LAC2006 Proceedings

4th International Linux Audio Conference

April 27 – 30, 2006

ZKM | Zentrum für Kunst und Medientechnologie

Karlsruhe, Germany

Published by
ZKM | Zentrum für Kunst und Medientechnologie
Karlsruhe, Germany
April, 2006

www.zkm.de/lac/2006

Contents

Thursday, April 27, 2006 – Lecture Hall

12:00 AM *Fons Adriaensen*
Acoustical Impulse Response Measurement with ALIKI 9

02:00 PM *Arthur Clay, Thomas Frey and Jürg Gutknecht*
Unbounded: Aos & The GoingPublik Software 15

03:00 PM *Lee Revell*
Realtime Audio vs. Linux 2.6 21

04:00 PM *Asbjørn Sæbø and Peter Svensson*
A Low-Latency Full-Duplex Audio over IP Streamer 25

05:00 PM *Marije Baalman*
swonder3Dq: software for auralisation of 3D objects with Wave Field Synthesis 33

Friday, April 28, 2006 – Lecture Hall

11:00 AM *Yann Orlarey, Albert Gräf and Stefan Kersten*
DSP Programming with Faust, Q and SuperCollider 39

Saturday, April 29, 2006 – Lecture Hall

11:00 AM *Fons Adriaensen*
Design of a Convolution Engine optimised for Reverb 49

12:00 AM *Josh Green*
Sampled Waveforms And Musical Instruments 55

02:00 PM *Frank Barknecht*
128 Is Not Enough - Data Structures in Pure Data 61

03:00 PM *Eric Lyon*
A Sample Accurate Triggering System for Pd and Max/MSP 67

04:00 PM *Victor Lazzarini*
Scripting Csound 5 73

05:00 PM *Hartmut Noack*
Linux and music out of the box 79

Saturday, April 29, 2006 – Media Theatre

03:00 PM *Daniel James and Free Ekanayaka*
64 Studio - creative and native 85

Sunday, April 30, 2006 – Lecture Hall

12:00 AM *Martin Rumori*
footils. Using the foo Sound Synthesis System as an Audio Scripting Language 91

14:00 AM *Jürgen Reuter*
Ontological Processing of Sound Resources 97

Preface

Being the fourth of its kind, the International Linux Audio Conference 2006 is once again taking place at the ZKM | Institute for Music and Acoustics in Karlsruhe, Germany. By now the conference has become an established event and is worth being marked in calendars early.

We are very happy about the ongoing interest and devotion of our participants and guests. This allows us to offer a wide range of different programme entries—presentations, lectures, demos, workshops, concerts and more.

As with the last conference all submitted papers have again undergone a review process. At least two independent experts have read and commented on each paper, and their feedback was used by the submitters to further improve on the clarity and correctness of their work. This has once again resulted in what we think is a fine selection of currently ongoing developments in the Linux/Audio software scene.

As each year, we want to thank everyone who has participated in bringing this LAC2006 conference to life—authors, composers, reviewers, helpers and anyone we may have forgotten—and we wish everyone a pleasant and enjoyable stay at the ZKM and in Karlsruhe.

Götz Dipper and Frank Neumann
Organization Team LAC2006
Karlsruhe, April 2006

The International Linux Audio Conference 2006 is supported by

Organization Team LAC2006

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

ZKM

Peter Weibel	CEO
Jürgen Betker	Graphic Artist
Hartmut Bruckner	Sound Engineer
Ludger Brümmer	Head of the Institute for Music and Acoustics
Tobias Ehni	Event Management
Uwe Faber	Head of the IT Department
Hans Gass	Technical Assistant
Joachim Goßmann	Tonmeister
Gabi Gregolec	Event Management
Achim Heidenreich	Project Development
Martin Herold	Technical Assistant
Martin Knötzele	Technical Assistant
Andreas Liefländer	Coordination, Technical Assistant
Philipp Mattner	Technical Assistant
Alexandra Mössner	Assistant of Management
Caro Mössner	Event Management
Chandrasekhar Ramakrishnan	Software Developer
Thomas Saur	Sound Engineer
Joachim Schütze	IT Department
Anatol Serexhe	Technical Assistant
Berthold Schwarz	Technical Assistant
Bernhard Sturm	Production Engineer
Manuel Weber	Technical Director of the Event Department
Monika Weimer	Event Management
Susanne Wurmnest	Event Management

LAD

Jörn Nettingsmeier	Essen, Germany
Eric Dantan Rzewnicki	Radio Free Asia, Washington

Reviewers

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
Götz Dipper	ZKM Karlsruhe/Germany
Takashi Iwai	SUSE Linux Products GmbH, Nürnberg/Germany
Daniel James	64 Studio Ltd./UK
Victor Lazzarini	National University of Ireland, Maynooth
Fernando Lopez-Lezcano	CCRMA/Stanford University, California/USA
Jörn Nettingsmeier	Essen/Germany
Frank Neumann	Karlsruhe/Germany
Dave Phillips	Findlay, Ohio/USA

Acoustical Impulse Response Measurement with ALIKI

Fons ADRIAENSEN
fons.adriaensen@skynet.be

Abstract

The Impulse Response of an acoustical space can be used for emulation of that space using a convolution reverb, for room correction, or to obtain a number of measures representative of the room's acoustical qualities. Provided the user has access to the required transducers, an IR measurement can be performed using a standard PC equipped with a good quality audio interface. This paper introduces a Linux application designed for this task. The theoretical background of the method used is discussed, along with a short introduction to the estimated measures. A short presentation of the program's features is also included.

Keywords

Acoustics, impulse response, convolution, reverb.

1 Introduction

Equipment for measuring the acoustical parameters of an environment has traditionally been the realm of a small group of highly specialised electronics manufacturers. During the last decade, the ready availability of mobile computers and of high quality portable audio interfaces has resulted in a move towards mainly software based solutions.

Early programs just emulated the standardised (hardware based) procedures to measure e.g. reverb time. The computing power being available today enables the use of other methods, such as a direct measurement of a room's impulse response, from which all interesting information can be derived. Several methods to capture impulse responses have been developed, and these will be discussed below.

It seems that very little free and Linux-based software is available for this task. The Digital Room Correction package from Denis Sbragion [1] includes some scripts to perform an impulse response measurement. It is possible to obtain very good results with these scripts, but they are not easy to use.

In the Windows based world, a number of solutions have been available for some time. Most of these are based on the use of pseudo-random sequences. The MLSSA system from DRA Laboratories [2] (requiring special hardware) was one of the first using this method, and is well known. As an example of a package using more advanced methods, the Aurora Plugins [3] from Angelo Farina should be mentioned.

This paper introduces a new Linux based integrated system [4] developed by the author, and available under the terms of the GPL. ALIKI will capture impulse responses in up to eight channels simultaneously. The recorded data can be used directly for e.g. digital room correction, edited and prepared for use in a convolution based reverb, or used to compute acoustical parameters such as reverb time and various energy ratios.

The following sections will describe the measurement and analysis methods used in this software.

2 IR measurement methods

The impulse response (IR) of a system is the output signal it produces for an input consisting of a single Dirac pulse. The mathematical definition of a Dirac pulse requires zero width and unit energy, which is not possible in the real world, so in practice finite-width impulses compatible with the required bandwidth are used. In a sampled system in particular, the Dirac impulse is a signal consisting of one sample of unit amplitude followed by all zeros. It contains all frequencies from zero to the Nyquist limit with equal energy and a known phase.

Provided the system is linear and time-invariant, the IR contains all information there is about its behaviour, and permits the calculation of the system's response to any input signal.

[1] http://drc-fir.sourceforge.net

[2] http://www.mlssa.com

[3] http://farina.eng.unipr.it/aurora/home.htm

[4] http://users.skynet.be/solaris/linuxaudio/aliki.html

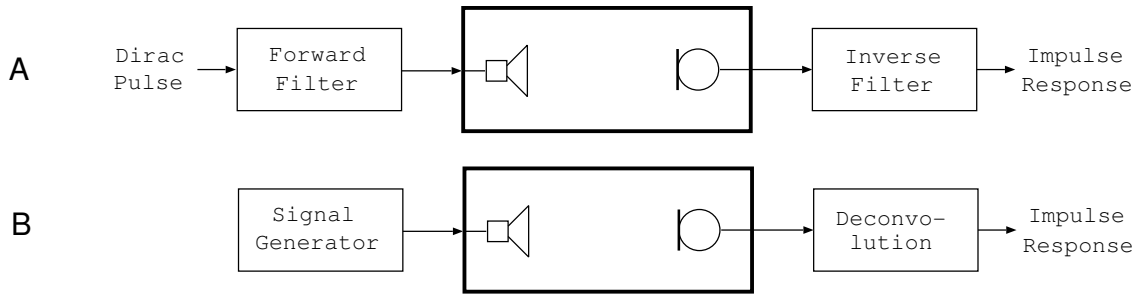

Figure 1: IR measurement using filtered Dirac pulse. A: theoretical model, B: practical realization.

The main problem with using Dirac pulses in an acoustical measurement is that as a result of their very short duration and finite amplitude, they contain very little energy, and measurement accuracy will be limited by the signal to noise ratio of the equipment used and of the system itself. While it is possible to use Dirac pulses reproduced by a loudspeaker in the controlled environment of an acoustics laboratory, this is all but infeasible in most real life situations, e.g. for measuring a room or concert hall, where there will always be background noises of some sort.

There are basically two ways to overcome this difficulty: either generate a high energy impulse directly as a sound, or find some method to spread the test signal over a longer time and to undo this operation after the measurement.

For the first approach, various methods have been used by acoustics engineers, ranging from exploding balloons and starter's pistols to very expensive special equipment to generate short high amplitude sound pulses. While such methods can be used e.g. to measure the reverb time of a concert hall, they still require a very large dynamic range in the measurement system, and they are not accurate and repeatable enough to obtain an IR to be used for room correction or for a convolution reverb.

The second solution is based on the following idea. Suppose we have a filter $\mathbf{H}$ with complex frequency response $H(\omega)$. If the filter has a non-zero gain at all frequencies, we can find an inverse filter $\mathbf{R}$ with frequency response $R(\omega) = z^{-n}/H(\omega)$. The z^{-n} is a pure delay required to make such a filter causal and physically realizable. Putting the two filters in series, only the (known) delay remains. Since the filters are linear, and if we assume the same of the system to be measured, we can put the system in between the two filters and obtain its impulse response using the filtered signal instead of the Dirac pulse (fig.1A).

Since we can regard any signal as the output of an FIR filter having the signal's sample values as its coefficients, we could in theory use any signal we want as long as the inverse filter exists and we can find some way to compute and implement it. For some classes of signals this can be done relatively easily, and that is the basis of the two methods discussed in the next sections. In practice the theoretical model of fig.1A is realized by generating the signal directly instead of filtering a Dirac pulse, and the inverse filtering is usually done by (de)convolution rather than a by real filter (fig.1B).

2.1 Maximum length binary sequences

Pseudo random binary sequences can be generated by a shift register with exclusive-or feedback from selected taps. Provided the correct feedback terms are used, a generator using N stages will produce a maximum length sequence (MLS) of length $L = 2^N - 1$. A sampled audio signal derived from such a sequence has exactly the same power spectrum as a Dirac pulse repeated every L samples, but it has L times more power for the same amplitude. For example, using an $L = 1023$ sequence will improve the signal to noise ratio by 30 dB.

The inverse filtering for such a signal can be done efficiently by using the Hadamard transform. Like for the Fourier transform, a 'fast' version of this transform exists (and it is even simpler than the FFT). This is the way the MLSSA software mentioned before (and many other systems) operate.

For acoustical measurements, the signal can be filtered further to obtain a 'pink' spectrum instead of 'white' noise, again improving the S/N ratio at low frequencies where it is usually the most problematic.

A more elaborate discussion of MLS based

techniques can be found in the references (Vanderkooy, 1994).

The main difficulty with the MLS method is its sensitivity to non-linear behaviour. Most loudspeakers produce substantial amounts of distortion, and this will interfere with the measurement and show up as spurious signals in the impulse response. The method discussed in the next section, while more complex to implement, does not have this problem.

2.2 Swept sine techniques

A second class of signals for which the inverse filter can be computed easily are linear and logarithmic frequency sweeps. For a linear sweep, the inverse is just the time reversal of the original signal. Such a signal has a 'white' spectrum, and for acoustical measurements a logarithmic sweep, having a 'pink' power spectrum is often preferred. In that case, provided the sweep is not too fast, the inverse filter is again the time-reversed original, but modified by a +6 dB per octave gain factor (6 dB, and not 3, since a +3 dB per octave correction applied to each filter separately would make both of them, and their product, 'white'). In both cases the inverse filter can be realized efficiently by using FFT-based convolution.

The advantage of using a sweep is that at any time we produce only a single frequency, and any distortion introduced will consist of the harmonics of that frequency only. If we use a *rising* frequency sweep, the harmonics will be generated ahead of the same frequencies appearing in the signal. So after deconvolution, any distortion will appear as spurious peaks in *negative* time in the impulse response, and most of it can then be edited out easily.

Another interesting feature of this method is that it does not depend on exact synchronisation of the playback and capture sample clocks. Any frequency error between these will result in a 'smearing' of the impulse response, in the sense that a Dirac pulse becomes itself a very short sweep. It is even possible to correct for this after the deconvolution.

The sweep method was pioneered by Angelo Farina (Farina, 2000), and it is the one used in ALIKI.

3 Measurements derived from the impulse response

This section provides a quick overview of some acoustical measures that can be calculated from a captured impulse response.

If IR measurements are performed for use in a convolution reverb system, then the choice of the transducers used is largely a matter of common sense combined with aesthetic preferences. The same is to some extent true if the object is room correction.

In contrast, in order to derive the measures described below, the IR measurement must be done according to a standardised procedure, and by using the correct equipment. In practice this means the use of true omnidirectional speakers (purpose built), and in some cases of a microphone calibrated for diffuse-field measurements (i.e. having a flat response integrated over all directions rather than on-axis).

The two ISO documents mentioned in the References section provide a good introduction to what is involved in such measurements.

All these values can be calculated for the full frequency range signal, for an A-weighted version, or octave or sub-octave bands.

3.1 The Schroeder integral

Most of the values described in the following sections can be obtained by computing the *Schroeder integral* of the IR, defined as follows. Let $p(t)$ be the impulse response, with $t = 0$ corresponding to the arrival of the direct sound. Then the Schroeder integral of $p(t)$ is the function

$$S(t) = \int_t^\infty p^2(t)dt \qquad (1)$$

In other words, $S(t)$ corresponds to the energy still remaining in the IR at time t. When plotted in dB relative to the maximum value at $t = 0$, $S(t)$ will be same as the level decay curve obtained after switching off a steady signal.

3.2 Reverb Time and Early Decay Time

The conventional definition of the *Reverb Time* is the time required for the sound level to decay to -60 dB relative to the original level, after a steady signal (normally white or filtered noise) is switched off. This time is normally denoted RT_{60} if the S/N ratio permits a reliable measurement down to that level, or RT_{30} if it is extrapolated from the -30 dB time.

For a measurement derived from an IR, it can be read directly from the Schroeder integral. The ISO standard prescribes that RT_{30} should be derived from the times the integral reaches respectively -5 dB and -35 dB, obtained by least-squares fitting, and extrapolated to the

60 dB range. The RT_{20} value is computed form the -5 dB and -25 dB times in the same way.

