora sponsored project opened its first CVS server a short while ago and will be a centralized and more official repository of packages, but probably exclusively dedicated to augmenting the latest Fedora Core release (as opposed to the more distribution agnostic RpmForge project). With the availability of Fedora Extras the “community” part of the Fedora Project is finally arriving and I’m looking forward to becoming a part of it.

7. PLANET FORGE

A short time ago I finally got all the remaining components, and finished building a new server here at CCRMA. It is a fast dual processor machine with a lot of memory and hard disk space completely dedicated to the Planet CCRMA project. The original goal was to create a fast build machine in which to queue packages to be rebuilt, as that process was fast becoming one of my main productivity bottlenecks in maintaining Planet CCRMA. A secondary, but no less important goal, is to try to create a collaborative environment in which more people could participate in the development and maintenance of Planet CCRMA packages and associated documentation. We’ll see what the future brings. A lot of work remains to be done to port my current build environment to the new machine and create a collaborative and more open environment.

8. FUTURE DIRECTIONS

One of the many things that are requested from time to time in the Planet CCRMA lists is the mythical “single media install” of Planet CCRMA (ie: “do I have to download all these cdroms?”). In its current form (and on purpose), a potential user of Planet CCRMA has to first install Fedora Core, and then add the kernel, drivers and packages that make up Planet CCRMA (this additional installation and configuration work has been substantially reduced in Fedora Core 2 and 3 releases as they use ALSA by default instead of OSS). While this is not that hard, specially with the help of meta packages and apt-get or synaptic, it appears that sometimes it is too much work :-) And I have to agree, it would be much nicer to have a single cd (hmm, actually a dvd given the size of current distributions) and at the end of the install have everything ready to go, low latency kernel active, just start the applications and make some music. I have long avoided going down this road and becoming a “distribution” because of the additional work that would involve. It is hard enough trying to keep up to date with the very fast evolution of Linux audio software.

But on and off I’ve been thinking about this idea, and lately I’ve been actually doing something about it. At the time of this writing (end of February 2005) I already have a single “proof of concept” dvd with everything in it, all of Fedora Core 2 - the distro I’ve been playing with, I obviously have to do this on Fedora Core 3 as well - plus all of Planet CCRMA. This test dvd is not small, about 3G of stuff, remember, all of Fedora Core is included!

Installing Planet CCRMA from it entails booting into the dvd, selecting the Planet CCRMA installation target, customizing the packages installed if desired and pressing “Install” (while going through the normal installation choices of a stock Fedora Core system install, of course). One reboot and you are up and running. Furthermore, the dvd creation process is pretty much automatic at this point (start a series of scripts, wait for some time and out comes a dvd iso image).

Of course things are not that easy. What kernel should I select for installation? The more stable or the more risky that has better latency performance? How will the idiosyncracies of this non-standard kernel interact with the Fedora Core install process? (for example, it may happen that it will fail to boot in some machines, while the original Fedora Core kernel would have succeeded - and I don’t think Anaconda, the RedHat installer, would be able to deal with more than one kernel at install time). Hopefully some or all of these questions will have answers by the time I attend LAC2005, and conference attendees will be able to test drive an “official” alpha release of Planet CCRMA, the distro (another question to be answered: why do I keep getting into a deeper pit of support and maintenance stuff??).

9. CONCLUSION

It is easy to conclude that Planet CCRMA is very cool. More seriously. Planet CCRMA as a project is alive and well. As a maintainer I’m (barely) alive, but have made it to another conference, no small feat.

10. ACKNOWLEDGEMENTS

The Planet CCRMA project would never have been possible without the support for GNU/Linux and Open Source at CCRMA, Stanford University, and in particular, the support of to Chris Chafe, CCRMA’s Director. It goes without saying that I extend my heartfelt thanks to the hundreds of commited developers whose software projects I package. Without them Planet CCRMA would not exist and I would live in much more boring world.

11. REFERENCES

[1] The Fedora Project. http://fedora.redhat.com/

[2] The Planet CCRMA Project. http://ccrma.stanford.edu/planetccrma/software/

[3] Ingo Molnar: Low latency patches for 2.2/2.4. http://people.redhat.com/mingo/lowlatency-patches/

[4] MontaVista: The Preemptible Kernel Patch. http://www.mvista.com/, see also http://www.linuxdevices.com/news/NS7572420206.html

[5] Robert Love: The Preemptible Kernel Patch. http://rlove.org/linux/

[6] Andrew Morton: Low latency patches for 2.4. http://www.zip.com.au/ akpm/linux/schedlat.html

[7] Takashi Iwai: low latency tweaks. http://kerneltrap.org/node/view/2702

[8] Ingo Molnar: Realtime Preemption patches for 2.6. http://people.redhat.com/mingo/realtime-preempt/

[9] Andrew Morton: the “mm” patches for 2.6. http://kernel.org/pub/linux/kernel/people/akpm/patches/2.6/

[10] Jack O’Quinn: the realtime lsm kernel module. http://sourceforge.net/projects/realtime-lsm/

[11] Linux Weekly News: Merging the realtime security module. http://lwn.net/Articles/118785/

[12] Weekly News: Low latency for Audio Applications. http://lwn.net/Articles/120797/

[13] Freshrpms: package repository. http://freshrpms.net/

[14] Dag: package repository. http://dag.wieers.com/home-made/apt/

[15] The Jack Audio Connection Kit, a low latency sound server. http://jackit.sf.net/

Linux As A Text-Based Studio
Ecasound – Recording Tool Of Choice

Julien CLAASSEN
Abtsbrede 47a
33098 Paderborn
Germany
julien@c-lab.de

Abstract

This talk could also be called "ecasound textbased harddisk recording". I am going to demonstrate a few of the most important features of ecasound and how to make good use of them in music recording and production.

This talk explains what ecasound is and what its advantages are, how a braille display works, Ecasound's basic features (playback, recording, effects and controllers), and a few of ecasound's more advanced features (real multitrack recording and playback and mastering).

Keywords

audio, console, recording, text-based

1 What is Ecasound?

1.1 Introduction to Ecasound

Ecasound is a textbased harddisk recording, effects-processing and mixing tool. Basically it can operate in two ways:

- It can work as a commandline utility. Many of its features can be used from the commandline, via a whole lot of options.
- It can also be operated from a shell-like interface. This interface accepts its own set of commands, as well as commandline options.

Ecasound supports more than your usual audio io modes:

- ALSA - Advanced Linux Sound Architecture
- Jack - Jack Audio Connection Kit
- ESD - Enlightenment Sound Daemon
- Oldstyle OSS - Open Sound System
- Arts - the Arts Sound Daemon

1.2 Advantages

1. Ecasound can easily be used in shell-scripts through its commandline options. Thus it can perform some clever processing.
2. Through its shell-interface you can access realtime controls. Via its set and get commands one can change and display controller values.
3. Because ecasound does not require an X-server and a lot of other GUI overhead, it is slim and fast. On a 700 MHz processor one can run an audio-server (JACK), a software synthesizer (fluidsynth) and ecasound with 3 or more tracks without problems.
4. Ecasound is totally accessible for blind people through its various textbased interfaces. Those interfaces provide full functionality!

1.3 Disadvantages

1. Its textbased interface is not as intuitive and easy to learn as a GUI for a sighted person .
2. Its audio routing capabilities still lack certain features known to some other big linux audio tools.
3. It does not provide much support for MIDI (only ALSA rawmidi for controlling effects and starting/stopping).

2 How I Work

I work with a braille display. A braille display can display 40 or 80 characters of a screen. In textmode this is a half or full line.

The braille display has navigation buttons, so you can move the focus over the whole screen, without moving the actual cursor. Usually the display tracks the cursor movement, which is very useful most of the time. For the rest of the time, you can deactivate tracking of the cursor.

So the best programs to use are line-oriented. Thus tools with shell-interfaces or commandline utilities are the best tools for me.

N.B.: As I heard such tools are also among the top suspects for users of speech-synthesizers.

3 Usage

This chapter will give several use cases of ecasound.

3.1 Using ecasound from the command line

As already stated ecasound can – in general – be used in two ways: From the commandline and from its interactive mode. The following examples will deal with ecasound's commandline mode.

3.1.1 Playing files from the commandline

One of the simplest uses of ecasound is playing a file from the commandline. It can look like calling any simple player – like i.e. aplay. If the ecasound configuration is adjusted correctly it looks like this:

```
ecasound myfile.wav
```

or

```
ecasound -i myfile.wav
```

The "-i" option stands for input. If you wish to specify your output explicitly and do not want to rely on the ecasoundrc configuration file, you can do it like that:

```
ecasound -i myfile.wav -o alsa,p1
```

`alsa,p1` marks the alsa output on my system-configuration running ALSA. The "-o" option means output.

3.1.2 Recording files from the commandline

It is as simple as playing files. The only thing one needs to exchange is the place of the sound-device (ALSA device) and the file (`myrecording.wav`). So if one intends to record from an ALSA device called "io1" to `myrecording.wav`, one would do it like that:

```
ecasound -i alsa,io1 -o
myrecording.wav
```

It looks just like the example from section 3.1.1 with sound-objects exchanged.

3.2 Interactive mode

Ecasound interactive mode offers a lot more realtime control over the things you mean to do like starting, stopping, skipping forward or backward etc. Thus in most cases it is more suited to the needs of a recording musician. Below there are some simple examples.

3.3 Playing a file

This method of playing a file is much closer to what one could expect of a nice player. The syntax for starting ecasound is very similar to the one from 3.1.1.

```
ecasound -c -i myfile.wav [-o
alsa,p1]
```

By pressing "h" on the ecasound shell prompt you get some basic help. For more info – when trying it at home, there is the ecasound-iam (InterActive Mode) manual page.

3.4 Interactive recording

The simplest way to record a file is almost as simple as playing a file. The only thing is you have to specify the audio-input source. Btw.: The same syntax can be used to convert files between different formats (wave-file, mp3, ogg, raw audio data...).

To do a simple interactive recording, type this:

```
ecasound -c -i alsa,io1 -o
myrecording.wav
```

Again you have the interactive capabilities of ecasound to support your efforts and extend your possibilities. Besides that, it is the same as in paragraph 3.1.2.

3.5 Effects in ecasound

Ecasound has two sources for effects: internal and external via LADSPA. In the following sections both are introduced with a few examples and explanations.

3.5.1 Internal effects

Ecasound comes with a substantial set of internal effects. There are filters, reverb, chorus, flanger, phaser, etc. All effect-options start with "e", which is good to know when looking for them in the manual pages. Here is a demo of using a simple lowpass filter on a wave-audio file:

```
ecasound -i myfile.wav -efl:1000
```

which performs a lowpass filter with a cutoff frequency of 1000Hz on the file `myfile.wav` and outputs the result to the default audio device.

3.5.2 External / LADSPA effects

Ecasound can also use LADSPA effects which makes it a very good companion in the process of producing and mastering your pieces.

There are two different ways of addressing LADSPA effects: By name or by unique ID.

3.5.3 Addressing by name

With `analyseplugin` you can determine the name of a LADSPA effect like:

```
babel:/usr/local/lib/ladspa #
  analyseplugin ./decimator_1202.so

Plugin Name: "Decimator"
Plugin Label: "decimator"
Plugin Unique ID: 1202
Maker: "Steve Harris "
Copyright: "GPL"
Must Run Real-Time: No
Has activate() Function: No
Has deativate() Function: No
Has run_adding() Function: Yes
Environment: Normal
Ports:  "Bit depth" input, control, 1 to
  24, default 24
        "Sample rate (Hz)" input, control,
  0.001*srate to 1*srate, default 1*srate
"Input" input, audio, -1 to 1
"Output" output, audio, -1 to 1
```

Thus one knows that "decimator" is the name – label – of the plugin stored in decimator_1202.so. Now you can use it like that:

```
  ecasound -i file.wav
-el:decimator,16,22050
```

which simulates the resampling of the file "file.wav" at 22.05 KHz.

3.5.4 Addressing by unique ID

`analyseplugin` not only outputs the label of a LADSPA plugin, but also its unique ID, which ecasound can also use. Mostly this way is simpler, because there is less to type and you do not have to look for upper- and lowercase letters. With the following command you can use the decimator plugin by its unique ID:

```
  ecasound -i file.wav
-eli:1202,16,22050
```

This command does the same as the one before.

Although it looks more cryptic to the naked eye, it is really shorter and (once you are used to it) much simpler to type – this is at least my personal experience.

3.5.5 Effect presets

Another powerful feature of ecasound are effect presets. Those presets are stored in a simple text-file, usually `/usr/local/share/ecasound/effect_presets`. An effect preset can consist of one or more effects in series, with constant and variable parameters. What does this mean in practice? The following illustrates the use of the metronome-effect:

```
ecasound -c -i null -pn:metronome,120
```

This provides a simple clicktrack at 120 BPM. Internally the ecasound "metronome" effect-preset consists of a sinewave at a specific frequency, a filter – for some reason – and a pulse gate. This gate closes at a certain frequency given in BPM. Would you use all those effects on the commandline directly, you would have to type a lot. Besides getting typos, you could also choose very inconvenient settings. If you use the effect preset, everything is adjusted for you.

The standard preset file contains a good collection to start with. From simple examples for learning, to useful things like a wahwah, metronome, special filter constellations, etc...

3.5.6 Controllers

Ecasound also offers a few controllers which you can use to change effect parameters while your music is playing. The simplest controller is a two-point envelope. This envelope starts at a given start value and moves over a period of time to a certain endvalue. In practice it could look like this: A user wants to fade in a track from volume 0 to 100 over 4 seconds:

```
  ecasound -i file.wav -ea:100
-kl:1,0,100,4
```

What does the first parameter of `-kl` mean? This parameter is the same for all `-k*` – controller – options. It marks the parameter you want to change. The amplifier (`-ea`) has only one parameter: the volume. Thus the first parameter is 1. The second is the start value (0), meaning the volume should start at 0, the third value is the endvalue for the envelope: Volume should go up to 100. The last value is the time in seconds that the envelope should use to move from start to end value.

Ecasound offers more controllers than this simple one. It has a sine oscillator and generic oscillators which can be stored in a file like effect presets. Besides that you can use MIDI controllers to do some **really** customised realtime controlling.

3.5.7 An interactive recording with realtime control

Now a short demonstration of the features presented so far: A short and simple recording with some realtime-controlled effects.