The *Early Decay Time EDT* is similar but derived from the -10 dB point of the integral, again by least-squares fitting.

3.3 Clarity, Definition and Central Time

The *Clarity* measure describes the ratio of the energies (in dB) before and after a given time referred to the arrival of the direct sound. This value provides a good indication of how 'clear' or 'transparent' the sound heard by a listener at the measurement position is. For speech, it is measured at 50 ms, while for music 80 ms is used. The two values are denoted C_{50} and C_{80}. The definition of C_{50} is

$$C_{50} = 10log_{10}\frac{\int_0^{0.050} p^2(t)dt}{\int_{0.050}^{\infty} p^2(t)dt} \quad (2)$$

$$= 10log_{10}\frac{S(0) - S(0.050)}{S(0.050)} \quad (3)$$

and similar for the 80 ms value.

The *Definition* is similar to Clarity, but is the simple ratio (not in dB) of the early sound energy to the *total* energy. In practice it's not necessary to compute both C and D, as they can easily be derived from each other.

The *Central Time* is the 'centre of gravity' of the energy in the IR, defined as

$$T_S = \frac{\int_0^{\infty} tp^2(t)dt}{\int_0^{\infty} p^2(t)dt} \quad (4)$$

3.4 Early Lateral Energy

To compute the values introduced in this section, two simultaneously captured IRs are required, one using an omnidirectional free-field microphone, and the second using a figure-of-eight (velocity) microphone pointing sideways (i.e. with the null pointing at the 'centre of stage'). These values provide some measure of the 'width' and 'spaciousness' of the sound. Extreme care and high quality equipment is required in order to obtain meaningful results from these computations.

Let $p_L(t)$ be the lateral IR, then

$$LF = \frac{\int_{0.005}^{0.080} p_L^2(t)dt}{\int_0^{0.080} p^2(t)dt} \quad (5)$$

and

$$LFC = \frac{\int_{0.005}^{0.080} |p_L(t)p(t)|dt}{\int_0^{0.080} p^2(t)dt} \quad (6)$$

LFC is said to correspond closer to subjective observation.

4 ALIKI program structure

ALIKI is written as an integrated package controlled by a graphical user interface[5]. Technically speaking it consists of two separate executables (the audio interface part is an independent process), but this is hidden from the user who just sees a single interface. Figure 2 shows the main modules and files used.

ALIKI can interface via JACK, or use ALSA devices directly, or it can be run without any audio hardware for off-line processing of stored data. It uses its own sound file format, but facilities to import or export WAV-format or raw sample files are included. The special format keeps all data for multichannel impulses conveniently together, facilitating handling (in particular when you have many files, all with similar and confusing names). It also allows to include specific metadata, for example parameters to be used by a convolution reverb. The metadata will become even more important when the program is extended to allow automated multiple measurements, e.g. to obtain polar diagrams.

4.1 The capture module

Functions of this module include

- input measurement parameters (frequency range, sweep time, channel names, etc.),
- generate and store the sweep and inverse filter waveforms,
- provide test signals and metering,
- perform the actual measurements and store the results.

ALIKI will handle up to 8 audio inputs. So it is possible to record e.g. an Ambisonics B-format, a stereo pair and a binaural format in one operation. Capturing the IRs can be also be done without using ALIKI, e.g. by using Ardour to play the sweep file and record the microphone signals (ALIKI will read Ardour's Broadcast Wave files).

4.2 The deconvolution module

This module reads the recorded waveforms and the inverse filter file, and calculates the actual impulse responses. It uses a fast FFT-based convolution algorithm. An optional correction

[5] At the time of writing, the GUI is still in full development, therefore no screenshots are yet available.

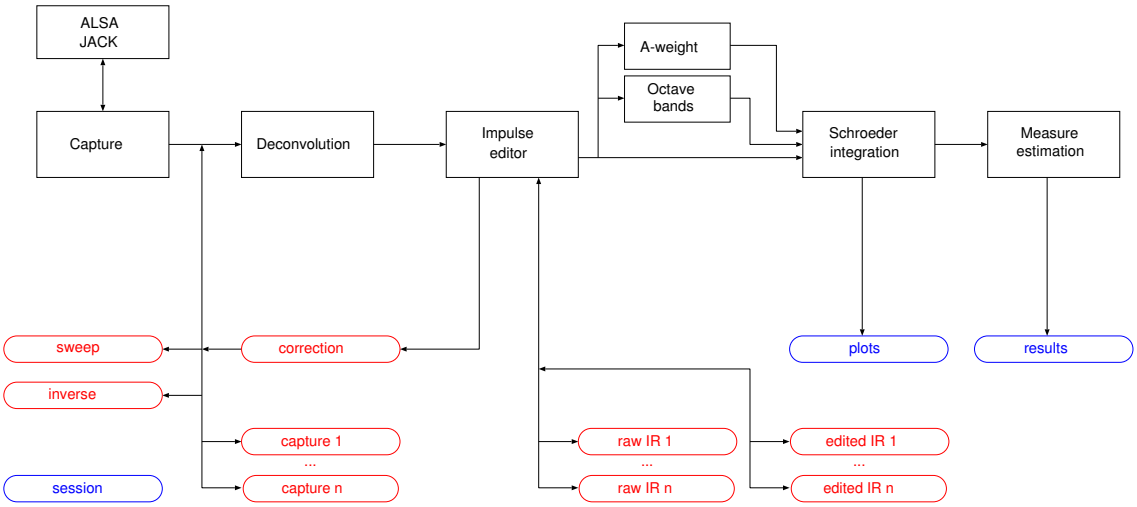

Figure 2: ALIKI program structure and files

filter compensating for the response of the loudspeaker and/or microphone can be used.

The results are saved to the raw impulse response files, and transferred to the editor module.

4.3 The editor module

This provides visualisation and basic editing of impulse responses. It is used to

- normalise the IR to a standard level,
- calibrate the time axis (i.e. put the direct sound at $t = 0$),
- trim the end of the IR to remove noise,
- remove the direct sound if required.

The editor will operate on groups of impulse responses (e.g. a stereo pair or B-format) preserving relative timing and levels. Edited IRs can be saved for later use. Unless the user really wants it, the editor will never overwrite the original waveforms.

This module has one additional function: the first few milliseconds of an IR can be used to compute an inverse FIR filter (up to 4096 taps) that will be used by the deconvolution engine to compensate for the response of the speaker and microphone. It uses a simple FFT-based inversion method, and an interactive procedure steered by the user in order to avoid major errors that could result from a simple automated calculation.

4.4 The filter, integration and measure modules

The remaining modules in fig.2 are closely integrated from the user's point of view.

The Schroeder integral can be computed and visualised for the filtered or full-range IRs. A-weighing and octave band filtering is again performed using FFT-base convolution, and is combined with the backwards integration. The integrals are stored with a resolution of about 1 millisecond. They can be exported in a format readable by applications such as Gnuplot, which can convert them to a number of graphical formats.

The first release of ALIKI computes EDT, RT_{20}, RT_{30}, RT_{user}, T_S, $C_{50,80,user}$, $D_{50,80,user}$, LF and LFC. Others (such as IACC) maybe added in future versions.

All measured values can be exported as text, CSV, or Latex table format for use in spreadsheets or reports.

5 Acknowledgements

The IR measurement method used by ALIKI is based on the work of Prof. Angelo Farina (Dipartimento Ingegneria Industriale, University of Parma, Italy). His many papers (freely available via his website [6]) are required reading for anyone exploring the subject of acoustical IR measurement, in particular in the context of surround sound.

Also the work of Anders Torger (author of BruteFir [7]) and Denis Sbragion (author of

[6] http://pcfarina.eng.unipr.it

[7] http://www.ludd.luth.se/ torger/brutefir.html

DRC) has been very inspiring.

References

Angelo Farina. 2000. Simultanuous measurement of impulse response and distortion with a swept-sine technique. *Audio Engineering Society Preprint 5093*.

ISO TC43/SC2. 2003. Acoustics — Measurement of the reverberation time — Part 1: Performance spaces. ISO CD 3382-1.

ISO TC43/SC2. 2004. Acoustics — Application of new measurement methods in building acoustics. ISO WD 18233.

John Vanderkooy. 1994. Aspects of MLS measuring systems. *Journal of the Audio Engineering Society*, 42(4):219–231.

Unbounded: Aos & The GoingPublik Software

Arthur Clay
Dept. of Computer Science
Clausiusstrasse 59,
CH-8092 Zurich, Switzerland
arthur.clay@inf.ethz.ch

Thomas Frey, Dr.
Dept. of Computer Science
Clausiusstrasse 59,
CH-8092 Zurich, Switzerland
thomas.frey@alumni.ethz.ch

Jürg Gutknecht, Dr. Prof.
Dept. of Computer Science
Clausiusstrasse 59,
CH-8092 Zurich, Switzerland
jürg.gutknecht@inf.ethz.ch

ABSTRACT

GoingPublik is a work for distributed ensemble and wearable computers. The core idea behind the work is a strategy of mobility employing a wearable computer system running a software based electronic scoring system. The score allows for 'composed improvisation', which permits improvisational elements within a compositional structure. By electronically monitoring the performer's physical positions during performance using universal inputs such as geographical positions obtained via satellites and sensors using the earth's magnetic field, the score makes suggestions to various degrees and times. This paper shows how electronic scoring can be self-regulating and depicts how performers using it are able to interact with one another and to create a unique choreographic dispersion of sound in space [1].

Keywords

Wearable Computers, Score Synthesis, HCIs, Aos, Bluebottle, Linux, Q-bic

1. INTRODUCTION

In GoingPublik sonic coherency is accomplished through a theory of 'distribution'. All of the electronic scoring systems used are matched and share similar sensor inputs, (3d-compass and GPS) which are the common denominator to virtually linked them. So despite the physical distribution, commonly shared elements can be structurally exploited. For example, at moments of close proximity between performers synchronized "tutti like" group movements such as rotation bring about synchronized changes in the score. The compositional quantities and qualities of the work are thereby based on spatial mobility; Intensity of form is held by changes in timbre and rhythmic modulations are brought about in conjunction with the sound distribution [2].

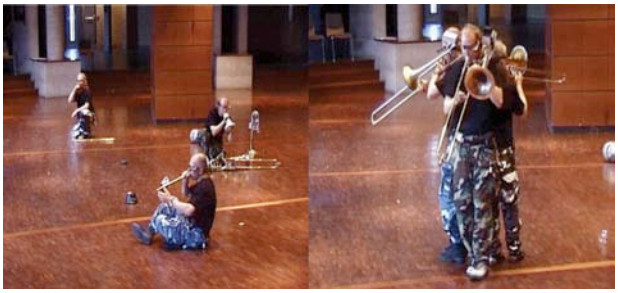

Figure 1. Members of the ensemble in "distributed" and "huddle" formations. Each player is wearing a Q-bic computer, sensor network and display glasses.

2. SOFTWARE & HARDWARE

The system hardware in its current form comprises a StrongARM/XScale based proprietary wearable computer (Q-bic)[2]; a custom made micro programmed 3D compass sensor, a Garmin GPS device, and a Micro Optical SV-6 head-mounted display. The main tasks of the wearable computer is reading the sensor data and computing the score in real time according to predefined rules. The scoring application is programmed in a Pascal-like language called Active Oberon[4]. It runs on Bluebottle (Aos)[5], a lean, open source system kernel enhanced by a highly efficient 2.5D graphics engine that supports sophisticated visual effects on the basis of general-purpose hardware.

3. BEYOND OPEN SOURCE

3.1 Open Sources and Useable Toolboxes

Open source software is like an open toolbox. It is a necessary but often insufficient step towards truly "malleable" software. What is actually needed is mastery and full control of the tools in the box. ETH's integrated programming language Oberon has been available as open source since its invention. Oberon differs from many comparable systems by its simplicity, minimalism and conceptual uniformity in the spirit of Niklaus Wirth's widely acknowledged lean-system tradition. The newest evolution along this line is the Active Oberon language and runtime. As an innovation, Active Oberon introduces a new computing model based on interoperable objects with encapsulated active behavior. This model is ideally suited for programming "the new media", and it easily scales up to distributed systems. This issue addresses the growing interest in the use of computers in the new arts in general and the quite apparent benefits of custom software design in particular. Using the "Going Publik" project software as a proof of concept, we shall argue in favor of both application-aware runtime kernels and small ad-hoc languages as an effective alternative to widely spread graphic builders.

3.2 The Lean versus the Fat System

The reusability of typical open source software is not guaranteed by its openness alone. Even if all the source code that is needed to rebuild an application is provided, it is often very difficult to reuse parts of a project for new purposes. Even reducing the functionality of an open source system can be difficult because of non-modular or inconsequent design. The C based programming languages typically used in open source projects does not encourage a clean modularization and can easily result in header-file incoherencies whose correlation can only be grasped after a long period of review. One could consider the Linux kernel as an example. Although its system is

"open", it is at the same time "closed", because only few people are able to really contribute to its development (in fact, all changes in the Linux kernel are still made by the original author, Linus Torvalds). Of course, kernels are not simple topics and this is not really that different in the case of Aos. However, in a lean and compact system like Aos there are simply fewer unnecessary pitfalls and time consuming debug sessions.

To continue the comparison between Linux (a well know system) and Aos (a less known system) in order to better understand the points made above, one can state that the Linux development can profit largely from a massive amount of manpower and available tools, where as Aos is more limited in this regard. However, most of the tool chain used for Linux development would not be necessary or would have been much easier to develop for a small and completely compile-time type safe programming language such as Active Oberon. Although the available Linux tools and the number of C oriented programmers who are willing to program far outweigh the time lost on insufficiencies of the used programming language, there is still no progress in obtaining malleable and more coherent tools for the future.

To improve code reusability and understandability, ETH follows the lean system initiative. Lean systems have a well-designed, clear and modular structure based on a unifying set of concepts. Lean systems are teachable and therefore understandable. With a deep understanding of a software layer, it becomes possible to adapt it to new needs or to only port or implement the needed parts that are essential in solving a given problem. Having the possibility of leaving out unneeded parts not only improves resource efficiency of a program, but also reduces the number of possible errors and potential exploits.

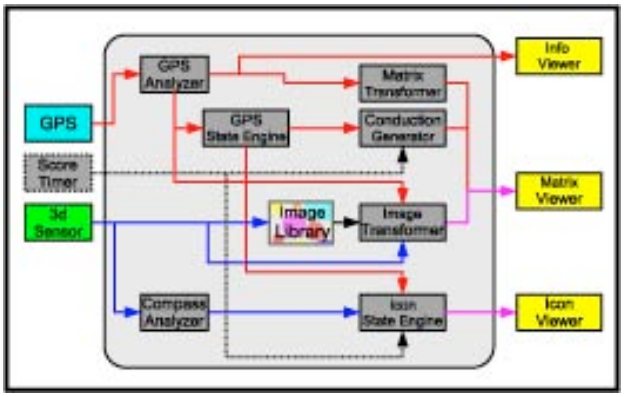

Fig. 2. Depicted is a schematic showing the relationships between the sensor systems and the component packages of the GP software.