The scenario is: One synthesizer recorded with ecasound and processed by a lowpass filter which is modulated by a sinewave. This will generate a simple wahwah effect. It might look like this:

```
  ecasound -c -i jack_auto,fluidsynth
-o my_file.wav -ef3:5000,0.7,1.0
-kos:1,800,5000,0.5,0
```

The `-ef3` effect is a resonant lowpass filter with these parameters: Cutoff frequency in Hz, resonance – from 0 to 1 (usually) – and gain. Values for gain should be between 0 and 1. The `-kos` controller is a simple sine oscillator with the following parameters:

1. effect-parameter – parameter of the effect to modify (first parameter of `-ef3` – the cutoff)
2. start-value – lowest value for the cutoff frequency
3. end-value – highest value for the cutoff
4. frequency in Hz – the frequency at which the cutoff should change from lowest to highest values – in this case 0.5 Hz. It takes 2 seconds.
5. iphase – initial phase of the oscillator. A sinus starts at 0 and moves upwards from there. Yet one can shift the wave to the right by supplying an iphase > 0.

4 More complex work

This chapter gives some more complex usage examples of ecasound.

4.1 Chains

4.1.1 What is a chain?

A chain is a simple connection of audio objects. A chain usually contains of:

- an audio input
- effects (optional)
- an audio output

You have already seen chains, without really knowing them because even a simple thing like:

```
  ecasound -i file.wav
```

uses a chain with the default output.

To explicitly specify a chain, you need to use the -a option. The above example with an explicit chain-naming, yet still unchanged behaviour looks like that:

```
  ecasound -a:my_first_chain -i
file.wav (-o alsa,p1)
```

4.1.2 What is a chain setup?

A chain setup can be seen as a map of all chains used in a session. You can perhaps imagine that you can have parallel chains – for mixing audiotracks – or even more complex structures for tedious mastering and effects processing. You can store a complete chain setup in a file. This is very useful while mastering pieces.

A simple example of an implicit chain setup includes all above examples. They have been chain setups with only one chain. To store chain setups in files you can use the interactive command cs-save-as or cs-save, if you've modified an existing **explicit** chain setup.

4.2 Playing back multiple files at once

Now the user can play back a multitrack recording before having generated the actual output-mixdown.

It could look like this:

```
  ecasound -c -a:1 -i track1.wav -a:2 i
track2.wav -a:3 -i track3.wav -a:1,2,3
-o alsa,p1
```

This also demonstrates another nice simplification: One can write something like `-a:1,2,3` to say that chain 1, 2 and 3 should have something in common. In this example it could be even shorter:

```
  ecasound -c -a:1 -i track1.wav -a:2
-i track2.wav -a:3 -i track3.wav -a:all
-o alsa,p1
```

This line does exactly the same as the last demo. The keyword `all` tells ecasound to apply the following options to **all** chains ever mentioned on the commandline.

4.3 Recording to a clicktrack

Now one can use chains to perform an earlier recording to a clicktrack:

```
ecasound -c -a:1,2 -i alsa,io1
-a:1 -o track1.wav -a:3 -i null
-pn:metronome,120 -a:2,3 -o alsa,p1
```

This does look confusing at first sight, but is not. There are three chains in total. Chain 1 and 2 get input from the soundcard (`alsa,p1`), chain three gets null input (null). Chain 2 (soundcard) and 3 (metronome) output to the soundcard so you hear what is happening. Chain 1 outputs to a file. Now you can use `track1.wav` as a monitor and your next track might be recorded with a line like this:

```
ecasound -c -a:1,2 -i alsa,io1
-a:1 -o track2.wav -a:3 -i null
-pn:metronome,120 -a:4 -i track1.wav
-a:2,3,4 -o alsa,p1
```

This extends the earlier example only by a chain with `track1.wav` as input and soundcard (`alsa,p1`) as output. Thus you hear the click-track – as a good guidance for accurate playing – and the first track as a monitor.

4.4 Mixing down a multitrack session

Having several tracks on harddisk, the mixdown is due. First one can take a listen to the multi-track session and then store the result to a file.

Listening to the multitrack can be achieved by issuing the following command:

```
ecasound -c -a:1 -i t1.wav -a:2 -i
t2.wav -a:3 -i t3.wav -a:all -o alsa,p1
```

Now adjusting of volumes can be managed by applying `-ea` (amplifier effect) to each track. i.e.:

```
ecasound -c -a:1 -i t1.wav -ea:220
-a:2 -i t2.wav -ea:150 -a:3 -i t3.wav
-a:180 -a:all -o alsa,p1
```

This amplifies t1.wav by 220%, t2.wav by 150% and t3.wav by 180%.

Being content with volume adjustment and possibly other effects, the only thing left is exchanging soundcard output by file-output. Meaning exchange `alsa,p1` with `my_output.wav`:

```
ecasound -c -a:1 -i t1.wav -ea:220
-a:2 -i t2.wav -ea:150 -a:3 -i t3.wav
-ea:180 -a:all -o my_output.wav
```

Now ecasound will process the files and store the mixdown to disk. The last – optional – step is to normalize the file `my_output.wav` which can be performed by `ecanormalize`:

```
ecanormalize my_output.wav
```

The normalized output file overwrites the original: So **be careful**!

5 Resume

Having in theory produced a piece ready for burning on CD or uploading to the Internet, here comes the resume. It is not the same way you would do it in a graphical environment, yet it still works fine!

For me ecasound is always the tool of choice. It is a very flexible tool. Its two general modes – commandline and interactive – combined with its chain-concept make it a powerful recording and mixing program. Because ecasound has LADSPA support and can be connected to the JACK audio server it is very simple to integrate it in a linux-audio environment. You can also use it in combination with graphical tools, if you so choose.

So for those who love text interfaces, need fast and simple solutions or those who start to learn about audio-recording, ecasound can be a tool of great value.

Besides that, ecasound is of course a very good example of what free software development can do: Produce a very up-to-date piece of fine software which is fully accessible to blind and visually impaired people. Yet still it was not written with this audience in mind. There is a fairly large crowd relying on ecasound for very different kinds of work. Though it lacks a few things that others have, it is not said that ecasound can never get there. Meanwhile there are other ways to achieve what one needs to achieve, thanks to the flexibility of ecasound and the tools you can combine/connect with it.

6 Thanks and Acknowledgements

Thanks and acknowledgements go to:

- Kai Vehmanen and the ecasound crew at http://www.eca.cx/ecasound
- Of course the ALSA crew with much work from Takashi Iwai and Jaroslav Kysela at http://www.alsa-project.org
- Richard E. Furse and companions, at www.ladspa.org, for creating LADSPA in the first place
- Steve Harris and his wonderful collection of LADSPA plugins at http://plugin.org.uk
- Paul Davis and friends at http://jackit.sf.net for jackd, our favourite realtime audio server

- http://www.fluidsynth.org, namely Josh Green, Peter Hanappe and colleagues for the soundfont-based softsynth fluidsynth
- Dave Phillips and his great collection of MIDI and audio links at http://linux-sound.org
- ZKM for hosting this conference, see the official LAC webpages at http://www.zkm.de/lac

Before thanking the great bunch of people who organised and host this event, I want to mention my own webpage at `http://ltsb.sourceforge.net`.

Great many thanks to Frank Neumann, Matthias Nagorni, Götz Dipper and ZKM for organising and hosting this conference! And many thanks and apologies to all those I forgot! Sorry to you, I didn't mean to!

"terminal rasa" - every music begins with silence

Frank EICKHOFF
Media-Art, University of Arts and Design
Lorenz 15
76135 Karlsruhe,
Germany,
feickhof@hfg-karlsruhe.de

Abstract

An important question in software development is: How can the user interact with the software? What is the concept of the interface? You can analyze the "interface problem" from two perspectives. One is the perspective of the software developer. He knows the main features of the software. From that point he can decide what the interface should look like, or which areas should be open for access. On the other side is the perspective of the user. The user wants an interface with special features.

The questions for audio software designed for live performance are: What should the program sound like? If software for live performance should have features like an instrument, what features does an acoustic instrument have, and what features should a computer music instrument have? The first part of this paper is concerned with music and sound, the special attributes of acoustic instruments, and the features of an average Personal Computer. The second part of the paper presents the audio software project "fui" as a solution to the aforementioned questions and problems.

Keywords

computer music instrument, interface problem

1 Introduction

The "interface problem" is a very important aspect in audio software development. The interface of a machine is not the device itself, but rather the parts of the machine which are used for interaction and exchange; the area between the inner and the outer world of the machine. The "interface problem" is the problem of interaction between human and machine. For an instrument, it is the area between sound production and sound modulation. Sound is produced through specific methods or physical phenomena. These methods of sound production can be controlled through the manipulation of various parameters. These parameters are the values which should be open for access by the user. It is possible to draw conclusions from the analysis of sound to the possibilities of sound modulation, which is interaction. Therefore, the first item to consider is music.

2 Computer Music Instrument? Music - Instrument - Computer

Silence, noise, and sound are the most basic elements of the phenomenon that is music. What music one wants to hear is an individual decision. Each has his own likes and dislikes. In the first place, music is a matter of taste. An instrument (acoustic or electronic) is a tool or a machine to make music. Any of these tools are designed with a special intention. The basis of this intention is a certain idea of sound and timbre. One can say that the instrument is a mechanical or electronical construcion of a sound idea. What should my instrument sound like? How can I construct this sound?

2.1 Acoustic Sound

Sound is nothing more than pressure differences in the air. One can hear sound, but one can not easily see it or touch it. The behavior of sound in a space is complex and depends on the physical properties of the space. Thus, any visual representation of sound must remain abstract, and is necessarily a simplified model of the real situation. The special character of sound is that one can NOT see it.

2.2 Digital Sound

A computer calculates a chain of numbers which can be played by a soundcard. Acoustic waves are simulated by combinations of algorithms. Such mathematical processes are abstract and not visible. An audio application can run within a shell process or even as a background process. It does not require any visual or even statistical feedback.

2.3 Instrument = Sound + Interface

At the point where one wants direct access to the sound manipulating parameters of his software or instrument, one needs some sort of an interface. The construction of the interface is derived on one hand from the timbre of the sound. On the other hand, the interface has influence on the playability of the instrument and, thus, on the sound aesthetic. The instrument is the connection of sound with an interface.

2.3.1 Classic, Acoustic Instruments

A classical instrument like the violin or the piano is very old compared to the computer. The structure and operation of acoustic instruments has been optimized through years of usage. One could say, then, that the instrument has a balance between optimized playability and a characteristic tone colour / timbre. Every instrument has its own unique sound.

2.3.2 Universal Computer

From the start the computer was developed as a universal flexible calculating machine. The "universal computer" works with calculating operations and algorithms. Alan Turing proved with his invention of the "Turing machine" that every problem which could be mechanized can be solved by a computer calculation[1]. Otherwise the computer ends up in an infinite loop and without result. The "Turing machine" does not stop. It is obvious that the computer can solve a huge amount of problems.

The computer interface is divided into hardware and software interface. The hardware setup of an ordinary personal computer is a keyboard, a monitor and a mouse. Software interfaces are programs which can interact with such hardware. The clarification of this concept makes it easy to deal with the complex possibilities of the computer.

2.4 Computer Music Instrument

When one wants to use the computer as an instrument, one must combine the features of an instrument with the features of a computer. One needs to create a balance between playability, unique sound and the special character of the computer, that is flexibility:

$\boxed{\text{Sound vs Playability vs Flexibility}}$

3 The "fui" Audio Application

The audio application "fui" is a sample loop sequencer. The program is designed for "live performance", as such it is playable like an instrument. It is a simple tool to create short rhythmic loops. It has a minimal sound characteristic and serial or linear rhythmic aesthetic. The user has two different interfaces. One is a terminal for keyboard commands. The other is a graphic window with a GUI (Graphical User Interface) for the interaction with the mouse.

[1]"On computable Numbers and an Application to the Entscheidungsproblem", Alan Turing, 1947

3.1 Short Description

The user can load audio samples into a sequence. Such sequences are played in a loop directly. He can move such samples to a specific point in time within a sequence. Samples are dragged and moved multiple times until the music gets interesting. With this method it is easy to construct rhythmic patterns. Every sample can be modulated through the control of different parameters (Filter, TimePitch, PitchShift, Incremental or Decremental Loop). It is possible to create multiple sequences, and to switch between them in a song like manner. Because of the playback of the loops, the user gets a direct response to all the changes he or she makes. The music develops through improvising and listening.

3.1.1 Sound Effects

Every sample can be modulated with different effects. The effect parameters are both static and random. The "pitch" control allows the user to manipulate the pitch of the sample. The "position" is the playback start value within a sample. "Loop" restarts the sample at the end and "count" is the number of repeats. "Incremental loop" or "decremental loop" starts the sample at the "position" point, and the sample length gets shorter or longer after every repeat.

3.2 Interaction - Interface

The "fui" application uses the standard interfaces of an ordinary computer. Every interface has its advantages and disadvantages. The terminal program is specialized on keyboard control. The GUI is specialized on mouse control. The "fui" application uses both features (see Figure 1).

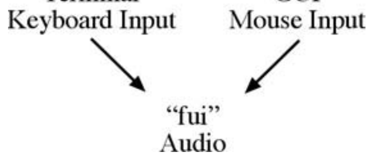


Figure 1: "fui" Interface

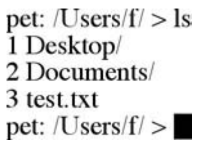
3.2.1 Terminal

Before the invention of the desktop computer with GUI control there was only a terminal. The terminal is one of the oldest software interfaces to the processes of the computer. The terminal works perfectly as an interface because it is incorporated on many operating systems. It operates simply on keyboard input and text output. It is difficult to implement a comparable interface in a mouse orientated GUI. When there is a terminal anyway, why shouldn't we use it?

3.2.2 "pet" - Pseudo emulated Terminal

The first thing which is launched by "fui" is "pet" (pseudo emulated terminal). The idea behind "pet" is to use the terminal as a keyboard input and text output interface during the runtime of the program. This object uses the standard streams (stdout, stdin) for reading and writing. The user can type in simple UNIX like commands (see Table 1) for file browsing, changing the working directory, loading files into the "fui" software.

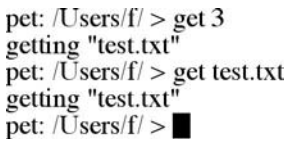
The "ls" command prints out a list of the current directory. The "pet" object numbers all files and folders in the directory (see Figure 2).



```
pet: /Users/f/ > ls
1 Desktop/
2 Documents/
3 test.txt
pet: /Users/f/ > █
```

Figure 2: list Directory

The "get" command loads a filename into the "pet" command parser. The command argument is the filename or the number printed out with "ls" (see Figure 3). Some other commands like "cd" use the same method of file identification. This method provides a simple and fast way to load files or browse directories.



```
pet: /Users/f/ > get 3
getting "test.txt"
pet: /Users/f/ > get test.txt
getting "test.txt"
pet: /Users/f/ > █
```

Figure 3: get Filename

3.2.3 "pet" and "fui"

The "fui" software uses "pet" file loading and file browsing. The "pet" object numbers all file in a directory chronologically. When the user loads a sample or creates a new sequence "fui" creates index numbers. The sample-ID is "ID" and the sequence-ID is "SID". For example, when the user wants to call a specific sample he has to know the sample-ID. The command "la" prints out a list with all sample filenames, information about position, pitch and IDs of the current sequence.

cd PATH or NUM	change directory
ls	list directory, files are numbered
start	start audio
stop	stop audio
open	open GUI
load 'name'	load sequence
save 'name'	save sequence
new	new sequence, generates SID
dels	delete sequence
la	list all samples, with ID and SID
get NAME or NUM	load sample
del ID	delete sample
seq SID	set current sequence
loop TIME	set loop time (ms)
loff ID	loop off
lon ID	infinite loop on
lr ID	random loop
ld ID POS NUM	decremental loop
li ID POS NUM	incremental loop
pi ID VALUE	set pitch value

Table 1: "pet" Commands

3.2.4 Graphic User Interface

Sometimes the possibility of visualization, graphical feedback of statistical values, or interaction is very useful. The "fui" GUI is rendered in OpenGL and has a very simple design. There are text buttons (strings which function like a button), text strings without any interactivity and variable numbers to adjust parameters with the mouse. Every control is listed in a simple menu (see Figure 4). Some text buttons have multiple states. Active GUI elements are rendered in black, inactive elements are grey (see Figure 5).

A vertical, dotted line is the loop cursor. The cursor changes the position from left to right, analog to the current time position of the loop. Audio samples are drawn as rectangles (see Figure 6).

The width of the rectangle is proportional to the length of the sample and the length of the

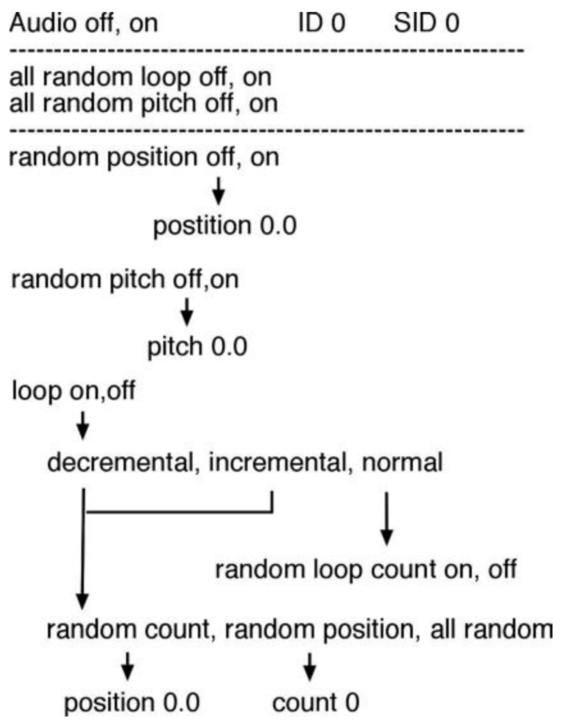


Figure 4: GUI Menu

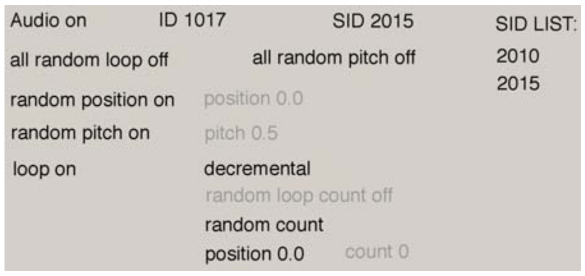


Figure 5: GUI

sample loop in seconds which is the width of the window. The sample can be moved with the mouse within a two dimensional area (like a desktop, table or ”tabula”).

Every new sequence has a blank area, a blank table (”tabula rasa”).

3.3 Example Usage

Every music begins with silence. ”fui” starts as a simple terminal application without any other window or GUI (”terminal rasa”). After startup the software waits for command line input (see Figure 7).

The ”open” command opens the GUI window. The ”new” command creates a new, empty sequence. ”fui” adds a new number to the ”SID LIST” in the GUI window. This number is the new ID for the current sequence. The ”start” command starts the audio playback. The loop cursor starts moving over the

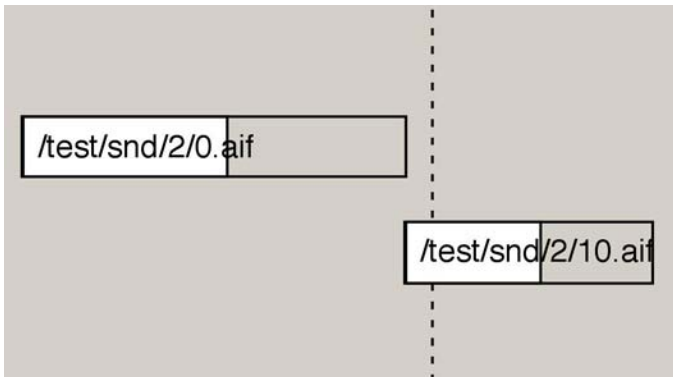
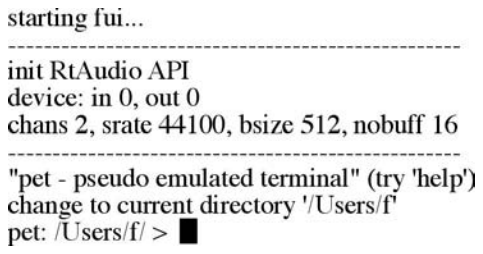


Figure 6: Vertical Cursor and two Samples



```
starting fui...
--------------------------------------------------
init RtAudio API
device: in 0, out 0
chans 2, srate 44100, bsize 512, nobuff 16
--------------------------------------------------
"pet - pseudo emulated terminal" (try 'help')
change to current directory '/Users/f'
pet: /Users/f/ > █
```

Figure 7: Start Screen - ”terminal rasa”

window. Now, the user can browse the harddisk for suitable samples. The ”get” command loads a sample into the sequence. The user can move the sample to a position within the GUI window. Every time the cursor reaches the sample, the sample will be played (see Figure 8).

3.4 Sound - Playability - Flexibility

Sound, playability and flexibility have a mutual influence on each other. The sound is determined by the implementation of audio playback and audio manipulation. Many interesting rhythms can be found by improvising and playing with the software. Different interfaces and the implementation of a powerful audio engine enhance the flexibility of ”fui”.

3.4.1 Sound

The characteristic ”fui” sound comes from the combination of short samples into rhythmic loops. All samples are freely arranged within the time frame of one sequence. There are no restrictions imposed by time grids or ”bpm” (beats per minute) tempo parameters. The user has a simple visualization and a direct audio playback.

3.4.2 Playability

The use of the UNIX like terminal and the simple GUI provide a simple and playful access to the software. Different sound effects with static or random modulation vary the sound. All changes are made intuitively by the user through listening. For example, a combination of two samples which might sound boring at first, can become very interesting with slight changes to the position of one sample within the time frame of the loop. A simple change of one parameter can have an interesting result in the music.

3.4.3 Flexibility - Audio API?

In the first place the source code should be portable. This project was developed on an Apple Macintosh PISMO, G3, 500 Mhz, OSX 10.3 using the gcc compiler. Later it was ported to Linux. The whole project was written in ANSI C/C++ with OpenGL for graphic rendering. The "Software Toolkit"[2] from Perry Cook and Gary Scavone is used for realtime audio file streaming. The platform independent window class from "plib"[3] is used for creating the render window.

Different audio engines are tested for the main audio streaming:

"RtAudio"[4] from Gary Scavone, "Portaudio"[5] from Ross Bencina, "FMOD"[6] from Firelight Technologies and "JACK"[7] from Paul Davis "and others".

The "JACK" API works as a sound server within the operating system. Completely different audio applications, which are compiled as "JACK" clients, can share audio streams with each other. Now the developer does not need to think about implementing some kind of plug-in architecture in the software. Audio streams can easily be shared in a routing program. It is simply perfect for audio software developers. From that point the use of the "JACK" API is the most flexible solution for the "fui" audio project.

4 Conclusions

Music is meant to be listened to. The idea of "fui" is to establish a balance between the interface and the characteristic sound of the computer as a musical instrument. When one is familiar with the special features and the historical background of acoustic instruments and computers in music AND the general differences between the two, it is possible to say that the ideal combination of both media is a hybrid and open environment. The design of the interface is simple, minimal and experimental. The sound aesthetic is linear with nested rhythmic patterns. The user deals with the program in a playful way and the music is created through listening.

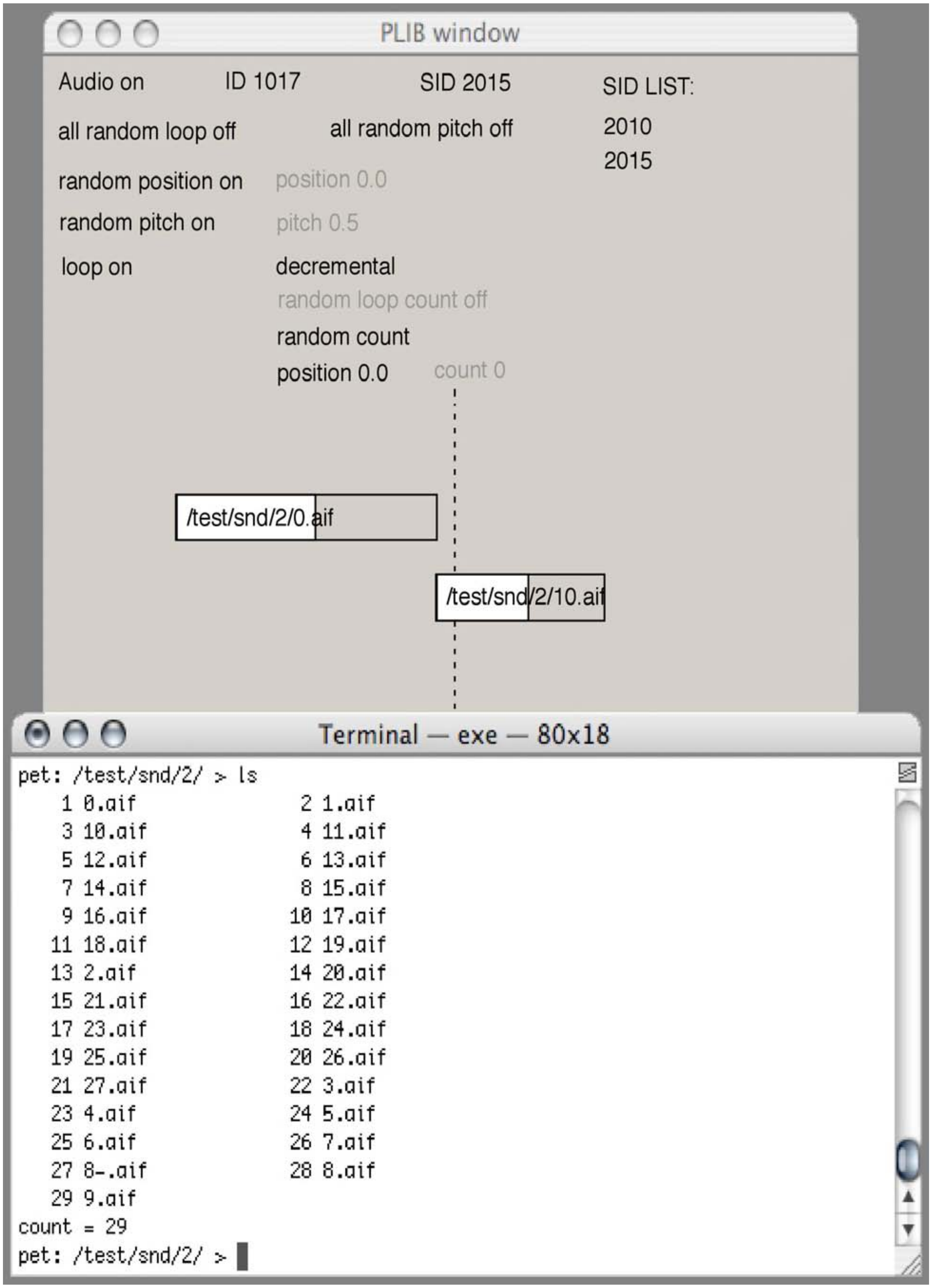
5 Acknowledgements

Götz Dipper, Frank Neumann, Anne Vortisch, Dan Santucci

6 Project Webpage

http://www.theangryyoungcomputers.de/fui

[2]http://ccrma.stanford.edu/software/stk/
[3]http://plib.sourceforge.net/
[4]http://music.mcgill.ca/
[5]http://www.portaudio.com
[6]http://www.fmod.org
[7]http://jackit.sourceforge.net/



```
pet: /test/snd/2/ > ls
    1 0.aif              2 1.aif
    3 10.aif             4 11.aif
    5 12.aif             6 13.aif
    7 14.aif             8 15.aif
    9 16.aif            10 17.aif
   11 18.aif            12 19.aif
   13 2.aif             14 20.aif
   15 21.aif            16 22.aif
   17 23.aif            18 24.aif
   19 25.aif            20 26.aif
   21 27.aif            22 3.aif
   23 4.aif             24 5.aif
   25 6.aif             26 7.aif
   27 8-.aif            28 8.aif
   29 9.aif
count = 29
pet: /test/snd/2/ >
```

Figure 8: "fui" Screenshot

The MusE Sequencer: Current Features and Plans for the Future

Werner SCHWEER
Ludgerweg 5
33442 Clarholz-Herzebrock, Germany
ws@seh.de

Frank NEUMANN
Bärenweg 26
76149 Karlsruhe, Germany
beachnase@web.de

Abstract

The MusE MIDI/Audio Sequencer[1] has been around in the Linux world for several years now, gaining more and more momentum. Having been a one-man project for a long time, it has slowly attracted several developers who have been given cvs write access and continuously help to improve and extend MusE.

This paper briefly explains the current feature set, gives some insight into the historical development of MusE, continues with some design decisions made during its evolution, and lists planned changes and extensions.

Keywords

MIDI, Audio, Sequencer, JACK, ALSA

1 Introduction

MusE is a MIDI/Audio sequencer which somewhat resembles the look 'n' feel and functionality of software like Cubase for Windows. It is based on the concepts of (unlimited) tracks and parts, understands both internal and external MIDI clients (through the ALSA[2] driver framework) and handles different types of tracks: MIDI, drum and audio tracks. For audio, it provides a built-in mixer with insert effects, subgroups and a master out section and allows to send the downmix to a soundcard, a new audio track or to a file.

Through support of the LADSPA[3] standard, a large amount of free and open effect plugins are available to be used on audio tracks.