4. THE SOFTWARE

4.1 Modularity and Malleability

Contained in the GoingPublik software package are eight modules, which can be more or less included or excluded during runtime. The elements depicted in Fig.2**.** within the rectangle are the software components, the elements depicted to the left are the sensor systems, and the elements depicted to the right are those drawn into the screen areas of the display glasses. The GPS and the 3d Compass sensor are connected per Bluetooth and are mounted as wearable sensors on the performer's clothing. The graphic elements drawn by the Info Viewer, the Matrix Viewer and the Icon Viewer modules are combined in the display glasses into a single score as depicted in Fig.3. To exemplify the system's malleability: When the software module "Icon State Engine" is not needed for the performance version without the behavioral icons, then it would simple be left out of the software package[1]. The icons would not appear in the viewer and the other modules would not be affected in any way and would operate as expected.

5. THE MATRIX WINDOW

5.1 The Modulating Matrix

The basis of the electronic score is a modulating matrix. The resolution of the matrix is determined by the performer's position within the performance space, which is obtained via GPS satellites or generated by a GPS simulator. In either case, the received GPS string is parsed and given further as x, y value pairs that reference the position within a predefined area. By moving within this area, the performer influences the position of the matrix's lines, therefore continuously adjusting the 'resolution' of it to parameterize sonic domains with frequency and time values. The 'Range-Lines' of the matrix move on the horizontal plane in relation to the North-South axis; the 'Time-Lines' move on the vertical plane in relation to the West-East axis. The Time-Lines move in contrary motion and modulate the spaces between the lines into equidistant and non-equidistant states. The 'Conduction-Arm' travels through the matrix from the left to facilitate score reading. The time taken by the Conduction-Arm to scan through the space between two Time-Lines is always a constant value in milliseconds (independent of the distance), but is dependent on walking speed measured in meters per minute. There are four discrete tempi: Rest, Relax, Work and Hurry. The speed of the Conduction-Arm therefore makes a quantitative difference in the amount of time the performers may 'stay' on an area of the score image.

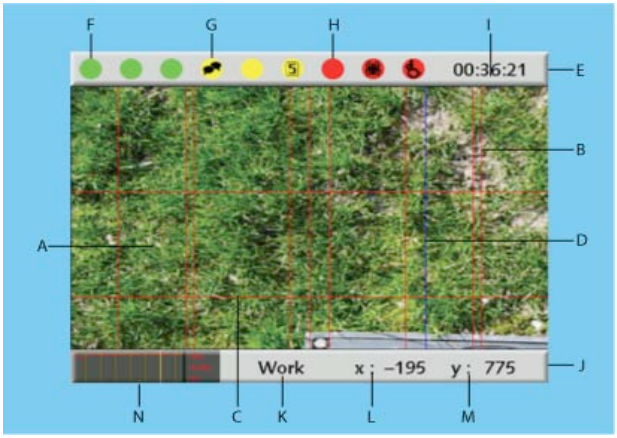

Figure 3. The Matrix Window: (A) Score Image, (B) TimeLines, (C) RangeLines, (D) Conducting Arm, (E) IconBar, (F) GoIcons, (G) ModIcons, (H) StopIcons, (I) Timer, (J) TempoBar, (K) Tempo, (L) GPS x-Coordinate, (M) GPS y-Coordinate, (N) Activity Graph.

The movement of the Range-Lines brings about equidistant

[1] Such a malleable system could also be realized by a program written in C, but its lack of modularization concepts on the language level would require more efforts of the programmer.

spaces, which limit and expand the instrumental range based on changes of position within the space. When all Range-Lines are present, seven range spaces can be seen. The available ranges would then be as follows: Outside, Very Low, Low, Middle, High, Very High and Outside. Ranges are always kept in consecutive order, the performers freely choosing the lowest initial range first and then continue upward from there. The performers decide where the boundaries the instrumental ranges are and what is meant by 'outside' the instrument.

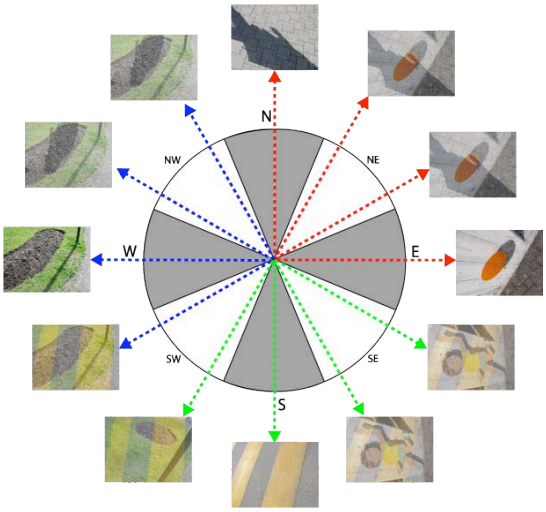

Fig. 4. The 360° of Directional Imaging: Single score images are at the poles and complex images are between these poles.

5.2 Directional Imaging

There are four score images and each is assigned a direction. A discrete resolution of eight possible 'headings' is used and these values determine the score image. Single score images are rendered at the poles of the compass and superimpositions between these. The 3d-compass also measures 'pitch' and 'roll', whose values distort the score image to create 'variations'. The larger the intensity of pitch and roll is, the greater the distortion of the score image is. The size of the displayed score image is dependent on walking activity. This is calculated using speed average over a given period of time. If the performer is 'standing' more than 'walking', the image will enlarge up to 200%; if the performer is 'walking' more than 'standing', the image will shrink back to its original size. Variations in sound material therefore not only arise on account of the Conduction-Arm speed but also due to score image distortion and changes in size.

Figure 5. The Icon Menu Bar. From left to right are three blank Go-Icons, the Mod-Icon for "Medium Playing Density", a blank Mod-Icon, the Mod-Icon for "Phrasing in Groups of Five", a blank Stop-Icon, the Stop-Icon for "Stop and Hide", and lastly the Stop-Icon for "Sit Down".

5.3 The Action Icons

Three groups of three icons contained on the "icon bar" are used to suggest actions to the performer. The green 'Go-Icon' and the red 'Stop-Icon' groups suggest changes in walking speed, the time spent doing so, and a random component. Related performative actions are associated with each of the icons to artistically integrate changes in walking activity, regulate the tempo in general and to integrate the performer's sonically into the environment.

Based on the rate of heading change, walking speed and a random component, the 'Mod-Icons' suggest how the score is to be read by designating parameters of 'style'. Here, eye movement through the matrix is confined by phrasing rules. These rules are PHRASE (the division of the matrix into units of material), PATH (the form of the curve used to read through the matrix) and PLAY (the degree of density in playing while reading through the matrix). By interpreting the score in this manner, contrapuntal differences between the performers are brought about, so that 'sonic windowing' is created through which unoccupied audio space and variation in voice density are guaranteed.

6. THE MAPPING WINDOW

6.1 Performance Modes

There are two software based performance modes: An 'indoor' mode is for closed spaces and an 'outdoor' mode for open spaces. The indoor mode relies on a route simulator and the outdoor mode relies on GPS satellite information to make changes to the matrix and icons. The software automatically switches between route simulator and GPS satellite information dependent on satellite reception so that a performance may take place in and between closed and open spaces. To use the route simulator, each player draws a predetermined route onto the provided map with a handheld mouse. When finished, the performer presses the range button and switches back to the Matrix-Window.

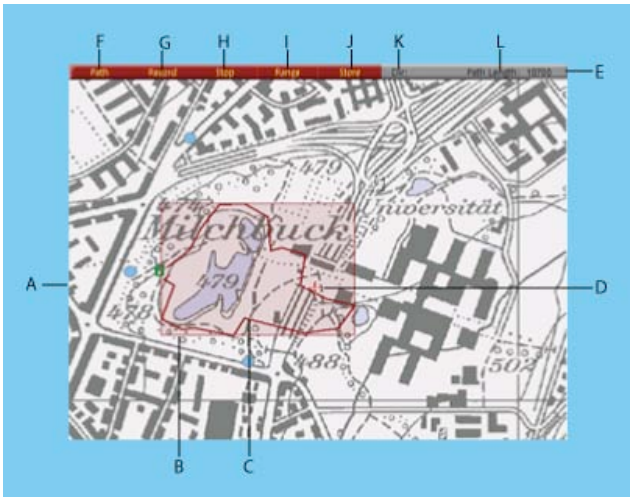

Figure 6. The Mapping Window: (A) Map Area, (B) Range Box, (C) Performer's Route, (D) X-Bow Position Indicator, (E) InfoBar, (F) Path Selector Button, (G) Path Record Button, (H) Stop- Record Button, (I) Calculate Range Button, (J) Store Route Button, (K) Direction Indicator, (L) Length Indicator.

6.2 Synchronization Types

There are two types of synchronization exploited in the work: 'local' and 'global'. Local synchronization is made possible by

the 3d compass sensor data and takes place when performers are "huddled" and change heading, pitch and tilt readings together, thus changing the score image at the same time and to the same degree. For an 'inside' performance the performers dismantle their instruments and spread the parts across the performance space. This action emphasizes the effect of distribution visually and creates a predetermined task that results in a predictable choreography of movements. The system sensors respond accordingly, and the electronic score changes in conjunction to all movements made as the performers re-assemble their instruments. Global synchronization is made possible via GPS and takes place via Conduction-Arm synchronization and when there is some form of coordination of routes between two or more performers. For an outside' performance three routes, one for each of the performers, are roughly designated. The greater the distance between the performers, the more varied their scores will appear; the lesser the distance between them, the more similar their scores will appear. So, when the separation between performers diminishes as they come to meet, their scores slowly grow in similarity until each score matrix is similar [6]. As listener, a gradual morphing from individual to collective sound masses can easily be heard.

7. FURTURE WORK

7.1 GPS Time Mark Function

In the next version of the GoingPublik software, the GPS Time Mark will be used to synch an on-board clock housed on the Q-bic computer. The GPS Time Mark arrives at the exact same moment for all of the performers of the distributed ensemble and this event makes it possible to synchronize the movement of the Conduction- Arms of all of the players. To employ it aesthetically as a compositional mechanism, the concept of "global synchronization" is used again as a uniting force. Each of the four tempi used in the scoring system are in ratio to one another: Tempo "Relax" is therefore 2 * Tempo "Rest", Tempo "Work" is 3 * Tempo "Rest" and Tempo "Hurry" is 4 * Tempo "Rest". The speed of the Conduction Line for the slowest tempo, "Rest" is used as the basis for determining the speeds of the other three tempi used in the score.

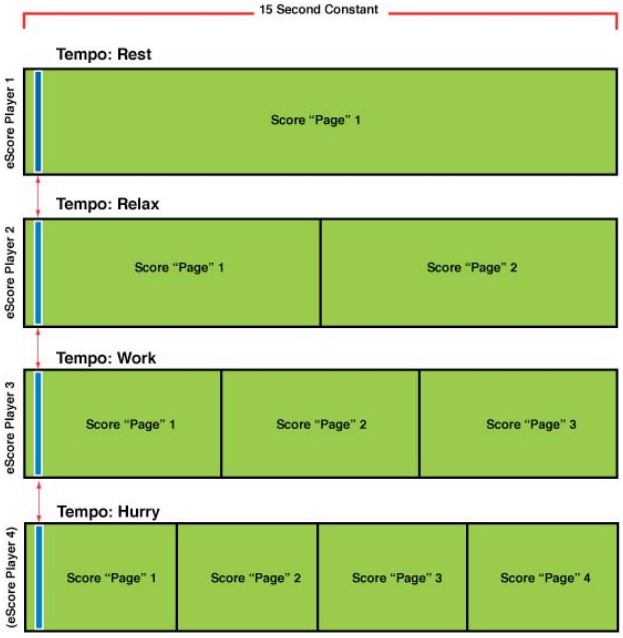

Figure 7. The figure illustrates how the Conducting Line (here indicated in blue) can be synchronized in relation to tempo for each of four players.

What by "page" is meant is the width of the score image. So a tempo is always in terms of how long it takes the Conduction-Line to get through one "page" of the score, One page is the therefore the time it takes for the Conduction-Line to transverse the entire score image. In this way all of the Conduction-Arms of all players stay synchronized and will always meet (after some number of pages) at the "beginning" of a score "page" just as certain polyrhythmic groupings meet at the initial beat they began at. This feature was tested outside of the GoingPublik in the work China Gates for gongs and GPS-Wrist Controller [7].

7.2 Collaborative Score

The GoingPublik system allows for the unique opportunity for composers and performers to research the possibilities of collaborative scores. Through changing the .xml file that is used to preset the software, it is possible to designate which score images are to be loaded into the matrix. Having each of the players or each member of a group of composers prepare a single score image for the system, would bring about a system score which would consist of four distinct scores, which would then "integrate" into a single whole via the system and who it works to parameterize and vary the score images during a performance. This possibility of collaboration demonstrates how the GoingPublik software is not a single work, but system, which can be used to interpret a body of, works written for it and also how it might be used to research collaboration in general amongst performers and composers.

8. CONCLUSION

The movements made by the ensemble players can be understood as choreographic patterns having an internal system of counter-point: 'bundled movements', or synchronized movements made together are analogue to polyphony in similar motion and 'free movement' or non synchronized movement carried out in non-relationship to one another are analogue to contrary motion. Therefore, a parallel can be drawn between the distribution amount of the performers and the degree of 'dissonance' in terms of rhythmic and range discord existing between them.

A slight comparison between Linux and Aos system software was drawn. The comparison did not point to the better system, but was intended to serve the purpose of proposing a focus on new paradigms of computer science, in order to develop languages that lead to more malleable and understandable tools. As an innovation, the Active Oberon language was introduced as such a new computing model, which is based on interoperable objects with encapsulated active behavior. By using the GoingPublik project as a proof of concept, it was argued that this computing model is ideally suited for programming need in the arts, in that it was made quite apparent that custom software design in the new arts has become quite common as has the needed adaptability of the that software for further developments of the same an new art works.

9. ACKNOWLEDGEMENT

Our thanks to Prof. Dr. Paul Lukowicz (Q-bic), Stijn Ossevort (Belt Design), Dr. Tom Stricker (Consulting), Dr. Emil Zeller (ARM Aos Porting), Mazda Mortasawi (Additional Programming) and Dr. Dennis Majoe of *MASC*, London (Sensor Systems).

10. REFERENCES

[1] http://homepage.mac.com/arthurclay/FileSharing8.html

[2] Clay, A/Frey, T/Gutknecht, J 2005: GoingPublik: Using Real-time Global Score Synthesis, in*: Proceedings NIME 2005*, Vancouver, Canada.

[3] Amft, O/Lauffer, M/Ossevort, S/Macaluso, F/Lukowicz, P/Tröster, G 2004: Design of the QBIC Wearable Computing Platform, in: *Proceedings of the 15th IEEE International Conference on Application-specific Systems, Architectures and Processors*, Galveston (Texas).

[4] Muller, P 2002: The Active Object System - Design and Multiprocessor Implementation. PhD thesis, Institute for Computer Systems, ETH Zurich.

[5] http://www.bluebottle.ethz.ch/

[6] Clay, A/Frey, T/Gutknecht, J 2004: GoingPublik: GoingPublik: Suggesting Creativity Inside the Ivory Tower, in*: Proceedings C&C 2004*, London

[7] Clay, A/Majoe D 2006: China Gates: A Work in Progress, in *Proceedings* 3rd International Mobile Muisc Workshop, Brighton England

[8] Martinsen, F 2004: Compositie in Camouflage, over de verhouding tussen interpretatie en improvisatie, in: *Cutup Magazine*, http://www.cut-up.com

Realtime Audio vs. Linux 2.6

Lee Revell
Mindpipe Audio
305 S. 11th St. 2R
Philadelphia, PA, 19107
USA,
rlrevell@joe-job.com

Abstract

From the beginning of its development kernel 2.6 promised latency as low as a patched 2.4 kernel. These claims proved to be premature when testing of the 2.6.7 kernel showed it was much worse than 2.4. I present here a review of the most significant latency problems discovered and solved by the kernel developers with the input of the Linux audio community between the beginning of this informal collaboration in July 2004 around kernel 2.6.7 through the most recent development release, 2.6.16-rc5. Most of these solutions went into the mainline kernel directly or via the -mm, voluntary-preempt, realtime-preempt, and -rt patch sets maintained by Ingo Molnar (Molnar, 2004) and many others.

Keywords

Latency, Preemption, Kernel, 2.6, Realtime

1 Introduction

In mid-2004 Paul Davis, and other Linux audio developers found that the 2.6 kernel, despite promises of low latency without custom patches, was essentially unusable as an audio platform due to large gaps in scheduling latency. They responded with a letter to the kernel developers which ignited intense interest among the kernel developers (Molnar, 2004) in solving this problem. Massive progress was made, and recent 2.6 releases like 2.6.14 provide latency as good or better than the proprietary alternatives. This is a review of some of the problems encountered and how they were solved. ...

2 Background

The main requirements for realtime audio on a general purpose PC operating system are application support, driver support, and low scheduling latency. Linux audio began in earnest around 2000 when these three requirements were met by (respectively) JACK, ALSA, and the low latency patches for Linux 2.4 ("2.4+ll"). The 2.6 kernel promised low scheduling latency (and therefore good audio performance) without custom patches, as kernel preemption was available by default. However early 2.6 kernels (2.6.0 through approximately 2.6.7) were tested by the Linux audio development community and found to be a significant regression from 2.4+ll. These concerns were communicated privately to kernel developer Ingo Molnar and 2.6 kernel maintainer Andrew Morton; Molnar and Arjan van de Ven responded in July 2004 with the "Voluntary Kernel Preemption patch" (Molnar, 2004). The name is actually misleading - 'Voluntary' only refers to the feature of turning might_sleep() debugging checks into scheduling points if preemption is disabled. The interesting features for realtime audio users, who will always enable preemption, are the additional rescheduling points with lock breaks that Molnar and van de Ven added wherever they found a latency over 1ms.

3 Latency debugging mechanisms

The first requirement to beat Linux 2.6 into shape as an audio platform was to develop a mechanism to determine the source of an xrun. Although kernel 2.6 claims to be fully preemptible, there are many situations that prevent preemption, such as holding a spinlock, the BKL, or explicitly calling preempt_disable(), or any code that executes in hard or soft interrupt context (regardless of any locks held).

The first method used was ALSA's "xrun debug" feature, about the crudest imaginable latency debugging tool, by which ALSA simply calls dump_stack() when an xrun is detected, in the hope that some clue to the kernel code path responsible remains on the stack. This crude mechanism found many bugs, but an improved method was quickly developed.

In the early days of the voluntary preemption patch, Molnar developed a latency tracing mechanism. This causes the kernel to trace every function call, along with any operation

that affects the "preempt count". The preempt count is how the kernel knows whether preemption is allowed - it is incremented or decremented according to the rules above (taking spinlock or BKL increments it, releasing decrements, etc) and preemption is only allowed when the count is zero. The kernel tracks the maximum latency (amount of time the preempt count is nonzero) and if it exceeds the previous value, saves the entire call stack from the time the preempt count became positive to when it became negative to /proc/latency_trace).

So rather than having to guess which kernel code path caused an xrun we receive an exact record of the code path. This mechanism has persisted more or less unchanged from the beginning of the voluntary preemption patches (Molnar, 2004) to the present, and within a week of being ported to the mainline kernel had identified at least one latency regression (from 2.6.14 to 2.6.15, in the VM), and has been used by the author to find another (in free_swap_cache()) in the past week. Dozens of latency problems have been fixed with Molnar's tracer (everything in this paper, unless otherwise noted); it is the one of the most successful kernel debugging tools ever.

4 The BKL: ReiserFS 3

One of the very first issues found was that ReiserFS 3.x was not a good choice for low latency systems. Exactly why was never really established, as the filesystem was in maintenance mode, so any problems were unlikely to be fixed. One possibility is that reiser3's extensive use of the BKL (big kernel lock - a coarse grained lock which dates from the first SMP implementations of Linux, where it was used to provide quick and dirty locking for code with UP assumptions which otherwise would have to be rewritten for SMP). ReiserFS 3.x uses the BKL for all write locking. The BKL at the time disabled preemption, which is no longer the case, so the suitability of ReiserFS 3.x for low latency audio systems may be worth revisiting. Hans Reiser claims that ReiserFS 4.x solves these problems.

5 The BKL: Virtual console switching

One of the oldest known latency issues involved virtual console (VC) switching (as with Alt-Fn), as like ReiserFS 3.x this process relies on the BKL for locking which must be held for the duration of the console switch to prevent display corruption. This problem which had been known since the 2.4 low latency patches was also resolved with the introduction of the preemptible BKL.

6 Hardirq context

Another issue discovered in the very early testing of the voluntary preemption patches was excessive latency caused by large IO requests by the ATA driver. It had previously been known that with IDE IO completions being handled in hard IRQ context and a maximum request size of 32MB (depending on whether LBA48 is in effect which in turn depends on the size of the drive), scheduling latencies of many milliseconds occurred when processing IO in IRQ context.

This was fixed by adding the sysfs tunables:
/sys/block/hd*/queue/max_sectors_kb
which can be used to limit the amount of IO processed in a single disk interrupt, eliminating excessive scheduling latencies at a small price in disk throughput.

Another quite humorous hardirq latency bug occurred when toggling Caps, Scroll, or Num Lock - the PS/2 keyboard driver actually spun in the interrupt handler polling for LED status (!). Needless to say this was quickly and quietly fixed.

7 Process context - VFS and VM issues

Several issues were found in the VFS and VM subsystems of the kernel, which are invoked quite frequently in process context, such as when files are deleted or a process exits. These often involve operations on large data structures that can run for long enough to cause audio dropouts and were most easily triggered by heavy disk benchmarks (bonnie, iozone, tiobench, dbench).

One typical VFS latency issue involved shrinking the kernel's directory cache when a directory with thousands of files was deleted; a typical VM latency problem would cause audio dropouts at process exit when the kernel unmapped all of that processes virtual memory areas with preemption disabled. The sync() syscall also caused xruns if large amounts of dirty data was flushed.

One significant process-context latency bug was discovered quite accidentally, when the author was developing an ALSA driver that re-

quired running separate JACK instances for playback and capture. A large xrun would be induced in the running JACK process when another was started. The problem was identified as mlockall() calling into make_pages_present() which in turn called get_user_pages() causing the entire address space to be faulted in with preemption disabled.

Process-context latency problems were fortunately the easiest to solve, by the addition of a reschedule with lock break within the problematic loop.

8 Process context - ext3fs

While ReiserFS 3.x did not get any latency fixes as it was in maintenance mode, EXT3FS did require several changes to achieve acceptable scheduling latencies. At least three latency problems in the EXT3 journalling code (a mechanism for preserving file system integrity in the event of power loss without lengthy file system checks at reboot) and one in the reservation code (a mechanism by which the filesystem speeds allocation by preallocating space in anticipation that a file will grow) were fixed by the maintainers.

9 Softirq context - the struggle continues

Having covered process and hardirq contexts we come to the stickiest problem - softirqs (aka "Bottom Halves", known as "DPCs" in the Windows world - all the work needed to handle an interrupt that can be delayed from the hardirq, and run later, on another processor, with interrupts enabled, etc). Full discussion of softirqs is outside the scope (see (Love, 2003)) of this paper but an important feature of the Linux implementation is that while softirqs normally run immediately after the hardirq that enabled them on the same processor in interrupt context, under load, all softirq handling can be offloaded to a "softirqd" thread, for scalability reasons.

An important side effect is that the kernel can be trivially modified to unconditionally run softirqs in process context, which results in a dramatic improvement in latency if the audio system runs at a higher priority than the softirq thread(s). This is the approach taken by the -rt kernel, and by many independent patches that preceded it.

The mainline Linux kernel lacks this feature, however, so minimizing scheduling latency requires limiting the amount of time spent in softirq context. Softirqs are used heavily by the networking system, for example looping over a list of packets delivered by the network adapter, as well as SCSI and for kernel timers (Love, 2003). Fortunately the Linux networking stack provides numerous sysctls that can be tuned to limit the number of packets processed at once, and the block IO fixes described elsewhere for IDE also apply to SCSI, which does IO completion in softirq context.

Softirqs are the main source of excessive scheduling latencies that, while rare, can still occur in the latest 2.6 kernel as of this writing (2.6.16-rc5). Timer based route cache flushing can still produce latencies over 10ms, and is the most problematic remaining softirq as no workaround seems to be available; however the problem is known by the kernel developers and a solution has been proposed (Dumazet, 2006).

10 Performance issues

The problems described so far mostly fit the pattern of too much work being done at once in some non-preemptible context and were solved by doing the same work in smaller units. However several areas where the kernel was simply inefficient were resolved, to the benefit of all users.

One such problem was kallsyms_lookup(), invoked in cases like printk(), which did a linear search over thousands of symbols, causing excessive scheduling latency. Paulo Marques solved this problem by rewriting kallsyms_lookup() to use a more efficient search algorithm. The frequent invocation of SHATransform() in non-preemptible contexts to add to the entropy pool was another latency problem solved by rewriting the code to be more efficient.

11 Non-kernel factors

The strangest latency problem identified was found to have an origin completely outside the kernel. Testing revealed that moving windows on the desktop reliably caused JACK to report excessive delays. This is a worse situation than an xrun as it indicates the audio device stopped producing/consuming data or a hardware level timing glitch occurred, while an xrun merely indicates that audio was available but JACK was not scheduled in time to process it. The problem disappeared when 2D acceleration was disabled in the X configuration which pointed clearly to the X display driver - on Linux all

hardware access is normally mitigated by the kernel except 2D XAA acceleration by the X server.

The VIA Unichrome video card used in testing has a command FIFO and a status register. The status register tells the X server when the FIFO is ready to accept more data. (Jones and Regehr, 1999) describes certain Windows video drivers which improve benchmark scores by neglecting to check the status register before writing to the FIFO; the effect is to stall the CPU if the FIFO was full. The symptoms experienced were identical to (Jones and Regehr, 1999) - the machine stalled when the user dragged a window. Communication with the maintainer of the VIA unichrome driver (which had been supplied by the vendor) confirmed that the driver was in fact failing to check the status register and was easily fixed.

12 The -rt kernel and the future

The above solutions all have in common that they reduce scheduling latencies by minimizing the time the kernel spends with a spinlock held, with preemption manually disabled, and in hard and soft IRQ contexts, but do not change the kernels behavior regarding which contexts are preemptible. Modulo a few remaining, known bugs, this approach is capable of reducing the worst case scheduling latencies to the 1-2ms range, which is adequate for audio applications. Reducing latencies further required deep changes to the kernel and the rules about when preemption is allowed. The -rt kernel eliminates the spinlock problem by turning them into mutexes, the softirq by the softirq method previously described, and the hardirq issue by creating a set of kernel threads, one per interrupt line, and running all interrupt handlers in these threads. These changes result in a worst case scheduling latency close to 50 microseconds which approaches hardware limits.

13 Conclusions

One of the significant implications of the story of low latency in kernel 2.6 is that I believe it vindicates the controversial "new kernel development process" (Corbet, 2004) - it is hard to imagine Linux 2.6 evolving into a world class audio platform as rapidly and successfully as it did under a development model that valued stability over progress. Another lesson is that in operating systems as in life, history repeats itself. Much of the work done on Linux 2.6 to support soft realtime applications, like IRQ threading, was pioneered by Solaris engineers in the early 1990s (Vahalia, 1996).

14 Acknowledgements

My thanks go to Ingo Molnar, Paul Davis, Andrew Morton, Linus Torvalds, Florian Schmidt, and everyone who helped to evolve Linux 2.6 into a world class realtime audio platform.

References

Jonathan Corbet. 2004. Another look at the new development model. https://lwn.net/Articles/95312/.

Eric Dumazet. 2006. Re: Rcu latency regression in 2.6.16-rc1. http://lkml.org/lkml/2006/1/28/111.

Michael B. Jones and John Regehr. 1999. The problems you're having may not be the problems you think you're having: Results from a latency study of windows nt. In *Proceedings of the 7th Workshop on Hot Topics in Operating Systems (HotOS VII)*, pages 96–101.

Robert Love. 2003. *Linux Kernel Development.* Sams Publishing, Indianapolis, Indiana.

Ingo Molnar. 2004. [announce] [patch] voluntary kernel preemption patch. http://lkml.org/lkml/2004/7/9/138.

Uresh Vahalia. 1996. *Unix Internals: The New Frontiers.* Prentice Hall, Upper Saddle River, New Jersey.

A Low-Latency Full-Duplex Audio over IP Streamer

Asbjørn SÆBØ and U. Peter SVENSSON
Centre for Quantifiable Quality of Service in Communication Systems
NTNU – Norwegian University of Science and Technology
O.S. Bragstads plass 2E
N-7491 Trondheim
Norway
asbjs@q2s.ntnu.no, svensson@q2s.ntnu.no

Abstract

LDAS (Low Delay Audio Streamer) is software for transmitting full duplex high-quality multi-channel audio with low end-to-end latency over IP networks. It has been designed and implemented as a tool for research into distributed multimedia interaction and quality of service in telecommunications. LDAS runs on Linux, using the ALSA sound drivers and libraries. It uses UDP as its transport protocol. A flow control scheme is used to keep the sender and receiver synchronised and to deal with transmission errors. Tests have shown end-to-end latencies (from analog input to analog output) down to around five milliseconds over a minimal network.

Keywords

audio, streaming, latency, IP network

1 Introduction

The field of telecommunications is changing, with new technologies making new services possible. One aspect of this is the increasing use of various kinds of transmission of audio and multimedia over packet-switched networks using the Internet Protocol (IP).

Related to this, "Quality of Service" (QoS) is a topic of current interest in telecommunications. Traditionally, this deals with technical specifications and guarantees. A wider interpretation may also take into account the quality of service as experienced by the user. In this respect, an increased quality of service may not only be an increase in the quality of a given service (like voice communication), but to improve the user experience by offering a service of inherently higher quality, something that is not only better, but more. (An example might be to replace a one-channel voice communication service with a voice communication service with 3D-audio.)

There is an untapped potential for services utilising audio and multimedia over IP that would give a better quality of service, as experienced by the user. The ultimate telecommunications system would enable one to achieve virtual *presence* and true *interaction* between the endpoints of the communication. Current common systems, like internet telephony (Voice over IP, VoIP) and video conferencing do not fully achieve this. A primary aspect of this topic is how such services should work, on a technical level. Further, and perhaps even more interesting, is how they may be used, and how the user will experience them.