By being based on the JACK audio framework, it is possible to both route other program's sound output into MusE and route MusE's output to other programs, for instance a mastering application like Jamin[4].

MusE's built-in editors for MIDI/audio data come close to the average PC application with expected operations like selection, cut/copy/paste, Drag&Drop, and more. Unlimited Undo/Redo help in avoiding dataloss through accidental keypresses by your cat.

2 Historical Rundown

MusE has been developed by german software developer Werner Schweer since roughly January 2000. Early developments started even years before that; first as a raw Xlib program, later using the Tcl/Tk scripting language because it provided the most usable API and nicest look at that time (mid-90s). It was able to load and display musical notes in a pianoroll-like display, and could play out notes to a MIDI device through a hand-crafted kernel module that allowed somewhat timing-stable playback by using the kernel's timers. MIDI data was then handled to the raw MIDI device of OSS. As the amount of data to be moved is rather small in the MIDI domain, reasonable timing could be reached back then even without such modern features as "realtime-lsm" or "realtime-preempt" patches that we have today in 2.6 kernels.

With a growing codebase, the code quickly became too hard to maintain, so the switch to another programming language was unavoidable. Since that rewrite, MusE is developed entirely in C++, employing the Qt[5] user interface toolkit by Trolltech, and several smaller libraries for housekeeping tasks (libsndfile[6], JACK[7]).

In its early form, MusE was a MIDI only sequencer, depending on the Open Sound System (OSS) by 4Front Technologies[8]. When the ALSA audio framework became stable and attractive (mid-2000), ALSA MIDI support was added, and later on also ALSA audio output. Summer 2001 saw the introduction of an important new feature, the "MESS" (MusE Experimental Soft Synth"). This allows for the development of pluggable software synthesizers, like the VSTi mechanism on Steinberg's Cubase sequencer software for Windows.

At some point in 2003 the score editor which was so far a part of MusE was thrown out (it was not really working very well anyway) and has then been reincarnated as a new SourceForge project, named MScore[9]. Once it stabilizes, it should be able to read and export files not only in standard MIDI file format, but also in MusE's own, XML-based .med format.

In October 2003 the project page moved to a new location at SourceForge. This made maintenance of some parts (web page, cvs access, bug reporting) easier and allowed the team of active developers to grow. Since this time, MusE is undergoing a more streamlined release process with a person responsible for producing releases (Robert Jonsson) and the Linux-typical separation into a stable branch (only bug fixes here) and a development branch (new features added here).

In November 2003 the audio support was reduced to JACK only. Obviously the ALSA driver code inside JACK was more mature than the one in MusE itself, and it was also easier to grasp and better documented than the ALSA API.

Additionally, fully supporting the JACK model meant instant interoperability with other JACK applications. Finally, it had also become too much of a hassle for the main developer to maintain both audio backends. The data delivery model which looked like a "push" model from outside MusE until now (though internally it had always used a separate audio thread that pulls the audio data out of the engine and pushes it into the ALSA pcm devices) has now become a clear "pull" model, with the jackd sound server collecting audio data from MusE and all other clients connected to it, and sending that data out to either pcm playback devices or back to JACK-enabled applications.

It has been asked whether MusE can be used as a simple "MIDI only" sequencer without any audio support. The current CVS source contains some "dummy JACK code" which gets activated when starting MusE in debug mode. If the interest in this is high enough, it might get extended into a real "MIDI only" mode.

In early 2004, the user interface underwent substantial changes when Joachim Schiele joined the team to redesign a lot of pixmaps for windows, menus, buttons and other user interface elements. Finally, MusE also received the obligatory splash screen!

Another interesting development of 2004 is that it brought a couple of songs composed, recorded and mixed with MusE. This seems to indicate that after years of development work, MusE is slowly becoming "ready for the masses".

3 Examples of Coding Decisions

(1) In earlier versions of MusE, each NoteOn event had its own absolute time stamp telling when this event should be triggered during playback. When a part containing a set of elements was moved to a different time location, the time stamps of all events in this part had to be offset accordingly. However, when the concept of "clone copies" was introduced (comparable to symlinks under Linux: Several identical copies of a part exist, and modifying an event in one part modifies its instance in all other cloned copies), this posed a problem: The same dataset is used, but of course the timestamps have to be different. This resulted in a change by giving all events only a timestamp relative to the start of the part it lives in. So, by adding up the local time stamp and the offset of the part's start from the song start, a correct event playback is guaranteed for all parts and their clone copies.

(2) MIDI controller events can roughly be separated into two groups: Those connected to note events, and those decoupled from notes. The first group is covered by note on/off velocity and to some degree channel aftertouch. The second group contains the classic controllers like pitchbender, modulation wheel and more. Now, when recording several tracks of MIDI data which are all going to be played back through the same MIDI port and MIDI channel, how should recording (and later playback) of such controller events be handled? Also, when moving a part around, should the controller data accompanying it be moved too, or will this have a negative impact on another (not moved) part on another track? Cubase seems to let the user decide by asking him this, but there might be more intelligent solutions to this issue.

(3) The current development or "head" branch of the MusE development allows parameter automation in some points. How is parameter

automation handled correctly, for both MIDI and audio? Take as an example gain automation. For MIDI, when interpolating between two volumes, you do normally not want to create thousands of MIDI events for a volume ramp because this risks flooding a MIDI cable with too much data and losing exact timing on other (neighbouring) MIDI events. For audio, a much finer-grained volume ramp is possible, but again if the rate at which automation is applied (the so-called "control rate") is driven to extremes (reading out the current gain value at each audio frame, at *audio rate)*, too much overhead is created. So instead the control rate is set to a somewhat lower frequency than the audio rate. One possible solution is to go for the JACK buffer size, but this poses another problem: Different setups use different values for sample rate (44.1kHz? 48kHz? 96kHz?) or period size, which means that the same song might sound slightly different on different systems. This is an ongoing design decision, and perhaps communication with other projects will bring some insight into the matter.

4 Weak Spots

There are a couple of deficiencies in MusE; for all of these efforts are already underway to get rid of them, though:

- There is a clear lack of documentation on the whole software. This is already tackled, however, by a collaborative effort to create a manual through a Wiki-based approach[10].
- The developer base is small compared to the code base, and there is a steep learning curve for prospective new developers wishing to get up to speed with MusE's internals. However, this seems to be true for a lot of medium-sized or large Open Source projects these days. Perhaps better code commenting (e.g. in the Doxygen[11] style) would help to increase readability and understandability of the code.
- Not all parts of MusE are as stable as one would want. One of the reasons is that the focus has been more on features and architecture than on stability for quite some time, though since the advent of stable and development versions of MusE this focus has changed a bit and MusE is getting more stable.

5 Future Plans

The plans for the future are manifold - as can be seen very often with open-source projects, there are gazillions of TODO items and plans, but too little time/resources to implement them all. What follows is a list of planned features, separated into "affects users" and "affects developers". Some of these items are already well underway, while others are still high up in the clouds.

5.1 Planned changes on the User level

- Synchronisation with external MIDI devices. This is top priority on the list currently, and while some code for MIDI Time Code (MTC) is already in place, it needs heavy testing. MusE can already send out MMC (MIDI Machine Control) and MIDI Clock, but when using MusE as a slave, there is still some work to be done. There have been plans for a while in the JACK team to make the JACK transport control sample-precise, and this will certainly help once it is in place.
- A file import function for the old (0.6.x) MusE .med files. This has been requested several times (so there *are* in fact people using MusE for a while! ☺), and as all .med files (XML-based) carry a file version string inside them, it is no problem to recognize 1.0 (MusE 0.6.x) and 2.0 (0.7.x) format music files.
- Complete automation of all controllers (this includes control parameters in e.g. LADSPA plugins).
- Mapping audio controllers to MIDI controllers: This would allow using external MIDI control devices (fader boxes, e.g. from Doepfer or Behringer) to operate internal parameters and functions (transport, mixer etc).
- A feature to align the tempo map to a freely recorded musical piece (which will alter the tempo map in the master track accordingly). This would bring a very natural and "human" way of composing and recording music while still allowing to later add in more tracks, e.g. drums.
- Support for DSSI soft synths (see below)
- A configurable time axis (above the track list) whose display can be switched between "measure/beat/tick", SMPTE (minute / second / frame) and "wallclock time".
- The "MScore" program which has been separated from MusE will start to become useful for real work. It will make use of the above mentioned new libraries, AWL and AL,

and will have a simple playback function built in (no realtime playback though), employing the fluidsynth software synthesizer. It will be able to work with both .mid and .med files, with the .med file providing the richer content of the two. MScore is still lacking support for some musical score feature like triplets and n-tuplets, but this is all underway. A lot of code is already in place but again requires serious testing and debugging.

5.2 Planned changes on the Developer level

- Better modularisation. Two new libraries are forming which will become external packages at some point: "AWL" (Audio Widget Library) which provides widgets typically found in the audio domain, like meters, location indicators, grids or knobs, and "AL" (audio library) which features house-holding functions like conversion between a tempo map (MIDI ticks) and "wallclock time" (SMPTE: hh:mm:ss:ff).
- Also, MESS soft synths shall get detached from the MusE core application so that they can get built easier and with less dependencies. This reduces the steepness of the learning curve for new soft synth developers.
- The new "DSSI"[12] (Disposable SoftSynth Interface) is another desired feature. Already now the "VST" and "MESS" classes have a common base class, and adding in support for DSSI here should not be too complicated.
- The creation of a "demosong regression test suite" will be helpful in finding weak spots of the MIDI processing engine. This could address issues like MIDI files with more than 16 channels, massive amounts of controller events in a very short time frame, SysEx handling, checks for "hung notes" and more. Getting input and suggestions from the community on this topic will be very helpful!

6 Conclusions

MusE is one of the best choices to compose music under Linux when the "classical" approach (notes, bars, pattern, MIDI, samples) is required.

Thanks to the integration with the existing ALSA and JACK frameworks, it can interact with other audio applications to form a complete recording solution, from the musical idea to the final master.

7 Other Applications

There are some other MIDI/audio applications with a similar scope as MusE; some of them are:

- Rosegarden[13], a KDE-based notation software and sequencer
- Ardour[14], turning a computer into a digital audio workstation
- seq24[15], a pattern-based sequencer (MIDI-only) and live performance tool
- Cheesetracker[16], a "classic" pattern-oriented tracker application
- Beast[17], an audio sequencer with a built-in graphical modular synthesis system

Besides these, there are already numerous soft synthesizers/drum machines/samplers etc that are able to read MIDI input through ALSA and send out their audio data through JACK; these programs can be controlled from within MusE and thus extend its sound capabilities. However, connecting them with MusE MIDI- and audiowise is more complicated, and can cause more overhead due to context switches.

8 Acknowledgements

The authors wish to acknowledge the work of everyone helping in turning a Linux-based computer into a viable and free alternative to typical commercial audio systems. No matter whether you are developing, testing, documenting or supporting somehow – thank you!

References

[1] http://www.muse-sequencer.org

[2] http://www.alsa-project.org

[3] http://www.ladspa.org

[4] http://jamin.sourceforge.net

[5] http://www.trolltech.com/products/qt/index.html

[6] http://www.mega-nerd.com/libsndfile/

[7] http://jackit.sourceforge.net

[8] http://www.4front-tech.com

[9] http://mscore.sourceforge.net

[10] http://www.muse-sequencer.org/wiki/index.php/Main_Page

[11] http://www.doxygen.org

[12] http://dssi.sourceforge.net

[13] http://www.rosegardenmusic.com

[14] http://www.ardour.org

[15] http://www.filter24.org/seq24/index.html

[16] http://www.reduz.com.ar/cheesetronic

[17] http://beast.gtk.org

ZynAddSubFX – an open source software synthesizer

Nasca Octavian PAUL
Tg. Mures, Romania
zynaddsubfx@yahoo.com

Abstract

ZynAddSubFX is an open source real-time software synthesizer that produces many types of sounds. This document will present the ZynAddSubFX synthesizer and some ideas that are useful in synthesizing beautiful instruments without giving too much (mathematical) detail.

Keywords

Synthesizer, bandwidth, harmonics.

1 Introduction

The ZynAddSubFX software synthesizer has polyphonic, multi-timbral and microtonal capabilities. It has powerful synth engines, many types of effects (Reverberation, Echo, Chorus, Phaser, EQ, Vocal Morpher, etc.) and contains a lot of innovations. The synthesizer engines were designed to make possible many types of sounds by using various parameters that allow the user to control every aspect of the sound. Special care was taken to reduce the amount of computation in order to produce the sound, but without lowering it's quality.

2 ZynAddSubFX structure

ZynAddSubFX has three synth engines and allows the user to make instrument kits. In order to make possible to play multiple at the instruments same time, the synth is divided into a number of parts. One part can contain one instrument or one instrument kit. The effect can be connected as System Effects, Insertion Effects and Part Effect.

The system effects are used by all parts, but the user can choose the amount of the effect for each part. The Insertion Effects are connected to one part or to the audio output. The Part Effects are a special kind of effect that belong to a single part, and they are saved along the instrument.

2.1 Synth engines

The engines of ZynAddSubFX are: ADDsynth, SUBsynth and PADsynth. In the Fig.1 it shows the structure of these engines:

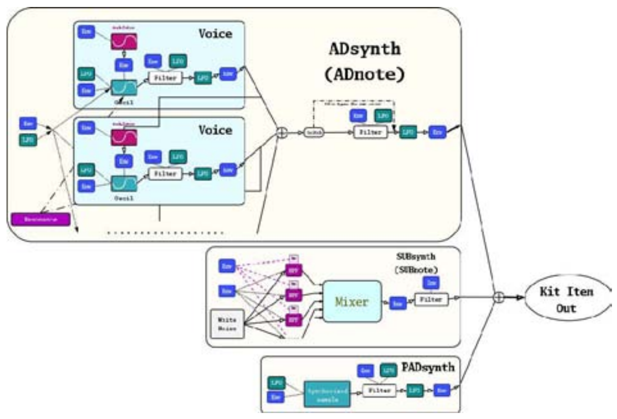


Fig. 1 Synth engines

1) ADDsynth
The sound is generated by the oscillator. The oscillator has different kind of parameters, like the harmonic type(sine, saw, square, etc.), the harmonic content, the modulations, waveshapers, filters. These parameters allow the oscillators to have any shape. A very interesting parameter of the oscillator is called "adaptive harmonics". This parameter makes possible very realistic sounds, because it allows to control how the resonances appear on different pitches. The oscillators includes a very good anti-aliasing filter that avoids aliasing even at the highest pitches. If the user wants, the oscillator can be modulated by another oscillator(called the "modulator") using the frequency modulation, phase modulation or the ring modulation. The frequency of the oscillators can be changed by the low frequency oscillators and envelopes. After the sound is produced by the oscillator, it passes through

filters and amplitude changers controlled by LFOs and envelopes.
An oscillator with a modulator and the amplitude/frequency/filter envelopes is called a “voice”. The ADDsynth contains more voices. The output of voices is added together and the result is passed through another amplitude/filter envelopes and LFO. A interesting feature is that the output of the voice can be used to modulate a oscillator from another voice, thus making possible to use modulation stacks. All the oscillators that are not modulators can pass through a resonance box.

2) SUBsynth
This module produces sound by generating a white noise, filtering each harmonic(with band-pass filters) from the noise and adding the resulting harmonics. The resulting sound passes through a amplitude envelope and a filter controlled by another envelope.

3) PADsynth
This synth engine is the most innovative feature in ZynAddSubFX. It was designed following to the idea that the harmonics of the sounds are not some simple frequencies, but are rather are spread over a certain band of frequencies. This will be discussed later..
Firstly there will be generated a long sample (or few samples) according to the settings in this engine (like the frequency spread of the harmonics, the bandwidth of each harmonic, the position of the harmonics, etc..). After this, the sample is played at a certain speed in order to achieve the desired pitch.
Even though this engine is more simpler than ADDsynth, the sounds generated by it are very good and make possible very easy generation of instruments like pads, choirs, and even metallic noises like bells, etc.

2.2 Instrument/Part structure

The structure of the Parts are drawn in Fig.2.

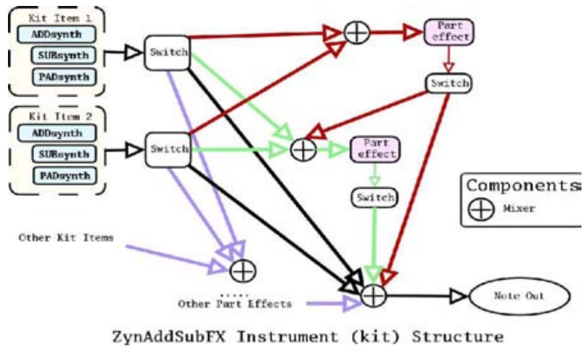


Fig. 2

The sum of the output of the ADDsynth, SUBsynth and PADsynth engines is called “kit item”, because ZynAddSubFX allows a part to contain several kit items. These kit's items can be used to make drum kits or even to obtain multi-timbrality for a single part (for dual instruments, like a bell+strings or rhodes+voice) The output of them can be processed by the part's effects. The instrument kit with the part effects is considered to be an instrument and saved/loaded into the instrument banks. An instrument, usually contains only one kit item.

2.3 ZynAddSubFX main structure

The main structure of ZynAddSubFX is drawn in the Fig.3.

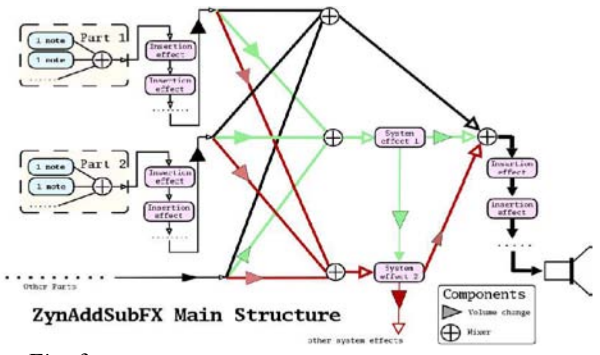


Fig. 3

As seen from the Fig.3, the part's output is sent to insertion effect and, after this, the signal can pass through system effects. A useful feature of the system effects is that the output of one system effect can go to the next system effect. Finally, the sound passes through the master insertion effects (they could be a EQ or a reverberation, or any other effect) and, after this, the sound is send to the audio output.

3 Design principles

This section presents some design principles and some ideas that were used to make the desired sounds with ZynAddSubFX.

3.1 The bandwidth of each harmonic

This considers that the harmonics of the pitched sounds are spread in frequency and are not a single sine function.

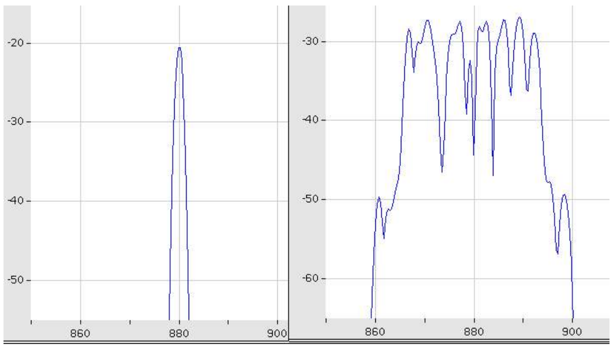


Fig. 4 A narrow band harmonic vs. a wide band harmonic

This helps to produce "warm" sounds, like choir, orchestra or any other ensembles. So the bandwidth of each harmonic can be used to measure the ensemble effect.

An important aspect about the bandwidth of each harmonic is the fact, that if you'll measure it in Hz, it increases for higher harmonics. For example, if a musical tone has the "A" pitch (440Hz) and the bandwidth of the first harmonic is 10 Hz, the bandwidth of the second harmonic will be 20 Hz, the bandwidth of the third harmonic will be 30 Hz, etc..

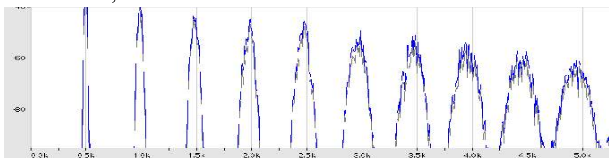


Fig.5 Higher harmonics has a higher bandwidth

Because of this, if the sound has enough harmonics, the upper harmonics merge to a continuous frequency band (Fig.6).

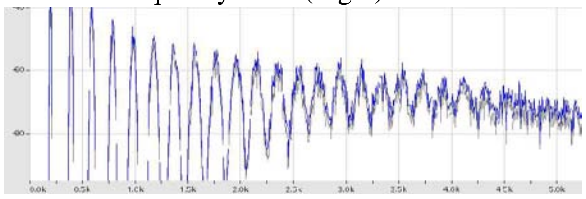


Fig. 6 Higher harmonics merges to a continuous frequency band

Each ZynAddSubFX module was designed to allow easy control of the bandwidth of harmonics easily:

- by detuning the oscillators from ADDsynth module and/or adding "vibrato".
- in SUBsynth, the bandwidth of each bandpass filter controls the bandwidth of the harmonics
- the PADsynth module uses this idea directly, because the user can control the frequency distribution of each harmonic.

3.2 Randomness

The main reason why the digital synthesis sounds too "cold" is because the same recorded sample is played over and over on each keypress. There is no difference between a note played first time and second time. Exceptions may be the filtering and some effects, but these are not enough. In natural or analogue instruments this does not happen because it is impossible to reproduce exactly the same conditions for each note. All three synth engines allow the user to use randomness for many parameters.

3.3 Amplitude decrease of higher harmonics on low velocity notes

All natural notes have this property, because on low velocity notes there is not enough energy to spread to higher harmonics. In ZynAddSubFX you can do this by using a lowpass filter that lowers the cutoff frequency on notes with low velocities or, if you use FM, by lowering the modulator index.

3.4 Resonance

If you change the harmonic content of the sound in order to produce higher amplitudes on certain frequencies and keep those frequencies constant, the listener will perceive this as if the instrument has a resonance box, which is a very pleasant effect to the ears. In ZynAddSubFX this is done by:

- using the Resonance function in ADDsynth and SUBsynth
- using the Adaptive Harmonics function from the Oscillators
- using filters, EQ or Dynamic filter effects

4 Basic blocks of ZynAddSubFX

4.1 Low Frequency Oscillators

These oscillators do not produce sounds by themselves, but rather change some parameters (like the frequency, the amplitude or the filters).

The LFOs have some basic parameters like the delay, frequency, start phase and depth. These parameters are shown in the Fig.7.

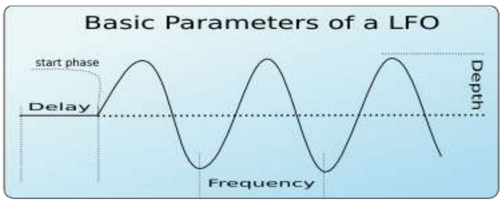


Fig. 7

Another important LFO parameter is the shape. There are many LFO types according to the shape. ZynAddSubFX supports the following LFO shapes (Fig. 8):

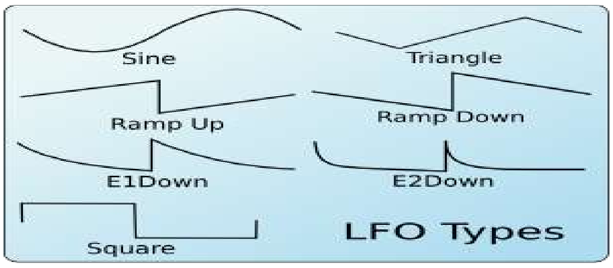


Fig. 8

ZynAddSubFX's LFOs have other parameters, like frequency/amplitude randomness, stretch, etc.

In the user interface the LFO interface is shown like this (fig.9):

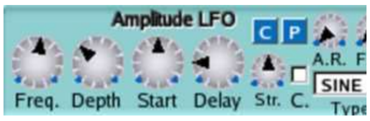


Fig. 9 LFO interface

4.2 Envelopes

Envelopes control how the amplitude, the frequency or the filter change over time. There are three types of envelopes: Amplitude Envelope, Frequency Envelope and Filter Envelope. All envelopes have 2 modes: parameter control (like ADSR – Attack-Decay-Sustain-Release, ASR – Attack-Sustain-Release) or Freemode, where the envelope can have any shape.

The ADSR envelopes control the amplitudes (fig. 10).

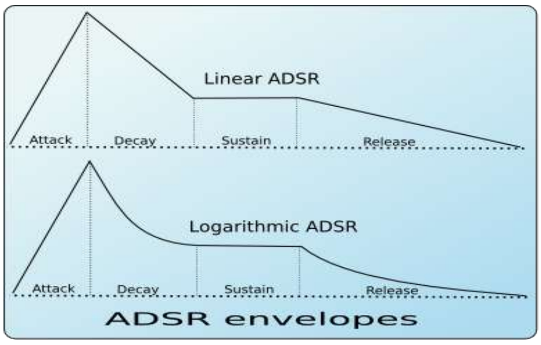


Fig. 10 Envelopes

The following images show the filter envelope as parameter control mode(Fig.11) or freemode (Fig.12).

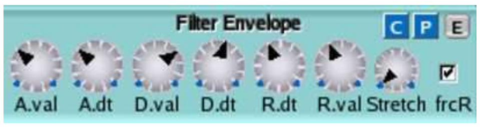


Fig. 11 Filter envelope user interface

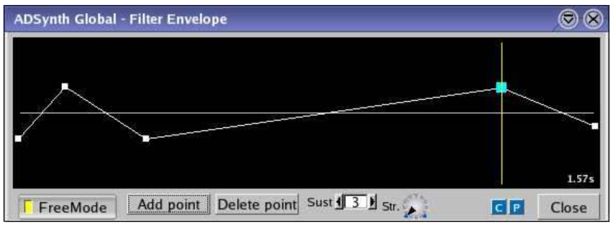


Fig. 12 Free-mode envelope user interface

4.3 Filters

ZynAddSubFX supports many types of filters. These filters are:

1. Analog filters:
   - Low/High Pass (1 pole)
   - Low/Band/High Pass and Notch (2 poles)
   - Low/High Shelf and Peak (2 poles)
2. Arbitrary format filters
3. State Variable Filters
   - Low/Band/High Pass
   - Notch

The analog filter's frequency responses are are shown in the Fig.13.

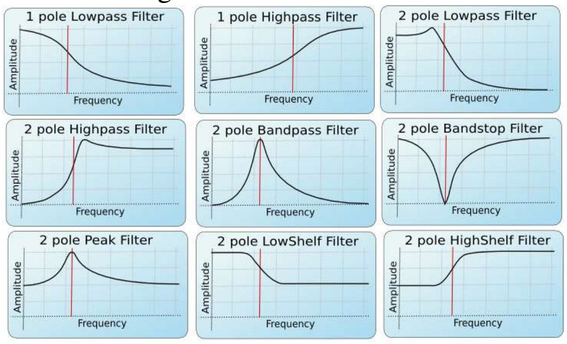


Fig. 13 Analog filter types and frequency response

The filters have several parameters that allow to get many types of responses. Some of these parameters are center/cutoff frequency, Q (this is the bandwidth of bandpass filters or the resonance of the low/high pass filters), gain (used by the peak/shelf filters).

Fig.14 shows how the Q parameter changes the filter response:

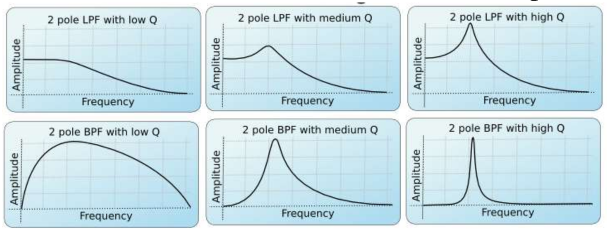


Fig. 14 "Q" parameter and filter frequency response

The Analog and State-Variable filters have a parameter that allows the user to apply the filtering multiple times in order to make a steeper frequency response, as is shown in the Fig.15.

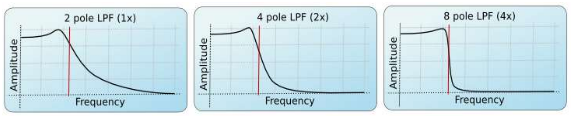


Fig. 15 Applying filter multiple times

The formant filters are a special kind of filter which can produce vowel-like sounds, by adding several formants together. A formant is a resonance zone around a frequency that can be produced by a bandpass filter. The user, also can specify several vowels that are morphed by smoothly changing the formants from one vowel to another.

Fig. 16 shows a formant filter that has an "A" and "E" vowel and how the morphing is done:

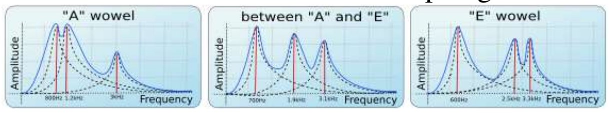


Fig. 16 Formant filter freq. response and morphing

5 ZynAddSubFX interaction to other programs

ZynAddSubFX receives MIDI commands from an OSS device or it can make a ALSA port that allows other programs (like Rosegarden or MusE sequencers) to interact with it. The audio output can be OSS or JACK.