Our aim is therefore to explore and investigate more advanced and demanding communication services, focusing on the audio side of *distributed multimedia interaction*. The general situation would be one where two acoustical situations are connected through the network in such a way as to achieve transmission of the complete acoustic environments. Examples of such applications, and our immediate targets, are ensemble playing and music performance over the network, and transmission of various kinds of 3D audio (binaural, Ambisonics, multichannel). This has been demonstrated in a number of studies, e.g. (Woszczyk et al., 2005).

Latency is the most interesting and demanding of the factors limiting these kinds of services. Available data storage capacity, data transmission capacity (bandwidth) and processing capacity are all steadily increasing, with no hard bounds in sight. Transmission time, however, is fundamentally limited by the the speed of light. And, as will be further discussed below, low latency is in many cases important and necessary in order to achieve interaction. In addition to high quality overall, low latency should therefore be emphasised.

For our exploration, a tool suitable for such services was needed. This tool should be capable of transmission of high quality, low latency audio. It should also be open, to facilitate control over all aspects and parameters of the transmission process. Several such applications exist, from the simple ones to the very advanced

ones. For many of these, source code is not accessible, making studies of their internals or changing of their workings next to impossible. Others do, in various ways, not meet our needs, although StreamBD, used at CCRMA, (Chafe et al., 2000), might have had adequate properties. The development of such a tool would give valuable insights into the specific problems associated with these topics, give full control of all aspects of the tool and its usage and be a good way to start the explorations. So, the task of developing LDAS - the Low Delay Audio Streamer, was undertaken, using previous work done at NTNU (Strand, 2002) as a starting point.

2 Requirements and specifications

The two immediate target applications are networked ensemble playing and transmission of acoustical environments. The first involves musicians at different locations, connected by a network, performing music together. The second involves the transmission of enough information from one site to another to be able to recreate the acoustic environment of the first site at the second site in a satisfactorily manner. The main requirements of these applications are discussed below.

2.1 Audio quality

Audio quality should be high. For simplicity, the lower limit for quality has been set equal to that of an audio Compact Disc. This means a sampling frequency of 44.1kHz (or 48kHz) or higher, and a sample depth (word length) of at least 16 bits for uncompressed PCM.

The number of channels available should be at least two (which will allow for transmission of stereo or binaural signals), but preferably eight or more (which will allow for various kinds of multi-channel and three-dimensional audio formats like Ambisonics).

2.2 Latency considerations

Latency is important for interaction. In particular, it is known that excessive inter-musician latency is detrimental to ensemble playing, one of our target applications (Bargar et al., 1998). The fundamental latency requirement is therefore that the latency should be so low that it will not hinder successful ensemble playing across the established connection.

Some investigations into what constitutes tolerable delay has been done, but few definite conclusions have been given. Experiments, using hand-clapping as the musical signal, have found an "optimal" delay (with respect to tempo stability) of 11.6 milliseconds (Chafe and Gurevich, 2004). Based upon these data and their own experiences, (Woszczyk et al., 2005) suggest that latencies of 20ms to 40ms are "easily tolerated", and that even higher latencies may be acceptable after training and practice.

Experiments conducted at the acoustics group at NTNU have concluded that latencies lower than 20 milliseconds do not seem to influence the ensemble playing much, while latencies above 20 milliseconds may may lead to a less "tight" rhythm, with the two musicians not following each other as well (Winge, 2003). On the other hand, (Lago and Kon, 2004) claims that latencies up to at least 30 milliseconds should be considered normal and in most situations acceptable, and that latencies of this order will not impair musical performance. Further, informal evidence, based upon the experience of a number of practising musicians, indicates that latencies of 30 ms and maybe up to 50ms may be tolerable.

For comparison: Sound in air at room temperature travels at approximately 345 meters per second. This gives a latency of about three milliseconds per meter. Two musicians two meters apart will experience a delay of six milliseconds. The width of a typical sympony orchestra on stage may be around 15 to 20 meters, corresponding to a latency between the outermost musicians on the order of 50 milliseconds. (It should be noted that an orchestra also has external synchronisation in the form of a conductor.)

Obviously, for networked ensemble playing, network transmission time may make up a large part of the total latency. (With factors like A/D and D/A conversion, buffering in the sound card, data processing and handling by application and OS and travel time for sound waves in air making up for the rest.) Whether a sufficiently low latency is possible is therefore to a large degree dependent upon the "network distance" between the participants.

Based upon the indications above, the operating requirement was set to have an end to end latency (analog signal to analog signal) of less than 20 milliseconds for transmission over a campus-wide LAN. To avoid unnecessary latency, it was decided that uncompressed audio should be transferred. While expending effort into keeping the latency low, it should be re-

membered that latency requirements must be balanced against the robustness of transmission. This compromise should be tunable.

2.3 Network protocol

Using IP (Internet Protocol) as the network layer protocol is a given, since the area for which LDAS is intended is audio over IP.

The two main transport protocols used over IP are TCP (Transmission Control Protocol) and UDP (User Datagram Protocol). TCP provides a reliable, connection-oriented transmission mechanism. Lost packets are retransmitted, and flow and congestion control is part of the protocol. UDP is a connectionless protocol with very limited delivery guarantees. Packets sent via UDP may not arrive at the destination at all, they may arrive out of order, or they may arrive as duplicates. The only guarantee given is that if a packet does arrive, its contents are intact. UDP supports broadcasting and multicasting (which TCP does not), but does not have built in flow and congestion control. (Stevens et al., 2004).

For networked ensemble playing, the reliability of TCP is not called for. A low latency is deemed more important than totally reliable delivery of all data. Some data loss may be acceptable, and retransmission of lost packets may be a waste of time and capacity, as they may well be discarded due to arriving too late when they finally arrive. Neither will TCP's flow and congestion control be the optimal way to rate-limit LDAS traffic. Methods taking into account the nature of the service, built into the application itself, will be preferable. (Until flow control is implemented in LDAS, LDAS traffic will not be TCP-friendly. But, at least for the research purposes and situations, this is acceptable.)

As multi-way communication is envisioned, it is possible, maybe also probable, that LDAS may be extended to multicast, for which UDP is needed. As the distinguishing features of TCP are not necessary for LDAS, not choosing TCP as the transport protocol is not a disadvantage. On the other hand, the multicast feature of UDP may be essential. On the basis of this, UDP was chosen as the transport protocol.

It is necessary for LDAS that the order of packets be maintained and the relative placement of packets in time be correct. This feature is not delivered by UDP. LDAS should therefore implement its own protocol, on top of UDP, to provide the necessary features for packet stream tracking and control.

Alternative protocols are DCCP (Datagram Congestion Control Protocol, `http://www.icir.org/kohler/dcp/`) and RTP (Real Time Protocol, `http://www.faqs.org/rfcs/rfc3550.html`). DCCP is intended as a substitute for UDP for applications like these, but is still a work in progress. RTP sits on top of UDP, and is a transport protocol for real time applications. It was, however, found to be more complex than needed for this case.

2.4 Synchronisation

It is mandatory that the sending and the receiving parts be kept synchronised, keeping the latency as low and as constant as circumstances will allow. To minimise the effects of network transmission time jitter, the receiving part should maintain a buffer of received audio data. The amount of data to be held in this buffer should be setable, so it can be balanced against the latency requirements.

3 Implementation

The high-level system architecture is currently a "pairwise peer-to-peer" system, i.e. two equal programs sending and receiving data to and from each other.

The structure of the application is shown in figure 1. There are three threads running in parallel, a recorder/sender, a receiver and a playback task. The recorder/sender thread is running independently of the two other threads, while the receiver and the playback threads are linked through a common data structure, the *receiver queue*. All three threads are running scheduled as SCHED_FIFO tasks.

3.1 The data stream and the packet format

Audio is a continuous signal, which is digitised by the audio interface into a stream of samples. It is, however, practical to handle chunks of the stream as units when processing and transmitting the data. The audio stream is therefore divided into a series of application level *packets*.

This packet stream is the connection between the sender and the receiver. The packet format is quite simple. Each packet consists of one *period*[1] of audio data, as delivered from the audio interface. (Currently, the ALSA interleaved format is used.) To these data is appended a

[1] In the ALSA sense.

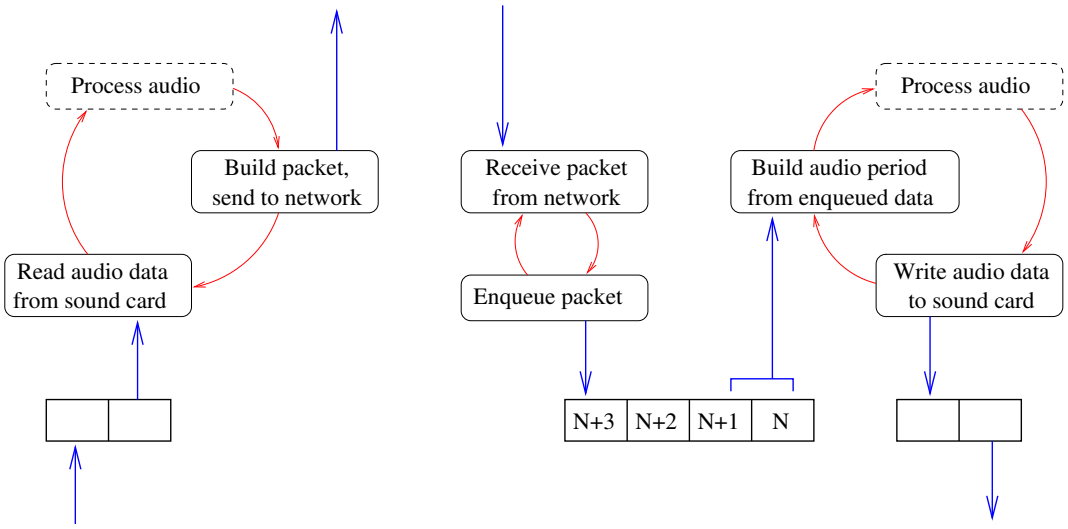

Figure 1: Application architecture and conceptual flowchart. Blue (straight, vertical) lines show the flow of the audio data, red (curved) lines show program flow. From left to right, there is the recorder/sender thread, the receiver thread and the playback thread. Audio is acquired and played back via the audio interfaces, at the bottom of the figure, and sent to, and received from, the network at the top of the figure. The square boxes at the bottom of the figure are, from left to right, the input sound card buffer, the receiver queue and the output sound card buffer. The dashed boxes indicates the possibility for additional processing, currently not present, of the audio data.

sequence number and a time stamp, as shown in figure 2.

Positions in the audio stream are identified by the sequence number of the packet and a frame-level offset into the packet. Of particular interest is the *playback position* kept by the receiver. This is the position of the data that is "next in turn" to be sent to the playback sound card.

3.2 The receiver queue

The receiver queue is the central data structure of the program. In this queue, received packets are temporarily stored until their audio data have been played back. (No copying of data takes place during enqueuing. The queue is implemented as an array of pointers to packets, addressed modulo the length of the array to give a circular buffer.) The queue, containing the packet corresponding to the playback position and any more recent packets, acts as a sliding window onto the incoming packet stream.

A primary purpose of the queue is to buffer the incoming packet stream, absorbing the effects of network transmission time jitter. The setting of the *nominal length* of the queue, i.e. the number of audio periods we try to keep in the queue, allows for the trading of latency for robustness. More data in the queue will give a higher latency, but also more leeway for the occasional late arriving packet to still come in time.

The queue, together with the packet format, implicitly defines a simple protocol that makes the application capable of handling the shortcomings of UDP. Packets are inserted into the queue in correct order, according to their sequence numbers. Lost (or missing) packets are detected, and dummy data (currently silence) substituted in their place. Duplicate packets, and packets arriving too late, are detected and rejected. Altogether, this gives a data stream that is ordered and without gaps.

The queue is further central in the synchronisation of sender and receiver, as discussed below.

3.3 Synchronisation and drift adjustment

There are two kinds of synchronisation issues, synchronisation on a large scale and drift adjustment. Both are handled in the receiver queue.

The first, synchronisation, is handled by the receiver thread. As mentioned above, the receiver queue is a sliding window onto the packet stream. The purpose of the synchronisation is to position this window correctly. This is done by monitoring the sequence numbers of the incoming packets. A packet is said to be "early" if

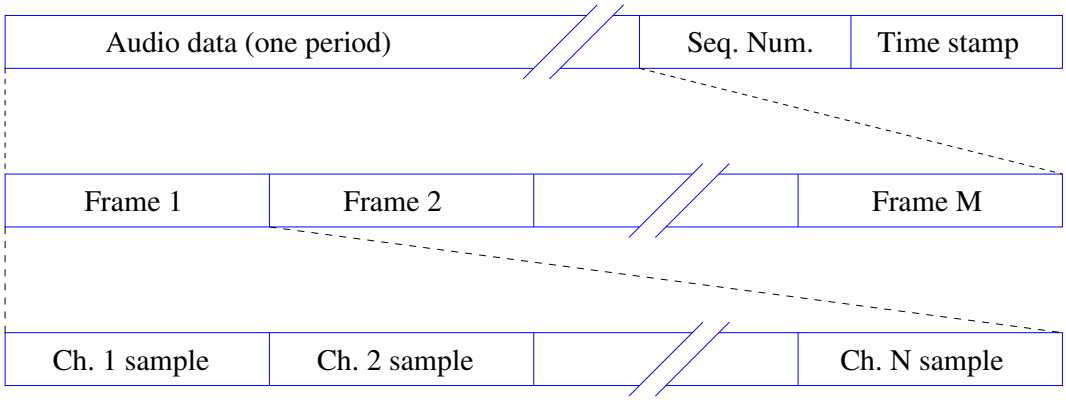

Figure 2: LDAS packet format, the payload of the UDP packet.

its sequence number is so high that it would fall outside of the queue. If an early packet arrives, this is a sign that the receiver is lagging, and the queue is resynchronised. Resynchronisation resets the queue, setting a new playback position, so that the length of the queue, as measured from the playback position to the end of the queue (including the early packet) equals the nominal queue length.

Similarly, "late" packets are packets arriving after they should have been played back, i.e. with a sequence number lower than the sequence number corresponding to the current playback position. From the number of recent late arriving packets, a "lateness factor" is computed. If this value gets too high, the receiver is assumed to be leading, and the receiver queue is resynchronised.

Although nominally equal, the sampling frequencies of the sound cards at the sending and receiving ends will in practice differ slightly. Over time, this difference will lead to more samples being produced than consumed, or vice versa, and this aggregation or starvation of data will cause the sender and the receiver to drift apart. To compensate for this, more smoothly than would be done by large scale synchronisation alone, the playback thread does drift adjustment. From the nominal queue length, an upper and a lower limit is computed. If the low pass filtered actual queue length increases or decreases beyond those limits, single samples are skipped or reused to adjust the rate of data consumption as appropriate to counteract the drift. There is at most one sample of adjustment per period, giving a range of adjustment that is inversely proportional to the period size.

A consequence of this adjustment is that the audio periods sent to the sound card will in general not correspond to those of the received packets, but be built from audio data from more than one packet.