6 Conclusion

ZynAddSubFX is an open source software synthesizer that produces sounds like commercial software and hardware synthesizers(or even better). Because it has a large number of parameters, the user has access to many types of musical instruments. Also, by using the idea of the bandwidth of each harmonic the sounds which are produced are very beautiful.

References

[1] http://zynaddsubfx.sourceforge.net

This document was written as accompanying material to a presentation at the 3rd International Linux Audio Conference 2005 in Karlsruhe, Germany.

Music Synthesis Under Linux

Tim Janik
University of Hamburg, Germany
timj@gtk.org

ABSTRACT

While there is lots of desktop software emerging for Linux which is being used productively by many end-users, this is not the case as far as music software is concerned. Most commercial and non-commercial music is produced either without software or by using proprietary software products. With BEAST, an attempt is made to improve the situation for music synthesis. Since most everything that is nowadays possible with hardware synthesizers can also be processed by stock PC hardware, it's merely a matter of a suitable implementation to enable professional music production based on free software. As a result, the development of BEAST focuses on multiple design goals. High quality demands are made on the mathematical characteristics of the synthesis, signals are processed on a 32-bit-basis throughout the program and execution of the synthesis core is fully real-time capable. Furthermore, the synthesis architecture allows scalability across multiple processors to process synthesis networks. Other major design goals are interoperability, so the synthesis core can be used by third-party applications, and language flexibility, so all core functionality can be controlled from script languages like scheme. In addition, the design of all components accounts for an intense focus on the graphical user interface to allow simple and if possible intuitive operation of the program.

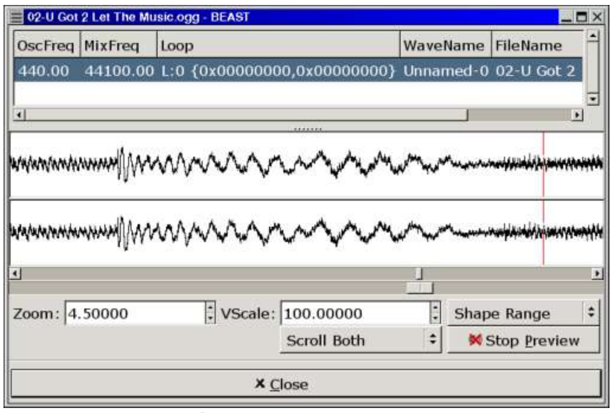
Keywords

Modular Synthesis, MIDI Sequencer, Asynchronous Parallel Processing, Pattern Editor.

1 BEAST/BSE - An Overview

BEAST is a graphical front-end to BSE which is a synthesis and sequencing engine in a separate shared library. Both are being released under the GPL and are being developed as free software for the best part of a decade. Since the first public release, some parts have been rolled out and reintegrated into other Projects, for instance the BSE Object system which became GObject in Glib. The programming interface of BSE is wrapped up by a glue layer, which allows for various language bindings. Currently a C binding exists which is used by BEAST. A C++ binding exists which is used to implement plugins for BSE and there also is a Scheme binding which is used for application scripting in BEAST or scripting BSE through the scheme shell bsesh.

BEAST allows for flexible sound synthesis and song composition based on utilization of synthesis instruments and audio samples. To store songs and synthesis settings, a special BSE specific hybrid text/binary file format is used which allows for seamless integration of audio samples, synthesis instruments and sequencing information.


02-U Got 2 Let The Music.ogg - BEAST

OscFreq	MixFreq	Loop	WaveName	FileName
440.00	44100.00	L:0 {0x00000000,0x00000000}	Unnamed-0	02-U Got 2

Zoom: 4.50000 VScale: 100.00000 Shape Range
Scroll Both

Wave View Dialog

Since the 0.5 development branch, BEAST offers a zoomable time domain display of audio samples with preview abilities. Several audio file formats are supported, in particular MP3, WAV, AIFF, Ogg/Vorbis and BseWave which is a hybrid text/binary file format used to store multi samples with loop and other accompanying information. A utility for creation, compression and editing of BseWave files is released with version 0.6.5 of BEAST. Portions of audio files are loaded into memory on demand and are decoded on the fly even for intense compression formats like Ogg/Vorbis or MP3. This allows for processing of very large audio files like 80 megabytes of MP3 data which roughly relates to 600 megabytes of decoded wave data or one hour of audio material. To save decoding processing power, especially for looped samples, decoded audio data is cached up to a couple of megabytes, employing a sensible caching algorithm that prefers trashing of easily decoded sample data (AIFF or WAV) over trashing processing intense data (Ogg/Vorbis).

The synthesis core runs asynchronously and performs audio calculations in 32-bit floating point arithmetic. The architecture is designed to support distribution of synthesis module calculations across multiple processors, in case multiple processors are available and the operating system supports process binding. In principle the sampling rate is freely adjustable, but it is in practice limited by operating system IO capabilities. The generated audio output can be recorded into a separate wave file.

The graphical user interface of BEAST sports concurrent editing of multiple audio projects, and unlimited undo/redo functionality for all editing functions. To easily give audio setups a try and for interactive alterations of synthesis setups, real-time MIDI events are processed. This allows utilization of BEAST as a ordinary MIDI synthesizer.

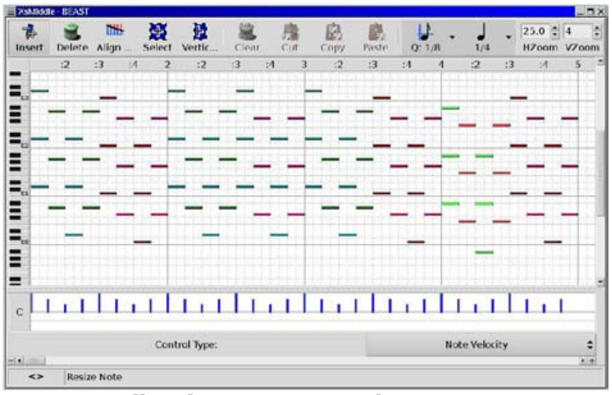
Since the complete programming interface of the synthesis core is available through a scheme shell, BEAST allows registration of scheme scripts at startup to extend its functionality and to automate complex editing tasks.

2 Song Composition

Songs consist of individual tracks with instruments assigned to them, and each track may contain multiple parts. A part defines the notes that are to be played for a specific time period.

The type of instrument assigned to a track is either a synthesis instrument, or an audio sample. Synthesis instruments are separate entities within the song's audio project and as such need to be constructed or loaded before use. In current versions, to enable sound compression or echo effects, post processing of audio data generated by a track or song is supported by assigning designated post processing synthesis meshes to them which simply act as ordinary audio filters, modifying the input signal before output.

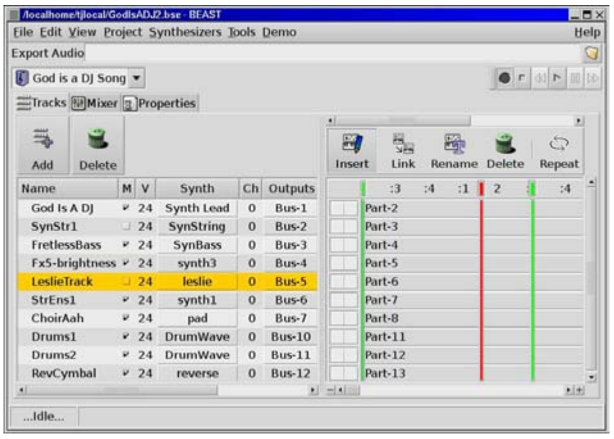
The post processing mechanism is currently being reworked, to integrate with the audio mixer framework that started shipping in recent versions of the 0.6 development branch. In the new audio mixer, audio busses can freely be created and connected, so volume metering or adjustment and effects processing is possible for arbitrary combinations of tracks and channels. Other standard features like muting or solo playback of busses are supported as well.



Piano Roll and MIDI Event Dialog

To allow editing of parts, a zoomable piano roll editor is supplied. Notes can be positioned by means of drag-and-drop in a two dimensional piano key versus time grid arrangement. This enables variation of note lengths and pitch through modification of note placement and graphical length. The piano keys also allow preview of specific notes by clicking on or dragging about. Also many other standard editing features are available via context menu or the toolbar, for instance note and event selection, cutting, pasting, insertion, quantization and script extensions. MIDI events other than notes, such as velocity or volume events can also be edited in an event editor region next to the piano roll editor. Newer versions of BEAST even sport an experimental pattern editor mode, which resembles well-known sound tracker interfaces. The exact integration of pattern mode editing with MIDI parts is still being worked out though.

Similar to notes within parts, the individual parts are arranged within tracks via drag-and-drop in the zoomable track view. Tracks also allow links to parts so a part can be reused multiple times within multiple tracks or a single track. The track view also offers editing abilities to select individual tracks to be processed by the sequencer, specification of the number of synthesis voices to be reserved and adding comments.


/localhome/tjlocal/GodIsADJ2.bse - BEAST
File Edit View Project Synthesizers Tools Demo Help
Export Audio
God is a DJ Song
Tracks Mixer Properties
Add
Delete
Insert Link Rename Delete Repeat
...Idle...

Name	M	V	Synth	Ch	Outputs	
God Is A DJ	✓	24	Synth Lead	0	Bus-1	Part-2
SynStr1		24	SynString	0	Bus-2	Part-3
FretlessBass	✓	24	SynBass	0	Bus-3	Part-4
Fx5-brightness	✓	24	synth3	0	Bus-4	Part-5
LeslieTrack		24	leslie	0	Bus-5	Part-6
StrEns1	✓	24	synth1	0	Bus-6	Part-7
ChoirAah	✓	24	pad	0	Bus-7	Part-8
Drums1	✓	24	DrumWave	0	Bus-10	Part-11
Drums2	✓	24	DrumWave	0	Bus-11	Part-12
RevCymbal	✓	24	reverse	0	Bus-12	Part-13

3 Synthesis Characteristics

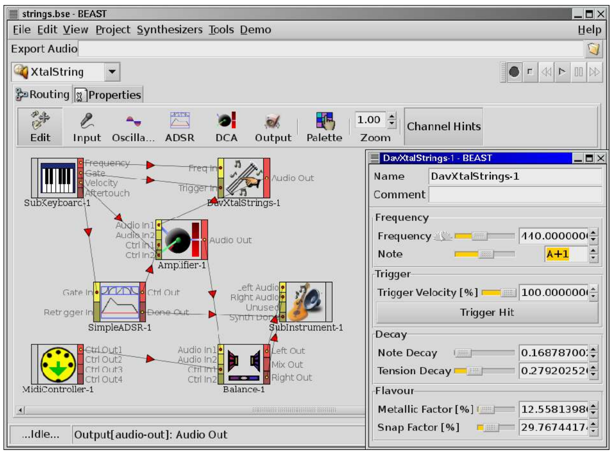
The synthesis facilities of the standard 0.6 development branch of the BEAST distribution, roughly equates the facilities of a simple modular synthesizer. However the quality and number of the supplied synthesis modules is constantly being improved.

Various synthesis modules are available. Amongst the audio sources are an Audio Oscillator, a Wave Oscillator, Noise, Organ and a Guitar Strings module. Routing functionality is implemented by modules like Mixer, Amplifier, ADSR-Envelope, Adder, Summation, Multiplier and Mini Sequencer. Various effect modules are also supplied, many based on recursive filters, i.e. Distortion, Reverb, Resonance, Chorus, Echos and the list goes on. Finally, a set of connection or IO modules is supplied for instrument input and output, MIDI input or synthesis mesh interconnection.

Apart from the synthesis modules shipped with the standard distribution, BSE also supports execution of LADSPA modules. Unfortunately, limitations in the LADSPA parameter system hinder seamless integration of LADSPA modules into the graphical user interface.

In general, the modules are implemented aliasing free, and highly speed optimized to allow real-time applicability. Per module, multiple properties (phase in an oscillator, resonance frequency of filters, etc...) are exported and can be edited through the user interface to alter synthesis functionality. A large part of mutable module parameters is exported through separate input or output channels, to allow for maximum flexibility in the construction of synthesis meshes.

BEAST generally does not differentiate between audio and control signals. Rather, the control or audio character of a signal is determined by the way of utilization through the user.



...Idle... Output[audio-out]: Audio Out

Name: DavXtalStrings-1
Comment:

Frequency
Frequency: 440.000000
Note: A+1

Trigger
Trigger Velocity [%]: 100.000000

Decay
Note Decay: 0.16878700
Tension Decay: 0.27920252

Flavour
Metallic Factor [%]: 12.5581398
Snap Factor [%]: 29.7674417

The graphical user interface provides for simple access to the construction and editing functionality of synthesis networks. Modules can be selected from a palette or context menu, and are freely placeable on a zoomable canvas. They are then connected at input and output ports via click-and-drag of connection lines. For each module, an information dialog is available and separate dialogs are available to edit module specific properties. Both dialogs are listed in the module context menu. Properties are grouped by functional similarities within editing dialogs, and many input fields support multiple editing metaphors, like fader bars and numeric text fields. All property and connection editing functions integrate with the project hosted undo/redo mechanism, so no editing mistakes committed can be finally destructive.

3.1 Voice-Allocation

The maximum number of voices for the playback of songs and for MIDI controlled synthesis can be specified through the graphical user interface. Increasing this number does not necessarily result in an increase in processor load, it just sets an upper limit within which polyphonic synthesis is carried out. To reduce processor load most effectively, the actual voice allocation is adjusted dynamically during playback time. This is made possible by running the synthesis core asynchronously to the rest of the application, and by preparing a processing plan which allows for concurrent processing of voice synthesis modules. This plan takes module dependencies into account which allows distribution of synthesis module processing tasks across multiple processors. Execution of individual processing branches of this plan can be controlled with sample granularity. This allows suspension of module branches from inactive voices. The fine grained control of processing activation which avoids block quantization of note onsets allows for precise realization of timing specifications provided by songs.

4 User experience and documentation

Like with most audio and synthesis applications, BEAST comes with a certain learning curve for the user to overcome. However, prior use of similar sequencing or synthesis applications may significantly contribute to reduction of this initial effort. The ever growing number of language translations can also be of significant help here, especially for novice users. BEAST does not currently come with a comprehensive manual, but it does provide a “Quick Start” guide which illustrates the elementary editing functions, and the user interface is equipped with tooltips and other informative elements explaining or exemplifying the respective functionality. Beyond that, development documentation for the programming interfaces, design documents, an FAQ, Unix manual pages and an online “Help Desk” for individual user problems are provided, accessible through the “Help” menu.

5 Future Plans

Although BEAST already provides solid functionality to compose songs and work with audio projects, there is still a long list of todo items for future development.

Like with any free software project with an open development process, we appreciate contributions and constructive criticism, so some of the todo highlights are outlined here:

- Extend the set of standard instruments provided.
- Implement more advanced effect and distortion modules.
- Adding a simple GUI editor for synthesis mesh skins.
- Implementing new sound drivers, e.g. interfacing with Jack.
- New instrument types are currently being worked on such as GUS Patches.
- Support for internal resampling is currently in planning stage.
- Extending language bindings and interoperability.

6 Acknowledgements

Our thanks go to the long list of people who have contributed to the BEAST project over the years.

7 Internet Addresses

BEAST home page:
http://beast.gtk.org
Contacts, mailing list links, IRC channel:
http://beast.gtk.org/contact.html
Open project discussion forums:
http://beast.gtk.org/wiki:BeastBse

8 Abbreviations and References

ADSR – *Attack-Decay-Sustain-Release*, envelope phases for volume shaping.
BEAST - *Bedevilled Audio System*, http://beast.gtk.org.
BSE - *Bedevilled Sound Engine*.
C++, C - Programming languages, http://www.research.att.com/~bs/C++.html.
FAQ – *Frequently Asked Questions*.
GLib - *Library of useful routines for C programming*, http://www.gtk.org.
GObject - GLib object system library.
GPL - *GNU General Public License*, http://www.gnu.org/licenses/gpl.html.
GUI – *Graphical User Interface*.
GUS Patch – *Gravis Ultrasound Patch* audio file format.
IRC – *Internet Relay Chat*.
Jack - *Jack Audio Connection Kit*, http://jackit.sourceforge.net.
LADSPA - *Linux Audio Developer's Simple Plugin API*, http://www.ladspa.org.
MIDI - *Musical Instruments Digital Interface*, http://www.midi.org/about-midi/specshome.shtml.
MP3, WAV, AIFF - sound file formats, http://beast.gtk.org/links-related.html.
Ogg/Vorbis - open audio codec, http://www.xiph.org/ogg/vorbis.

AGNULA Libre Music - Free Software for Free Music

Davide FUGAZZA and Andrea GLORIOSO
Media Innovation Unit - Firenze Tecnologia
Borgo degli Albizi 15
50122 Firenze
Italy
d.fugazza@miu.firenzetecnologia.it, a.glorioso@miu.firenzetecnologia.it

Abstract

AGNULA Libre Music is a part of the larger AGNULA project, whose goal as a european–funded (until April 2004) and as mixed private–volunteer driven (until today) project was to spread Free Software in the professional audio and sound domains; specifically, AGNULA Libre Music (ALM from now on) is a web–based datase of music pieces licensed under a "libre content" license. In this paper[1] Andrea Glorioso (former technical manager of the AGNULA project) and Davide Fugazza (developer and maintainer of AGNULA Libre Music) will show the technical infrastructure that powers ALM, its relationship with other, similar, initiatives, and the social, political and legal issues that have motivated the birth of ALM and are driving its current development.

Keywords

AGNULA, libre content, libre music, Creative Commons

1 The AGNULA project — a bit of history

In 1998 the situation of sound/music Free Software applications had already reached what could be considered well beyond initial pioneeristic stage. At that time, the biggest problem was that all these applications were dispersed over the Internet: there was no common operational framework and each and every application was a case-study by itself.

But when Marco Trevisani proposed (this time to Nicola Bernardini, Günter Geiger, Dave Phillips and Maurizio De Cecco) to build DeMuDi (*Debian Multimedia Distribution*) an unofficial Debian-based binary distribution of sound/music Free Software, something happened.

[1]This paper is Copyright © 2005 Fugazza, Glorioso and Copyright © 2005 Firenze Tecnologia. It is licensed under a Creative Commons BY-SA 2.0 License (see `http://creativecommons.org/licenses/by-sa/2.0/legalcode`).

Nicola Bernardini organized a workshop in Firenze, Italy at the beginning of June 2001, inviting an ever–growing group of supporters and contributors (including: Marco Trevisani, Günter Geiger, Dave Phillips, Paul Davis, François Déchelle, Georg Greve, Stanko Juzbasic, Giampiero Salvi, Maurizio Umberto Puxeddu and Gabriel Maldonado). That was the occasion to start the first concrete DeMuDi distribution, the venerable *0.0 alpha* which was then quickly assembled by Günter Geiger with help from Marco Trevisani. A bootable CD-version was then burned just in time for the ICMC 2001 held in La Habana, Cuba, where Günter Geiger and Nicola Bernardini held a tutorial workshop showing features, uses and advantages of DeMuDi(Déchelle et al., 2001).

On November 26, 2001 the European Commission awarded the AGNULA Consortium — composed by the Centro Tempo Reale, IRCAM, the IUA-MTG at the Universitat Pompeu Fabra, the Free Software Foundation Europe, KTH and Red Hat France — with consistent funding for an accompanying measure lasting 24 months (IST-2001-34879). This accompanying measure, which was terminated on March 31st 2004, gave considerable thrust to the AGNULA/DeMuDi project providing scientific applications previously unreleased in binary form and the possibility to pay professional personnel to work on the distribution.

After the funded period, Media Innovation Unit, a component of Firenze Tecnologia (itself a technological agency of the Chamber of Commerce of Firenze) has decided to partly fund further AGNULA/DeMuDi developments. Free Ekanayaka[2] is the current maintainer of the distribution.

AGNULA has constituted a major step in the direction of creating a full-blown Free Software infrastructure devoted to audio, sound and mu-

[2]`free@miu-ft.org`

sic, but there's much more to it: it is the first example of a European-funded project to clearly specify the complete adherence of its results to the Free Software paradigm in the project contract, thus becoming an important precedent for similar projects in the future (Bernardini et al., 2004).

2 AGNULA Libre Music: sociopolitics

On February 2003 Andrea Glorioso was appointed as the new technical manager of the AGNULA project, replacing Marco Trevisani who had previously served in that position but was unable to continue contributing to the project due to personal reasons.

This is not the place to explain in detail how the new technical manager of the AGNULA project tackled the several issues which had to be handled in the transition, mainly because of the novelty of the concept of "Free Software" for the European Commission (a novelty which sometimes resulted in difficulties to "speak a common language" on project management issues) and of the high profile of the project itself, both inside the Free Software audio community — for being the first project completely based on Free Software and **funded with european money** — and in the European Commission — for being the first project **completely based on Free Software** and funded with european money (Glorioso,).

The interesting point of the whole story — and the reason why it is cited here — is that the new Technical Manager, in agreement with the Project Coordinator (Nicola Bernardini, at the time research director of Centro Tempo Reale) decided to put more attention on the "social" value of project, making the life of the project more open to the reference community (i.e. the group of users and developers gravitating around the so called LA* mailing lists: `linux-audio-announce`,[3] linux-audio-users,[4] linux-audio-dev[5]) as well as creating an AGNULA community *per se*.

In September 2003, when the first idea of AGNULA Libre Music was proposed to the Project Coordinator by the Technical Manager for approval,[6] the *zeitgeist* was ripe with the "Commons".

A number of relevant academic authors from different disciplines had launched a counter–attack against what was to be known as the "new enclosure movement", (Boyle, 2003): the attempt of a restricted handful of multinational enterprises to lobby (quite succesfully) for new copyright extension and a stricter application of neighbouring rights.

The result of this strategy on behalf of the multinational enterprises of the music business was twofold: on the one hand, annoying tens of thousands of mostly law–abiding consumers with silly lawsuits that had no chance of standing in the court[7];[8] on the other hand, motivating even more authors to escape the vicious circle of senseless privatization that this system had taken to its extremes.

It seemed like a good moment to prove that AGNULA really wanted to provide a service to its community, and that it really had its roots (and its leaves, too) in the sort of "peer-to-peer mass production" (Benkler, 2002) that Free Software allowed and, some would argue, called for. After investing a major part of its human and financial resources on creating the project management infrastructure for working on the two GNU/Linux distributions the project aimed to produce, it was decided that a web–accessible database of music would be created, and the music it hosted would be shared and made completely open for the community at large.

Davide Fugazza was hired as the chief architect and lead developer of AGNULA Libre Music, which saw its light in February 2004.[9]

2.1 Libre Content vs Libre Software

What might be missing in the short history of ALM is that the decision to allow for the European Commission funding to be spent on this

[3]`http://www.music.columbia.edu/mailman/listinfo/linux-audio-announce`

[4]`http://www.music.columbia.edu/mailman/listinfo/linux-audio-announce`

[5]`http://www.music.columbia.edu/mailman/listinfo/linux-audio-announce`

[6]The reader should remember that AGNULA, being a publicly financed project, had significant constraints on what could or could be done during its funded lifetime — the final decision and responsibility towards the European Commission rested in the hands of the Project Coordinator.

[7]`http://www.groklaw.net/article.php?story=-20040205005057966`

[8]In fact, it can be argued that the real strategic reason of these lawsuits had a marketing/PR reason rather than substantial grounds, which does not make them less effective in the short term.

[9]See `http://lists.agnula.org/pipermail/announce/2004-February/000041.html`

sub–project of the main AGNULA project was not an easy one, for several reasons:

- The European Commission, as all large political bodies, is under daily pressure by several different lobbies;[10] the “all rights reserved” lobby, which is pressuring for an extension of copyright length and of the scope of neighbouring rights, was particularly aggressive at the time the ALM project was launched (and still is, by the way). This made financing a project, whose primary goal was to distribute content with flexible copyright policies, questionable in the eyes of the EC (to say the least);
- Software is not content in the eyes of the European Commission, which maintains a very strict separation between the two fields in its financing programmes.[11] Using money originally aimed at spreading Free **Software** in the professional audio/sound domain to distribute **content** was potentially risky, albeit the reasons for doing so had been carefully thought out;
- The licensing scheme which ALM applies, mainly based on the Creative Commons licenses,[12], did not and does not map cleanly on the licensing ontology of Free Software.

  Although there are striking similiarities in the goals, the strategies and the tactics of Creative Commons Corporation, Free Software Foundation and other organizations which promote Free Software, not all the Creative Commons licenses can be considered “Free” when analyzed under the lens of “Software” (Rubini, 2004). This point is discussed with more detail in section 4

3 AGNULA Libre Music: technique

To make a long story short, AGNULA Libre Music is a Content Management and online publishing system, optimized and specialized for audio files publication and management.

Registered users is given complete access to his/her own material. The system takes care of assuring data integrity and the validation of all information according to the given specifications.

Registration is free (as in free speech and in free beer) and anonymous — the only request is a valid e-mail address, to be used for automatic and service communications.

In the spirit of libre content promotion, no separation of functionalities between “simple users” and “authors” has been implemented: both classes of users can benefit from the same features:

- Uploading and publishing of audio files with automatic metatag handling;
- Real–time download statistics;
- Creation of personalized playlist, to be exported in the `.pls` and `.m3u` formats, themselves compatibles with the majority of players around (`xmms`,[13] `winamp` (TM),[14] `iTunes` (TM)[15]);

Other features which are available to anonymous users, too, are:

- A search engine with the possibility of choosing title, artist or album;
- RSS 2.0 feed with enclosures, to be used with “podcasting” supporting clients;[16];
- For developers and for integration with other services, ALM offers a SOAP (Group, 2003) interface that allows queries to be remotely executed on the database;

3.1 The web and tagging engine

ALM uses the `PostgreSQL` database[17] as the back–end and the `PHP` language[18] for its web–enabled frontend. PHP also handles a page templating and caching system, though the `Smarty` library.

File uploading on the server is handled through a form displayed on users’ browsers; first `HTTP` handles the upload on a temporary location on the server, and then a `PHP` script copies the audio files to their final destination.

It is in this phase that the `MP3` or `OGG Vorbis` *metags*, if already available in the file, are read.

[10]Please note that in this paper the term “lobby” is used with no moral judgement implied, meaning just a “pressure group” which tries to convince someone to apply or not apply a policy of a certain kind.

[11]It could be argues that, in the digital world, the difference between data (“content”) and computer programs is rather blurred.

[12]See `http://creativecommons.org/about/licenses/`.

[13]See `http://www.xmms.org/`.

[14]See `http://www.winamp.com/`.

[15]See `http://www.apple.com`.

[16]See `http://en.wikipedia.org/wiki/Podcasting`.

[17]See `http://www.postgresql.org/`.

[18]See `http://www.php.net`.

Besides, a form for the modification/creation of such tags is presented to the user.

The system ask which license should be applied to the files — without this indication files are not published and remain in an "invisible" state, except for the registered user who uploaded them in the first place.

To avoid abuses of the service and the uploading of material which has not been properly licensed to be distributed, all visitors (even anonymous ones) can signal, through a script which is present in every page, any potential copyright violation to the original author. The script also puts the file into an "invisible" status until the author either reviews or modifies the licensing terms.

3.2 Metadata and license handling

To guarantee a correct usage of the files and an effective way to verify licenses, the scheme proposed by the Creative Commons project has been adopted (Commons, 2004). Such scheme can be summarized as follows:

- using metagas inside files;
- using a web page to verify the license;

ALM uses the "TCOP" Copyright tag, which the ID3v2 metadata format provides (Nilsson, 2000), to show the publishing year and the URL where licensing terms can be found.

This page, which lives on the AGNULA Libre Music server, contains itself the URL of the Creative Commons licensing web page; moreover, it contains an RDF (Group, 2004) description of the work and of the usage terms.

In this way it is possible:

- to verify the authenticity of the license;
- to make it available a standardized description to search engines or specialized agents;

4 AGNULA Libre Music: legalities

4.1 Licensing policy

AGNULA Libre Music has decided to accept the following licenses to be applied on the audio files published and distributed through the system:

- Creative Commons Attribution-ShareAlike 2.0[19]
- Creative Commons Attribution 2.0[20]
- EFF Open Audio License[21]

The overall goal was to allow for the broadest possible distribution of music, leaving to the author the choice whether to apply or not a "copyleft" clause (Stallman, 2002a) — i.e. that all subsequent modifications of the original work should give recipients the same rights and duties that were given to the first recipient, thus creating a sort of "gift economy" (Stallman, 2002b), albeit of a very particular nature, possible only thanks to the immaterial nature of software (or digital audio files, in this case).

We chose not to allow for "non-commercial uses only" licenses, such as the various Creative Commons licenses with the `NC` (`Non Commercial`) clause applied. The reason for this choice are various, but basically boil down to the following list:

- Most of the AGNULA team comes from the Free Software arena; thus, the "non commercial" clause is seen as potentially making the work non-free. Further considerations on the difference between software and music, video or texts, and the different functional nature of the two sets would be in order here; but until now, an "old way" approach has been followed;
- It is extremely difficult to define what "non commercial" means; this is even more true when considering the different jurisdiction in which the works will be potentially distributed, and the different meanings that the term "commercial" assumes. Besides, what authors often really want to avoid is speculation on their work, i.e. a big company using their music, but have no objection against smaller, "more ethical" entities doing so.[22] However, "non commercial" licensing does not allow such fine–grained selection (Pawlo, 2004).