4 Similar solutions

For comparison purposes, this section gives a brief overview of other available open source software similar to LDAS. The information is gleaned from available documentation and various other sources. Except for llcon, the authors of this paper have not done actual experiments with these other solutions.

Netjack (Hohn et al., 2006) is a mechanism for transporting realtime audio over an IP Network using UDP. It is fully integrated with Jack. Full duplex transmission of uncompressed stereo data between two computers is offered. Netjack slaves one jackd to the other, and synchronises the slave jack using the incoming packet data. Which technique is used for the synchronisation is not clear from the available documentation.

An earlier solution is jack.udp (Drape, 2005). This is an udp packet transport that sends jack audio data, using UDP, over local networks between two jack enabled computers. For correct operation, jack.udp needs some form of external synchronisation between the soundcards of the computers.

The StreamBD software (SoundWIRE Group, CCRMA, Stanford, 2002) may, among other things, be used for transmission of audio over a network. It uses ALSA or OSS audio drivers, and provides multiples channels of uncompressed audio. TCP is used for one way transmission, UDP or non-guaranteed variants of TCP for bidirectional transmission. Whether, or how, synchronisation is achieved is

not described in the available documentation.

The llcon (Fischer, 2006) software differs from the rest in that it is aimed at situations with limited bandwidth, like 256kbps DSL lines. It may also connect more than two endpoints. Endpoint clients connect to a central server, which mixes all incoming signals and returns the mix to the clients. At the clients, stereo is input and mixed to one channel before sending the data to the server. The sampling frequency is 24 kHz, and IMA-ADPCM coding, which gives a coding delay of one sample, is used to compress the audio data rate to 96 kbps.

5 Latency measurements

For initial testing and latency measurements of LDAS, two PCs were used. PC-1 was equipped with an Intel P4 CPU, PC-2 had an Intel P3 CPU at 1Ghz. The computers had 512 MB of RAM each, and were equipped with M-Audio Delta44 audio interfaces. Both computers were running Linux set up for real time operation, with 2.6.12 multimedia kernels and associated setup from DeMuDi 1.2.0. (PC-2 was a regular DeMuDi installation, while PC-1 was "sidegraded" from Debian Sarge.)

The computers were connected via a switch of the regular campus LAN. The network round trip time, as reported by "ping" was 0.1 ms. For the measurements, LDAS was version 0.1.1 plus a small fix for a hard coded period size. Audio transmission was two-channel full duplex, using the ldas_mate executable. PC-2 was controlled via SSH logins from PC-1, with the corresponding network traffic between the two computers following the same network route as the LDAS transmissions.

Latency measurements were done on one channel, one way, using an impulse response measurement system, a computer running the WinMLS (Morset, 2004) measurement software. Responses were measured from the analog input of PC-1's sound card to the analog output of PC-2's sound card. Several combinations of period size and nominal receiver queue length were tried. For each combination, six impulse response measurements were taken. From the impulse responses, the total latency through the transmission chain was computed. The results are given in table 1.

While measuring, audio from a CD player was transmitted on the remaining channel from PC-1 to PC-2, and also on both channels in the opposite direction. The quality of the transmission was subjectively evaluated by monitoring the received audio signals (mostly one direction at a time) for audible drop-outs and other forms of distortion or quality reduction. For the transmission from PC-1 to PC-2, combinations A, B and D were robust, with no artifacts noted. For combinations C and E, a few distortions (over the course of a few minutes) were noted. The distortions had the character of brief crackles. Transmission from PC-2 to PC-1 seemed somewhat less robust. Infrequent distortions were noted for combination D, and more frequent distortions for E, F and G. Especially for the lower latency setups, transmission from PC-2 to PC-1 was susceptible to other use of PC-1, with e.g. a "find /usr" with output to a terminal window leading to quite a bit of "stuttering" in the transmitted audio.

	Period size	Queue length	Mean latency	Standard deviation
A	128	3	15.2	0.8
B	128	1	11.0	0.8
C	128	0.1	8.2	0.6
D	64	1	5.8	0.4
E	64	0.1	4.9	0.4
F	32	0.1	3.1	0.2
G	16	1	2.6	0.1

Table 1: Total latency for audio transmission with LDAS. The latency is measured from analog input to analog output for various combinations (marked A to G) of audio period size and nominal receiver queue length. The latency mean value and standard deviation are given in milliseconds. Period size is measured in frames (samples), and the nominal length of the receiver queue in periods. Sampling frequency is 48kHz.

6 Conclusions

A software tool for transmission of high-quality multichannel audio over IP networks has been designed and implemented. As one target application of the tool is networked ensemble playing, emphasis has been placed on achieving low latency. Testing has shown that the tool is capable of achieving robust full duplex transmission while keeping latencies below our stated goal of 20 milliseconds on a local area network. If small transmission artifacts are accepted, even lower latencies may be obtained. While not finished, the software is in a state where it may be

used for experimental work and research into user experienced quality of service in telecommunications.

7 Acknowledgements

Richard Lee Revell, of Mindpipe Audio, has written parts of LDAS, in cooperation with Asbjørn Sæbø. We are grateful for his assistance. The authors also thank Svein Sørsdal and Georg Ottesen of SINTEF, Jon Kåre Hellan of UNINETT and Paul Calamia, currently at Rensselaer Polytechnic Institute, for valuable discussions and advice.

8 Availability

The LDAS software has been released under the GNU General Public License. It may be downloaded from `http://www.q2s.ntnu.no/~asbjs/ldas/ldas.html`.

References

Robin Bargar, Steve Church, Akira Fukuda, James Grunke, Douglas Keislar, Bob Moses, Ben Novak, Bruce Pennycook, Zack Zettel, John Strawn, Phil Wiser, and Wieslaw Woszczyk. 1998. Technology report TC-NAS 98/1networking audio and musing using internet2 and next-generation internet capabilities. Technical report, Technical Council, Audio Engineering Society, Inc.

Chris Chafe and Michael Gurevich. 2004. Network time delay and ensemble accuracy: Effects of latency, assymetry. In *117th Audio Eng. Soc. Convention Preprints*. Audio Eng. Soc., October. Preprint 6208.

Chris Chafe, Scott Wilson, Randal Leistikow, Dave Chisholm, and Gary Scavone. 2000. A simplified approach to high quality music and sound over IP. In *COST-G6 Conference on Digital Audio Effects (DAFx-00)*, December 7 – 9.

Rohan Drape. 2005. jack.udp. `http://www.slavepianos.org/rd/sw/sw-23`.

Volker Fischer. 2006. Low latency (internet) connection. `http://llcon.sourceforge.net/`.

Torben Hohn, Dan Mills, and Robert Jonsson. 2006. Netjack v. 0.7. `http://netjack.sourceforge.net/`.

Nelson Posse Lago and Fabio Kon. 2004. The quest for low latency. In *International Computer Music Conference*, Miami, November.

Lars Henrik Morset. 2004. WinMLS 2004. `http://www.winmls.com`.

SoundWIRE Group, CCRMA, Stanford. 2002. StreamBD 1.0b. `http://ccrma.stanford.edu/groups/soundwire/software/newstream/docs/`.

W. Richard Stevens, Bill Fenner, and Andrew M. Rudoff. 2004. *UNIX Network Programming*, volume 1. Addison-Wesley, third edition.

Ola Strand. 2002. Distribuert multimedia samhandling. Master's thesis, Norwegian University of Science and Technology, NTNU, Dept. of Telecomm., Acoustics Group. (In norwegian).

Håkon Liestøl Winge. 2003. Musikksamspill over IP. Term project, NTNU, Norwegian University of Science and Technology, September. (In norwegian).

Wieslaw Woszczyk, Jeremy R. Cooperstock, John Roston, and William Martens. 2005. Shake, rattle and roll: Getting immersed in multisensory interactive music via broadband networks. *J. Audio Eng. Soc.*, 53(4):336 – 344, April.

swonder3Dq: Auralisation of 3D objects with Wave Field Synthesis

Marije BAALMAN
Institute for Communication Sciences, Technische Universität Berlin
Einsteinufer 17
Berlin, Germany
baalman@kgw.tu-berlin.de

Abstract

swonder3Dq is a software tool to auralise three dimensional objects with Wave Field Synthesis. It presents a new approach to model the radiation characteristics of sounding objects.

Keywords

Wave field synthesis, spatialisation, radiation characteristic, physical modelling

1 Introduction

Wave Field Synthesis (WFS) is an interesting method for spatialisation of electronic music. Its main advantage is that it has no sweet spot, but instead a large listening area, making the technology attractive for concert situations.

Yet, whenever acoustic instruments are combined with electroacoustic sounds in concerts, a common problem is that the electroacoustic part tends to lack depth and extension in comparison with the sound from the acoustic instruments. Misdariis (1) and Warusfel (2) have proposed special 3D loudspeaker arrays to simulate different radiation characteristics. A problem with this technique is that the object itself is static; a movement of the source can only be created by physically moving the loudspeaker array.

In current Wave Field Synthesis applications there are only solutions for monopole point sources and plane waves. There are some reports of ad hoc solutions to simulate larger sources, such as the Virtual Panning Spots (3) and the auralisation of a grand piano (4) by using a few point sources. Currently, work is done on implementing radiation characteristics for point sources (5). By definition radiation characteristics are only applicable at a certain distance from the object modelled. Since WFS sources can get very close, it makes sense to look for a general solution for auralising arbitrarily shaped sources.

2 Theory

2.1 Source model

The wave field of a sounding object can be approximated by superposition of the wave fields of a number of point sources. The locations of these point sources and the number can be chosen arbitrarily, but it make sense to choose the locations on the surface of the object. This separates the calculation of the vibration of the object itself (which can be done with various methods, such as finite element methods or modal synthesis) from the calculation of the radiation of the sound from the object into the air.

For a correct calculation of the radiated sound field, the vibration of the surface must be spatially sampled. Here there is the danger of undersampling (6); this danger is twofold: first, spatial amplitude variations of the surface vibration may be lost, and secondly, depending on the frequency content of the sound, spatial aliasing may occur in a similar way as for the WFS speaker array itself.

In practice the sound emitted from different points on the object's surface will be correlated. For a practical implementation it is useful to assume that for a point Ψ on the surface:

$$S_\Psi(\vec{r}_\Psi, \phi, \theta, \omega) = S(\omega)G(\vec{r}_\Psi, \phi, \theta, \omega) \quad (1)$$

i.e. from each point source that is part of the distribution a source signal $S(\omega)$ filtered with G is emitted (G depending on the location $\vec{r}_\Psi$ of the point source Ψ, the angular frequency ω of the sound and the direction angles ϕ and θ). Applied to the reproduction of electronic music, this will allow a composer to determine the filter characteristics of his source object and the sound signal emitted by the source independently. Thus, the resulting filtering function for each speaker can be determined in advance and the signal input can be convolved in realtime with this filter.

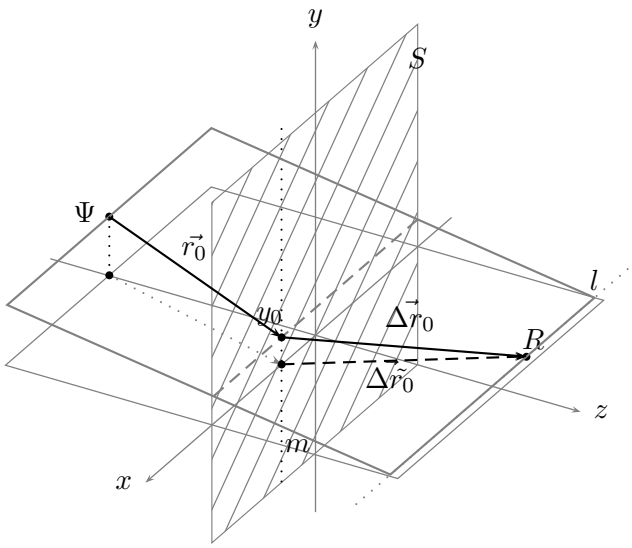

Figure 1: The stationary point $(x_m, y_0, 0)$ lies on the cross-section of the line m (in plane S) and the plane through Ψ and the reference line l.

2.2 Adaptation of the WFS-operator

For WFS reproduction of a 3-dimensional source object the commonly used $2\frac{1}{2}D$-operator (7) is not sufficient as its derivation only sources in the same plane as the array and reference line are taken into account. In (8) the WFS operator for points outside of the horizontal plane was derived, starting from the Rayleigh integrals. The main difference from the $2\frac{1}{2}D$-operator is the calculation of the stationary point:

$$y_0 = y_R + (y_\Psi - y_R)\frac{z_R}{z_\Psi + z_R} \qquad (2)$$

where z_R is the z-coordinate of the receiver and z_Ψ of the source point (see also figure 1). The driver function of a speaker for the contribution of one monopole source point becomes:

$$Q(x,\omega) = S(\omega)\sqrt{\frac{jk}{2\pi}}\sqrt{\frac{\Delta r_0}{\Delta r_0 + r_0}}cos(\phi_0)\frac{e^{-jkr_0}}{\sqrt{r_0}} \qquad (3)$$

the speaker being assumed to be on the x-axis.

It should be noted that the actual elevation will not be heard, when the elevated source is played back by the WFS-array. The elevated points are mainly of interest, because their contributions will interfere with those of the points in the horizontal plane.

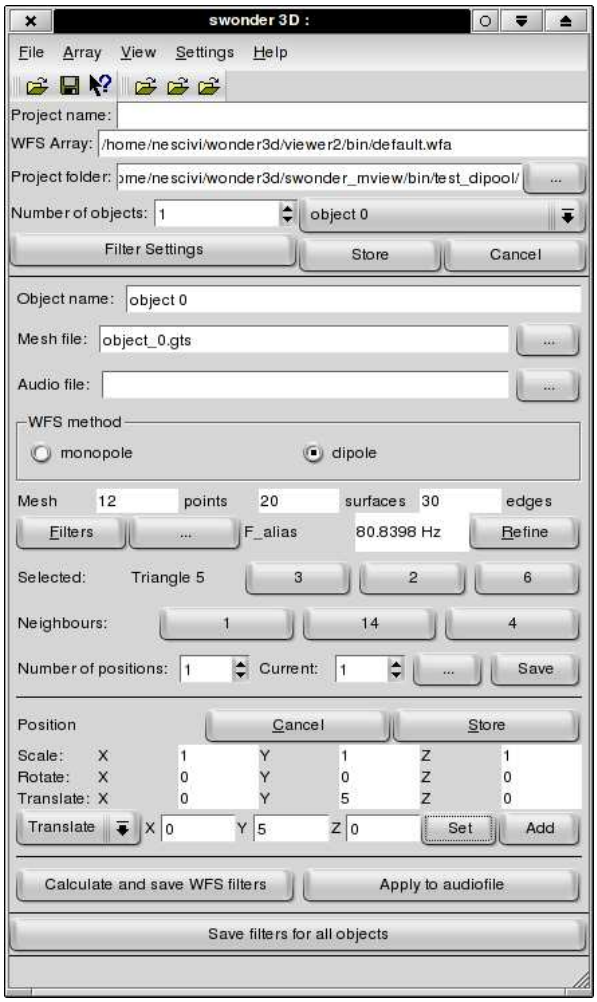

Figure 2: Snapshot of the graphical user interface of *swonder3Dq*

3 Implementation

The software enables to calculate the filters for WFS reproduction for several sound objects. The objects themselves are defined by their geometrical data and the radiation filters at several points on the surface. Objects can be positioned and given different orientations in space. Figure 2 is a snapshot of the graphical user interface. There is an OpenGL viewer included to look at the object. The user can choose between defining filters of each node, or choose a multichannel audiofile which has the sound for each node (e.g. calculated with *Modalys* (9)). The program then calculates the WFS operators or the loudspeaker signals and saves them to disk.