5 Future directions

AGNULA Libre Music is far from reaching its maximum potential. There are several key areas which the authors would like to explore;

[19]See `http://creativecommons.org/licenses-/by-sa/2.0/`.

[20]See `http://creativecommons.org/licenses/by-/2.0/`.

[21]See `http://www.eff.org/IP/Open_licenses-/20010421_eff_oal_1.0.html`.

[22]The decision of what constitutes an "ethical" business vs a non–ethical one is of course equivalent to opening a can of worms, and will not be discussed here.

moreover — and perhaps, much more interestingly for the reader — the AGNULA project has always been keen to accept help and contributions from interested parties, who share our commitment to Free Software[23] and circulation of knowledge.

More specifically, the ares which the ALM project is working on at the moment are:

- Integration with BitTorrent

  BitTorrent[24] has shown its ability to act as an incredibly efficient and effective way to share large archives (Cohen, 2003). AGNULA Libre Music is currently implementing a system to automatically and regularly create archives of its published audio files. The ALM server will act as the primary seeder for such archive.

- Integration with Open Media Streaming (OMS)

  Open Media Streaming[25] is

  > a free/libre project software for the development of a platform for the streaming of multimedia contents. The platform is based on the full support of the standard IETF for the real-time data transport over IP. The aim of the project is to provide an open solution, free and interoperable along with the proprietary streaming applications currently dominant on the market."

  ALM is currently analyzing the necessary step to interface its music archive with OMS, in order to have a platform completely based on Free Software and Open Standards to disseminate its contents. Besides, OMS is currently the only streaming server which "understands" Creative Commons licensing metadata, thus enabling even better interaction with ALM metatag engine (De Martin et al., 2004).

[23]It should be noted that Free Software Foundation Europe holds a trademark on the name "AGNULA"; the licensing terms for usage of such trademark clearly state that only works licensed under a license considered "free" by the Free Software Foundation can use the name "AGNULA".

[24]See `http://bittorrent.com/`.

[25]See `http://streaming.polito.it/`.

6 Acknowledgements

As the reader may expect, projects such as AGNULA and AGNULA Libre Music are the result of the common effort of a very large pool of motivated people. And indeed, giving credit to any deserving individual that contributed to these projects would probably fill completely the space allotted for this paper. Therefore, we decided to make an arbitrarily small selection of those without whose help AGNULA and AGNULA Libre Music would not probably exist. First of all, we would like to thank Richard Stallman, without whose effort Free Software would not exist at all; Lawrence Lessig, whose steadfast work on behalf of the Digital Commons has given justice to all the less known persons that worked on the subject in unriper times. Special thanks go to Roberto Bresin and to the Speech, Music and Hearing department of the Royal Institute of Sweden (KTH) for hosting the main AGNULA Libre Music server. Other people that deserve our gratitude are: Philippe Aigrain and Jean-François Junger, the European Commission officials that have been promoting the idea that AGNULA was a viable project against all odds inside the Commission itself; Dirk Van Rooy, later AGNULA Project Officer, Marc Leman and Xavier Perrot, patient AGNULA Project Reviewers; Luca Mantellassi and Giovanni Nebiolo, respectively President of Firenze's Chamber of Commerce and CEO of Firenze Tecnologia, for their support.

References

Y. Benkler. 2002. Coase's penguin, or, linux and the nature of the firm. *The Yale Law Journal*, 112.

N. Bernardini, D. Cirotteau, F. Ekanayaka, and A. Glorioso. 2004. The agnula/demudi distribution: Gnu/linux and free software for the pro audio and sound research domain. In *Sound and Music Computing 2004*, http://smc04.ircam.fr/.

J. Boyle. 2003. The second enclosure movement and the construction of the public domain. *Law and Contemporary Problems*, 66:33–74, Winter-Spring.

B. Cohen. 2003. Incentives build robustness in bittorrent. http://bittorrent.com/bittorrentecon.pdf, May.

Creative Commons. 2004. Using creative commons metadata. Technical report, Creative Commons Corporation.

J.C. De Martin, D. Quaglia, G. Mancini, F. Varano, M. Penno, and F. Ridolfo. 2004. Embedding ccpl in real-time streaming protocol. Technical report, Politecnico di Torino/IEIIT-CNR.

F. Déchelle, G. Geiger, and D. Phillips. 2001. Demudi: The Debian Multimedia Distribution. In *Proceedings of the 2001 International Computer Music Conference*, San Francisco USA. ICMA.

A. Glorioso. Project management, european funding, free software: the bermuda triangle? forthcoming in 2005.

XML Protocol Working Group. 2003. Soap version 1.2 part 0: Primer. Technical report, World Wide Web Consortium.

Semantic Web Working Group. 2004. Rdf primer. Technical report, World Wide Web Consortium.

M. Nilsson. 2000. Id3 tag version 2.4.0 - main structure. Technical report.

M. Pawlo, 2004. *International Commons at the Digital Age*, chapter What is the Meaning of Non Commercial? Romillat.

A. Rubini. 2004. Gpl e ccpl: confronto e considerazioni. CCIT 2004.

R Stallman, 2002a. *Free Software, Free Society: Selected Essays of Richard M. Stallman*, chapter What is Copyleft? GNU Books, October.

R. Stallman, 2002b. *Free Software, Free Society: Selected Essays of Richard M. Stallman*, chapter Copyleft: pragmatic idealism. GNU Books, October.

Where Are We Going And Why Aren't We There Yet ?

A Presentation Proposal for LAC 2005, Karlsruhe

Dave Phillips
linux-sound.org
400 Glessner Avenue
Findlay OH USA 45840
dlphillips@woh.rr.com

Abstract

A survey of Linux audio development since LAC 2004. Commentary on trends and unusual development tracks, seen from an experienced user's perspective. Magic predictions and forecasts based on the author's experience as the maintainer of the Linux Sound & Music Applications website, as a professional journalist specializing in Linux audio, and as a Linux-based practicing musician.

Keywords

history, survey, forecast, user experience, magic

1 Introduction

Linux sound and music software developers have created a unique world populated by some remarkable programs, tools, and utilities. ALSA has been integrated with the kernel sources, the Rosegarden audio/MIDI sequencer has reached its 1.0 milestone, and Ardour and JACK will soon attain their own 1.0 releases. Sophisticated audio and GUI toolkits provide the means to create more attractive and better-performing sound and music programs, and users are succeeding in actually using them.

2 A Brief Status Report

The Linux Sound & Music Applications site is the online world's most popular website devoted to Linux audio software. Maintaining the site is an interesting task, one in which we watch the philosophy of “Let 10,000 flowers blossom!” become a reality. It can be difficult to distinguish between a trend and the merely trendy, but after a decade of development there are definite strong currents of activity.

The past year has been a year of maturities for Linux audio software at both the system and application development levels. To the interested user, the adoption of the ALSA sound system into the Linux kernel means that Linux can start to provide sound services to whatever degree required. Desktop audio/video aficionados can enjoy better support for the capabilities of the their soundcards. Users seeking support for more professional needs can find drivers for some pro-audio hardware.

In addition to this advanced basic support there are patches for the Linux kernel that can dramatically reduce performance latency, bringing Linux into serious consideration as a viable professional-grade platform for digital audio production needs, at least at the hardware level. It is important to note that these patches are not merely technically interesting, that they are being used on production-grade systems now. Furthermore, there is a continuing effort to reduce or eliminate the need for patching at all, giving Linux superior audio capabilities out-of-the-box.

ALSA has passed its 1.0 release, as has Erik de Castro Lopo's necessary libsndfile. JACK is currently at 0.99, and the low-latency kernel patches have been well-tested in real-world application. The combined significance of these development tracks indicates that Linux is well on its way to becoming a viable contender in the sound and MIDI software arenas.

Support for the LADSPA plugin API has been an expected aspect of Linux audio applications for a few years. LADSPA limits are clear and self-imposed, but users want services more like those provided by VST/VSTi plugins on their host platforms. Support for running VST/VSTi plugins under Linux has also inspired users to ask for a more flexible audio/MIDI plugin API. At thistime the most likely candidate is the DSSI (Disposable SoftSynth Interface) from the Rosegarden developers. The DSSI has much to recommend it, including support for LADSPA and an interface for plugin instruments (a la VSTi plugins).

In this author's opinion the union of ALSA, JACK, and LADSPA should be regarded as the base system for serious audio under Linux. However, the world of Linux audio is not defined only by the AJL alliance. Other interesting and useful projects are going on with broader intentions that include Linux as a target platform.

The PortAudio/MIDI libraries have been adopted as the cross-platform solution to Csound5's audio/MIDI needs. Support for PortAudio has appeared in Hydrogen CVS sources, and it is already a nominal driver choice for JACK.

GRAME's MidiShare is not a newcomer to the Linux sound software world, but it is beginning to see some wider implementation. Among its virtues, MidiShare provides a flexible MIDI multiplexing system similar to the ALSA sequencer (it can even be an ALSA sequencer client). The system has been most recently adopted by Rick Taube's Common Music and the fluidsynth project.

Sound support in Java has been useful for a few years. All too often more attention has been paid to Java's licensing issues than to its audio capabilities. Many excellent Java-based applications run quite nicely on Linux, including the jMusic software, JSynthEdit, and Phil Burk's excellent jSyn plugin synthesizer.

At the level of the normal user the applications development track of Linux audio is simply amazing. Most of the major categories for music software have been filled or are being filled soon by mature applications. Ardour is designed for high-end digital audio production, Rosegarden covers the popular all-in-one Cubase-style mode, Audacity, Snd, and ReZound provide excellent editing software, Hydrogen takes care of the drum machine/rhythm programmer category, and MusE and Rosegarden cover the standard MIDI sequencer environment. Denemo and Rosegarden can be used as front-ends for LilyPond, providing a workpath for very high-quality music notation.

Notably missing from that list are samplers and universal editor/librarian software for hardware synthesizers. However, the LinuxSampler project is rapidly approaching general usability, and while the JSynthEdit project's pace is slow it does remain in development. Some similar projects have appeared in the past year, but none have advanced as far as JSynthEdit.

A host of smaller, more focused applications continues to thrive. Programs such as Jesse Chappell's FreqTweak and SooperLooper, Rui Capela's QJackCtl, and holborn's midirgui indicate that useful Linux audio software is becoming more easily written and that there is still a need for small focused applications. Of course the on-going development of graphics toolkits such as GTK, QT, and FLTK has had a profound effect on the usability of Linux applications.

Csound represents yet another significant class of sound software for Linux, that of the traditional language-based sound synthesis environment. The currently cutting-edge Csound is Csound5, basically a complete reorganization and rewrite (where necessary) of the Csound code base. Improvements include extensive modularization, internal support for Python scripting, and an enhanced cross-platform build system. The Linux version of Csound5 is already remarkable, with excellent realtime audio and MIDI performance capability.

One downside to the increasing capabilities of the Linux sound system is the increasing complexity of Linux itself. For most users it is decidedly uncomfortable and uninteresting to perform the necessary system modifications themselves, but happily the AGNULA/Demudi and Planet CCRMA systems have brought near-painless Linux audio system installation to the masses. However, given the resistance of said masses, we have seen the rise of the “live” Linux multimedia-optimized CD. These systems allow provide a safe and very effective means of introducing not only Linux audio capabilities but Linux in general, without alteration of the host system. The Fervent Software company has taken advantage of this trend and released their Studio To Go! commercially. I believe that these live CDs have enormous potential for Linux evangelization generally, and they may be a particular blessing for the expansion of interest in Linux audio capabilities.

3 Visibility Is Clear

Linux audio software is becoming a serious alternative for serious users. Composers of all sorts, pro-audio recordists, sound synthesis mavens, audio/video DJs and performance artists, all these and many other sound & music people are using this software on a productive daily basis. More “music made with Linux” has appeared in the past year than in the entire previous decade, and coverage of Linux audio regularly appears in major Linux journals. Articles on Linux audio software have appeared in serious audio journals such as Sound On Sound and the Computer Music Journal.

Some of the significant events acknowledging Linux audio software included Ron Parker's demonstrations of the viability of Ardour and JAMin in a commercial recording environment, Criscabello's announcement that he'd recorded Gilberto Gil with software libre, and the awards received for Hydrogen and JACK. Small steps perhaps, but they mark the steady progress of the development in this domain.

4 Some Problems

There is no perfection here. Lack of documentation continues to be a primary issue for many new users. Hardware manufacturers still refuse to massively embrace Linux audio development. Many features common in Win/Mac music software are still missing in their Linux counterparts. Many application types are still poorly represented or not represented at all.

Community efforts towards addressing documentation issues include various wikis (Ardour, Pd) and a few focus groups (Hydrogen, Csound5), and while many applications do have excellent docs, the lack of system-comprehensive documentation still plagues the new user, particularly when troubleshooting or attempting to optimize some aspect of an increasingly complex system. The problem is familiar and remains with us: writing good documentation is difficult and there are too few good writers with the time and energy to spare for the work required. Nevertheless, the impact of the documentation wikis is yet to be felt, they may yet prove to be a salvation for the befuddled user.

Hardware support still remains problematic. ALSA support expanded to the Echo cards, and the AudioScience company announced native Linux driver support for their high-end audoi boards, but no train of manufacturers hopped on the Linux sound support bandwagon. I'm not sure what needs to happen to convince soundcard and audio hardware manufacturers that they need to support Linux, and I believe that this issue needs some more focused discussion in the community.

Limited hardware support is often worse than none at all. ALSA developers are working to provide sound services as complete as their Win/Mac counterparts, but there are still problems with regard to surround sound systems (3D, 5.1) and access to on-board DSP chipsets.

A glance through any popular music publication clearly shows that Linux audio software, wonderful as it is, definitely lacks the variety of the Win/Mac worlds. Users new to the Linux audio world often lament the absence of programs such as Acid, Fruity Loops, or Ableton Live, and I have already mentioned the dearth of editor/librarian software for hardware MIDI synthesizers. The situation is surely improving, but there are still several application types awaiting project involvement.

5 Summary Conclusions

The good news far outweighs the bad, and the bad news itself can be dealt with in productive ways. The development and user communities continue to thrive, long-term projects proceed, more people are coming into our world, and more music is being made. Coordinated efforts need to be made to bring about greater program documentation and manufacturer participation, but whatever difficulties we encounter, the history of Linux software development advises us to never say never.

6 Acknowledgements

The author thanks the entire community of Linux audio developers for their enormous contribution to music and sound artists the world over. The author also thanks the community of Linux audio software users for their experiences, advice, and suggestions regarding a thousand sound- and music-related topics. Finally, the author extends great gratitude to the faculty and staff at ZKM for their continued support for this conference.