A second part of the software enables the user to load a project and listen to the desired object on the desired location with the desired orientation. This part of the software can be controlled with OSC. *BruteFIR* (10) is used as the convo-

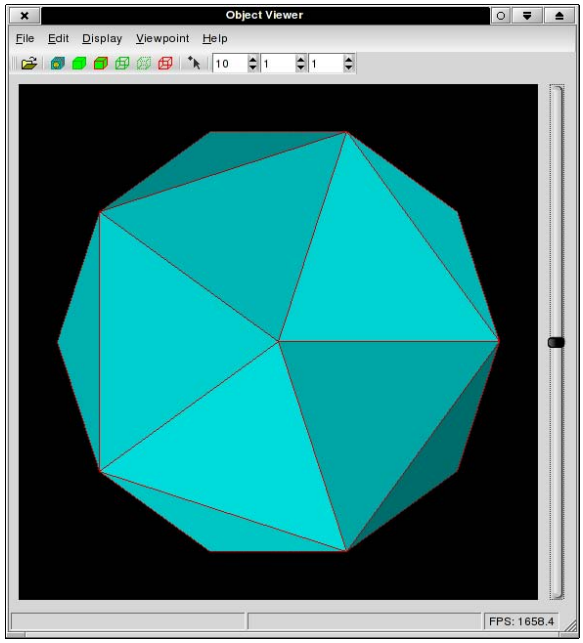

Figure 3: Graphical display of an object with *mview*

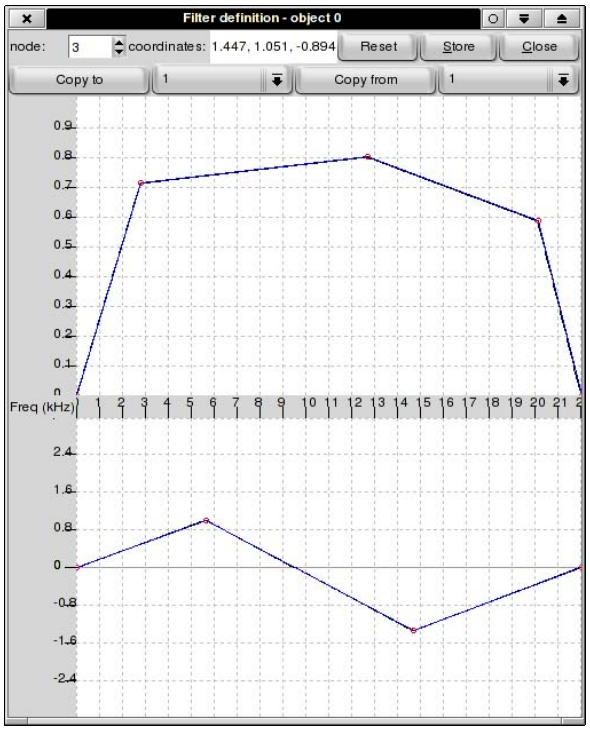

Figure 4: Snapshot of the dialog to define a filter for a point on the source object surface

lution engine.

3.1 Geometry

There are a multitude of programs and libraries available for manipulating and visualising 3D data. The *GTS-library* (11) encompasses many functions to read, write and work with meshes. A disadvantage of this library is that the points on the mesh (the vertices) are not ordered or tagged while they are loaded, so it is not possible to connect the data for the radiation filters to them.

In the program *mview* (12) a lot of methods for working with meshes were already implemented and with only a few additions to implement the filter definition per source point, it was included into *swonder3Dq* for the graphical display of the objects.

GeomView (13) was used as a second viewer to view the whole scene: the WFS speaker array as well as several sounding objects. *GeomView* is used as an external program, interfaced via stdin and stdout.

3.2 Filter definition and calculation

There is a simple graphical representation of the frequency response of the filter, where the user can define breakpoints (figure 4). The filter settings can be copied between source points and there is an interaction between the picked triangle and its corner points (which can be selected in the main gui) and the current source point for which a filter is defined. It is also possible to load filters from file.

The filter is then calculated based on the defined breakpoints with a method as described in (14)[1]. For the fast fourier transform the *FFTW Library* is used (15).

3.3 Refinement of the surface

To make the discretisation distance smaller, an algorithm is needed to calculate more points on the object surface. A simple method (partly taken from the *GTS Library*) is the midvertex insertion method. On each edge of a triangle, a point is added in the middle to divide the edge in two. Then, every midpoint is connected to those of the other edges (figure 5). This method can be applied more than once, to create a fine raster of points. The aliasing frequency (6) can be calculated by finding the longest edge in the mesh and calculate the corresponding aliasing frequency.

Secondly the filter for the newly calculated points needs to be determined. This is done by an average of the filter of the neighbouring points, using an inverse distance weighting method (16) to determine the contribution of each point:

[1] Chapter 17, pages 297-300

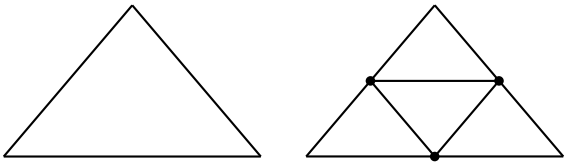

Figure 5: Refinement of a triangle with the *mid-vertex insertion* method

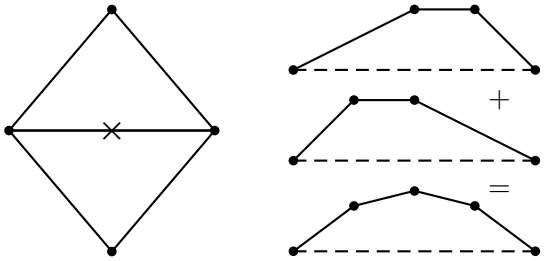

Figure 6: Averaging of the filter values. On the left is shown from which point the average is taken: $\times$ is the new point, $\bullet$ are the neighbourpoints. On the left is shown how the breakpoints are added when averaging.

$$Z_j = \frac{\sum_{i=1}^{n} \frac{Z_i}{h_{ij}^{\beta}}}{\sum_{i=1}^{n} \frac{1}{h_{ij}^{\beta}}} \tag{4}$$

Z_j is the value of the new point j, Z_i of the neighbour point i, h_{ij} the distance from point i to j, and β a factor that defines the weighting of the distance, usually set to $\beta = 2$. In this case the factor β can be interpreted as a kind of measure how well the sound is propagated through the material of the object.

The averaging between two filters is calculated as follows (see also figure 6): for each breakpoint from either filter the corresponding value on that frequency value is calculated for the other filter. The new filter then has a breakpoint value at that frequency value, which is an average of the two breakpoints of the two filters. The average is taken from the real and imaginary parts of the coefficients.

3.4 3D WFS Calculation

Steps in the calculation of the WFS filters (for each object, at each location specified):

1. Per source point:
   - Check: is the source point audible (visible) for the loudspeaker?
   - Calculation of delay and volume factor according to the WFS operator
   - Convolution of the WFS-delay and volume with the filter for that source point
2. Addition of all filters of the source for each loudspeaker
3. Save filter to disk

As the software is still in an experimental state, there is a choice between defining the source points as monopoles or as dipoles. In the case of a dipole, the main axis of radiation is in the direction of the normal (pointing outwards) on the surface; in the case of points on the corners of triangles which are not on the same plane, this normal is an average of the normals of the triangles it is a corner point of.

When a source point is at the back side of the object, a direct path of the sound to the loudspeaker is not possible and this source point should not be taken into account. Diffraction of sound waves around the object is neglected in this case.

4 First tests

Preliminary listening tests on a frontal WFS-speaker array of 24 loudspeakers (at 12.5cm distance) show that this approach does give a stronger spatial impression of a sound source.

However, it became apparent that the neglection of diffraction cannot be allowed. As for different speakers, different points of the object will be at the backside (figure 7), sound will arrive from speakers to parts of the listening area that should be obscured when neglecting diffraction. Thus some sense of diffraction is created, but at a much later stage than should

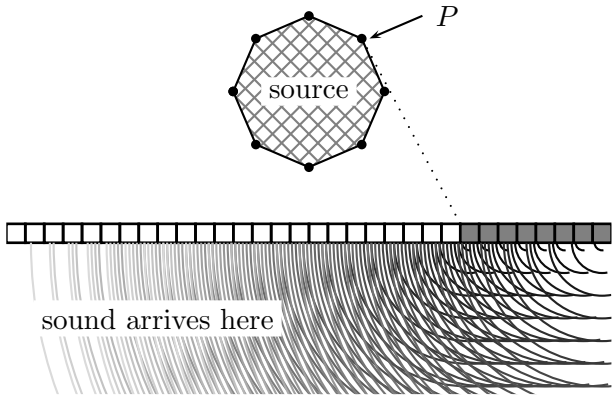

Figure 7: Illustration of the audibility problem. The point P is only audible for the gray speakers on the right, yet the sound from these speakers will arrive on the left side of the listening area.

be. We will need to find a different approach in order to take the diffraction of the sound around the object into account properly.

5 Conclusions and future work

An extension of the WONDER software (17) was presented which enables the calculation of WFS auralisation for complex sound sources. While its first aim is to be used for WFS auralisation, several concepts introduced could be used in other 3D audio applications, such as binaural representation.

Further research will be done on how to take the diffraction of waves around objects into account, before implementing that feature. Future work will include listening experiments, as well as doing usability tests by working with composers using the software.

6 Acknowledgements

The work on *swonder3Dq* is part of a Ph.D. research project funded by the *NaFöG* in Berlin, Germany.

7 References

[1] N. Misdariis, O. Warusfel & R. Caussé. Radiation control on a multi-loudspeaker device. In *ISMA 2001*, 2001.

[2] N. Misdariis & T. Caulkins O. Warusfel, E. Corteel. Reproduction of sound source directivity for future audio applications. In *ICA 2004*, 2004.

[3] G. Theile, H. Wittek & M. Reisinger. Wellenfeld-synthese verfahren: Ein weg für neue möglichkeiten der räumlichen tongestaltung. In *22nd Tonmeistertagung, Hannover, Germany, 2002 November*, 2002.

[4] I. Bork, & M. Kern. Simulation der schallabstrahlung eines flügels. In *DAGA '03*, 2003.

[5] O. Warusfel & N. Misdariis. Sound source radiation synthesis: from stage performance to domestic rendering. In *AES 116th Convention, Berlin, Germany, 2004, May, Preprint 6018*, 2004.

[6] M.A.J. Baalman. Discretisation of complex sound sources for reproduction with wave field synthesis. In *DAGA '05, 14 - 17 March 2005, München*, 2005.

[7] E.N.G. Verheijen. *Sound Reproduction by Wave Field Synthesis*. PhD thesis, TU Delft, The Netherlands, 1998.

[8] M.A.J. Baalman. Elevation problems in the auralisation of sound sources with arbitrary shape with wave field synthesis. In *ICMC 2005, 1-6 September 2005, Barcelona, Spain*, 2005.

[9] IRCAM. Modalys. http://www.ircam.fr/, 1991-2005.

[10] A. Torger. Brutefir. http://www.ludd.luth.se/∼torger/brutefir.html, 2001-2005.

[11] Gnu triangulated surface library. http://gts.sourceforge.net/, 2000-5.

[12] Helmut Cantzler. Mesh viewer. http://mview.sourceforge.net/, 2001-5.

[13] Geometry Technologies. Geomview. http://www.geomview.org/, 1992-2005.

[14] Steven W. Smith. *The Scientist and Engineer's Guide to Digital Signal Processing*. California Technical Publishing, 1997.

[15] Matteo Frigo and Steven G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005. special issue on "Program Generation, Optimization, and Platform Adaptation".

[16] M.L. Green. *GEOGRAPHIC INFORMATION SYSTEM BASED MODELING OF SEMI-VOLATILE ORGANIC COMPOUNDS TEMPORAL AND SPATIAL VARIABILITY*. PhD thesis, University of New York at Buffalo, 2000.

[17] M.A.J. Baalman. Updates of the wonder software interface for using wave field synthesis. In *3rd International Linux Audio Conference, April 21-24, 2005, ZKM, Karlsruhe*, 2005.

DSP Programming with Faust, Q and SuperCollider

Yann ORLAREY
Grame, Centre National de Creation Musicale
Lyon, France
orlarey@grame.fr

Albert GRÄF
Dept. of Music Informatics
Johannes Gutenberg University
Mainz, Germany
Dr.Graef@t-online.de

Stefan KERSTEN
Dept. of Communication Science
Technical University
Berlin, Germany
stefan.kersten@tu-berlin.de

Abstract

Faust is a functional programming language for real-time signal processing and synthesis that targets high-performance signal processing applications and audio plugins. The paper gives a brief introduction to Faust and discusses its interfaces to Q, a general-purpose functional programming language, and SuperCollider, an object-oriented sound synthesis language and engine.

Keywords

Computer music, digital signal processing, Faust programming language, functional programming, Q programming language, SuperCollider

1 Introduction

Faust is a programming language for real-time signal processing and synthesis that targets high-performance signal processing applications and audio plugins. This paper gives a brief introduction to Faust, emphasizing practical examples rather than theoretic concepts which can be found elsewhere (Orlarey et al., 2004).

A Faust program describes a *signal processor*, a DSP algorithm that transforms input signals into output signals. Faust is a *functional* programming language which models signals as functions (of time) and DSP algorithms as higher-order functions operating on signals. Faust programs are compiled to efficient C++ code which can be included in C/C++ applications, and which can also be executed either as standalone programs or as plugins in other environments. In particular, in this paper we describe Faust's interfaces to Q, an interpreted, general-purpose functional programming language based on term rewriting (Gräf, 2005), and SuperCollider (McCartney, 2002), the well-known object-oriented sound synthesis language and engine.

2 Faust

The programming model of Faust combines a functional programming approach with a block-diagram syntax. The functional programming approach provides a natural framework for signal processing. Digital signals are modeled as discrete functions of time, and signal processors as second order functions that operate on them. Moreover Faust block-diagram *composition operators*, used to combine signal processors together, fit in the same picture as third order functions.

Faust is a compiled language. The compiler translates Faust programs into equivalent C++ programs. It uses several optimization techniques in order to generate the most efficient code. The resulting code can usually compete with, and sometimes outperform, DSP code directly written in C. It is also self-contained and doesn't depend on any DSP runtime library.

Thanks to specific *architecture files*, a single Faust program can be used to produce code for a variety of platforms and plugin formats. These architecture files act as wrappers and describe the interactions with the host audio and GUI system. Currently more than 8 architectures are supported (see Table 1) and new ones can be easily added.

alsa-gtk.cpp	ALSA application
jack-gtk.cpp	JACK application
sndfile.cpp	command line application
ladspa.cpp	LADSPA plugin
max-msp.cpp	Max MSP plugin
supercollider.cpp	Supercollider plugin
vst.cpp	VST plugin
q.cpp	Q language plugin

Table 1: The main architecture files available for Faust

In the following subsections we give a short and informal introduction to the language through two simple examples. Interested readers can refer to (Orlarey et al., 2004) for a more complete description.

2.1 A simple noise generator

A Faust program describes a signal processor by combining primitive operations on signals (like $+, -, *, /, \sqrt{}, \sin, \cos, \ldots$) using an algebra of high level *composition operators* (see Table 2). You can think of these composition operators as a generalization of mathematical function composition $f \circ g$.

$f \sim g$	recursive composition
f `,` g	parallel composition
f `:` g	sequential composition
f `<:` g	split composition
f `:>` g	merge composition

Table 2: The five high level block-diagram *composition operators* used in Faust

A Faust program is organized as a set of *definitions* with at least one for the keyword `process` (the equivalent of `main` in C).

Our noise generator example `noise.dsp` only involves three very simple definitions. But it also shows some specific aspects of the language:

```
random  = +(12345) ~ *(1103515245);
noise   = random/2147483647.0;
process = noise * checkbox("generate");
```

The first definition describes a (pseudo) random number generator. Each new random number is computed by multiplying the previous one by 1103515245 and adding to the result 12345.

The expression `+(12345)` denotes the operation of adding 12345 to a signal. It is an example of a common technique in functional programming called *partial application*: the binary operation `+` is here provided with only one of its arguments. In the same way `*(1103515245)` denotes the multiplication of a signal by 1103515245.

The two resulting operations are *recursively composed* using the $\sim$ operator. This operator connects in a feedback loop the output of `+(12345)` to the input of `*(1103515245)` (with an implicit 1-sample delay) and the output of `*(1103515245)` to the input of `+(12345)`.

The second definition transforms the random signal into a noise signal by scaling it between -1.0 and +1.0.

Finally, the definition of process adds a simple user interface to control the production of the sound. The noise signal is multiplied by a GUI checkbox signal of value 1.0 when it is checked and 0.0 otherwise.

2.2 Invoking the compiler

The role of the compiler is to translate Faust programs into equivalent C++ programs. The key idea to generate efficient code is not to compile the block diagram itself, but *what it computes*.

Driven by the semantic rules of the language the compiler starts by propagating symbolic signals into the block diagram, in order to discover how each output signal can be expressed as a function of the input signals.

These resulting signal expressions are then simplified and normalized, and common subexpressions are factorized. Finally these expressions are translated into a self contained C++ class that implements all the required computation.

To compile our noise generator example we use the following command :

```
$ faust noise.dsp
```

This command generates the C++ code in Figure 1. The generated class contains five methods. `getNumInputs()` and `getNumOutputs()` return the number of input and output signals required by our signal processor. `init()` initializes the internal state of the signal processor. `buildUserInterface()` can be seen as a list of high level commands, independent of any toolkit, to build the user interface. The method `compute()` does the actual signal processing. It takes 3 arguments: the number of frames to compute, the addresses of the input buffers and the addresses of the output buffers, and computes the output samples according to the input samples.

The `faust` command accepts several options to control the generated code. Two of them are widely used. The option `-o` *outputfile* specifies the output file to be used instead of the standard output. The option `-a` *architecturefile* defines the architecture file used to wrap the generate C++ class.

For example the command `faust -a q.cpp -o noise.cpp noise.dsp` generates an external object for the Q language, while `faust -a jack-gtk.cpp -o noise.cpp noise.dsp` generates a standalone Jack application using the GTK toolkit.

Another interesting option is `-svg` that generates one or more SVG graphic files that represent the block-diagram of the program as in Figure 2.

```
class mydsp : public dsp
{
 private:

  int     R0_0;
  float   fcheckbox0;

 public:

  virtual int getNumInputs()  {
    return 0;
  }
  virtual int getNumOutputs() {
    return 1;
  }
  virtual void init(int samplingFreq) {
    fSamplingFreq = samplingFreq;
    R0_0 = 0;
    fcheckbox0 = 0.0;
  }
  virtual void buildUserInterface(UI* ui) {
    ui->openVerticalBox("faust");
    ui->addCheckButton("generate",
                               &fcheckbox0);
    ui->closeBox();
  }
  virtual void compute (int count,
           float** input, float** output) {
    float* output0; output0 = output[0];
    float ftemp0 = 4.656613e-10f*fcheckbox0;
    for (int i=0; i<count; i++) {
      R0_0 = (12345 + (1103515245 * R0_0));
      output0[i] = (ftemp0 * R0_0);
    }
  }
};
```

Figure 1: The C++ implementation code of the noise generator produced by the Faust compiler

2.3 The Karplus-Strong algorithm

Karplus-Strong is a well known algorithm first presented by Karplus and Strong in 1983 (Karplus and Strong, 1983). Whereas not completely trivial, the principle of the algorithm is simple enough to be described in few lines of Faust, while producing interesting metallic plucked-string and drum sounds.

The sound is produced by an impulse of noise that goes into a resonator based on a delay line with a filtered feedback. The user interface contains a button to trigger the sound production, as well as two sliders to control the size of both the resonator and the noise impulse, and the amount of feedback.

2.3.1 The noise generator

We simply reuse here the noise generator of the previous example (subsection 2.1).

```
random  = +(12345) ~ *(1103515245);
```

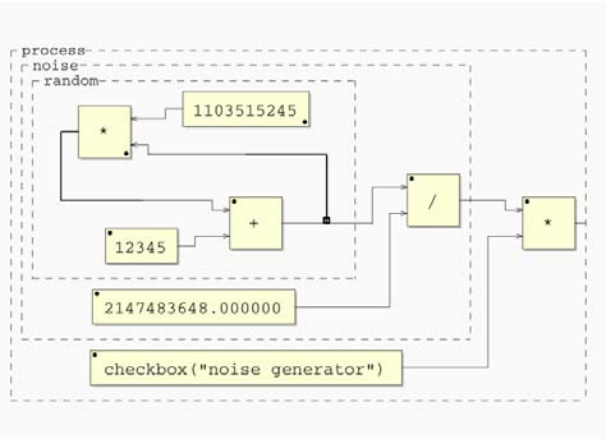

Figure 2: Graphic block-diagram of the noise generator produced with the -svg option

```
noise   = random/2147483647.0;
```

2.3.2 The trigger

The `trigger` is used to transform the signal delivered by a user interface button into a precisely calibrated control signal. We want this control signal to be 1.0 for a duration of exactly n samples, independentely of how long the button is pressed.

```
impulse(x) = x - mem(x) : >(0.0);
decay(n,x) = x - (x>0.0)/n;
release(n) = + ~ decay(n);
trigger(n) = button("play") : impulse
           : release(n) : >(0.0);
```

For that purpose we first transforms the button signal into a 1-sample impulse corresponding to the raising front of the button signal. Then we add to this impulse a kind of *release* that will decrease from 1.0 to 0.0 in exactly n samples. Finally we produce a control signal which is 1.0 when the signal with release is greater than 0.0.

All these steps are combined in a four stages sequential composition with the operator ':'.

2.3.3 The resonator

The resonator uses a variable delay line implemented using a table of samples. Two consecutive samples of the delay line are averaged, attenuated and fed back into the table.

```
index(n)    =  &(n-1) ~ +(1);
delay(n,d,x)= rwtable( n, 0.0, index(n),
            x, (index(n)-int(d))&(n-1) );
average(x)  = (x+mem(x))/2;
resonator(d,a) = (+ : delay(4096, d-1))
               ~ (average : *(1.0-a));
```

2.3.4 Putting it all together

The last step is to put all the pieces together in a sequential composition. The parameters of the trigger and the resonator are controlled by two user interface sliders.

```
dur = hslider("duration",128,2,512,1);
att = hslider("attenuation",
                          0.1,0,1,0.01);
process = noise
        : *(trigger(dur))
        : resonator(dur,att);
```

A screen shot of the resulting application (compiled with the `jack-gtk.cpp` architecture) is reproduced in Figure 3. It is interesting to note that despite the fact that the duration slider is used twice, it only appears once in the user interface. The reason is that Faust enforces *referential transparency* for all expressions, in particular user interface elements. Things are uniquely and unequivocally identified by their definition and naming is just a convenient shortcut. For example in the following program, `process` always generate a null signal:

```
foo = hslider("duration", 128, 2, 512, 1);
faa = hslider("duration", 128, 2, 512, 1);
process = foo - faa;
```

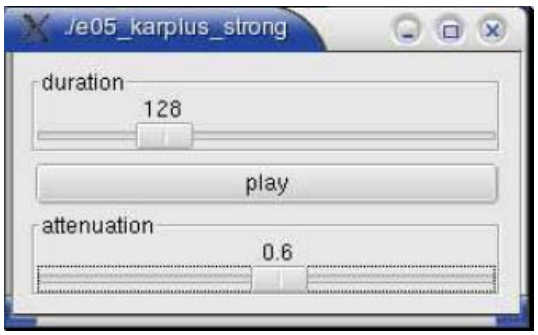

Figure 3: Screenshot of the Karplus-Strong example generated with the `jack-gtk.cpp` architecture

3 Faust and Q

Faust is tailored to DSP programming, and as such it is not a general-purpose programming language. In particular, it does not by itself have any facilities for other tasks typically encountered in signal processing and synthesis programs, such as accessing the operating system environment, real-time processing of audio and MIDI data, or presenting a user interface for the application. Thus, as we already discussed in the preceding section, all Faust-generated DSP programs need a supporting infrastructure (embodied in the architecture file) which provides those bits and pieces.

One of the architectures included in the Faust distribution is the Q language interface. Q is an interpreted functional programming language which has the necessary facilities for doing general-purpose programming as well as soft real-time processing of MIDI, OSC a.k.a. Open Sound Control (Wright et al., 2003) and audio data. The Q-Faust interface allows Faust DSPs to be loaded from a Q script at runtime. From the perspective of the Faust DSP, Q acts as a programmable supporting environment in which it operates, whereas in Q land, the DSP module is used as a "blackbox" to which the script feeds chunks of audio and control data, and from which it reads the resulting audio output. By these means, Q and Faust programs can be combined in a very flexible manner to implement full-featured software synthesizers and other DSP applications.

In this section we give a brief overview of the Q-Faust interface, including a simple but complete monophonic synthesizer example. For lack of space, we cannot give an introduction to the Q language here, so instead we refer the reader to (Gräf, 2005) and the extensive documentation available on the Q website at `http://q-lang.sf.net`.

3.1 Q module architecture

Faust's side of the Q-Faust interface consists of the *Q architecture file*, a little C++ code template `q.cpp` which is used with the Faust compiler to turn Faust DSPs into shared modules which can be loaded by the Q-Faust module at runtime. This file should already be included in all recent Faust releases, otherwise you can also find a copy of the file in the Q-Faust distribution tarball.

Once the necessary software has been installed, you should be able to compile a Faust DSP to a shared module loadable by Q-Faust as follows:

```
$ faust -a q.cpp -o mydsp.cpp mydsp.dsp
$ g++ -shared -o mydsp.so mydsp.cpp
```

Note: If you want to load several different DSPs in the same Q script, you have to make sure that they all use distinct names for the `mydsp` class. With the current Faust version this can be achieved most easily by just redefining

mydsp, to whatever class name you choose, during the C++ compile stage, like so:

```
$ g++ -shared -Dmydsp=myclassname
   -o mydsp.so mydsp.cpp
```

3.2 The Q-Faust module

The compiled DSP is now ready to be used in the Q interpreter. A minimal Q script which just loads the DSP and assigns it to a global variable looks as follows:

```
import faust;
def DSP = faust_init "mydsp" 48000;
```

The first line of the script imports Q's faust module which provides the operations to instantiate and operate Faust DSPs. The faust_init function loads a shared module (mydsp.so in this example, the .so suffix is supplied automatically) and returns an object of Q type FaustDSP which can then be used in subsequent operations. The second parameter of faust_init, 48000 in this example, denotes the sample rate in Hz. This can be an arbitrary integer value which is available to the hosted DSP (it is up to the DSP whether it actually uses this value in some way).

In the following examples we assume that you have actually loaded the above script in the Q interpreter; the commands below can then be tried at the interpreter's command prompt.

The faust_info function can be used to determine the number of input/output channels as well as the "UI" (a data structure describing the available control variables) of the loaded DSP:

```
==> def (N,M,UI) = faust_info DSP
```

To actually run the DSP, you'll need some audio data, encoded using 32 bit (i.e., single precision) floating point values as a byte string. (A *byte string* is a special kind of data object which is used in Q to represent arbitrary binary data, such as a C vector with audio samples in this case.) Suppose you already have two channels of audio data in the IN1 and IN2 variables and the DSP has 2 input channels, then you would pass the data through the DSP as follows:

```
==> faust_compute DSP [IN1,IN2]
```

This will return another list of byte strings, containing the 32 bit float samples produced by the DSP on its output channels, being fed with the given input data.

Some DSPs (e.g., synthesizers) don't actually take any audio input, in this case you just specify the number of samples to be generated instead:

```
==> faust_compute DSP 1024
```

Most DSPs also take additional control input. The control variables are listed in the UI component of the faust_info return value. For instance, suppose that there is a "Gain" parameter listed there, it might look as follows:

```
==> controls UI!0
hslider <<Ref>> ("Gain",1.0,0.0,10.0,0.1)
```

The second parameter of the hslider constructor indicates the arguments the control was created with in the .dsp source file (see the Faust documentation for more details on this). The first parameter is a Q *reference* object which points to the current value of the control variable. The reference can be extracted from the control description with the control_ref function and you can then change the value with Q's put function before invoking faust_compute (changes of control variables only take effect between different invokations of faust_compute):

```
==> def GAIN = control_ref (controls UI!0)

==> put GAIN 2.0
```

3.3 Monophonic synthesizer example

For a very simple, but quite typical and fully functional example, let us take a look at the monophonic synthesizer program in Figure 4. It basically consists of two real-time threads: a control loop which takes MIDI input and changes the synth DSP's control variables accordingly, and an audio loop which just pulls audio data from the DSP at regular intervals and outputs it to the audio interface. The Faust DSP we use here is the simple additive synth shown in Figure 5.

The header section of the Q script imports the necessary Q modules and defines some global variables which are used to access the MIDI input and audio output devices as well as the Faust DSP. It also extracts the control variables from the Faust DSP and stores them in a dictionary, so that we can finally assign the references to a corresponding collection of global variables. These variables are then used in the control loop to set the values of the control variables.

```
import audio, faust, midi;

def (_,_,_,_,SR) = audio_devices!AUDIO_OUT,
  SR = round SR, BUFSZ = 256,
  IN = midi_open "Synth",
  _ = midi_connect (midi_client_ref
                    "MidiShare/ALSA Bridge") IN,
  OUT = open_audio_stream AUDIO_OUT PA_WRITE
        (SR,1,PA_FLOAT32,BUFSZ),
  SYNTH = faust_init "synth" SR,
  (N,M,UI) = faust_info SYNTH, CTLS = controls UI,
  CTLD = dict (zip (map control_label CTLS)
                   (map control_ref CTLS));

def [FREQ,GAIN,GATE] =
  map (CTLD!) ["freq","gain","gate"];

/*******************************************/

freq N     = 440*2^((N-69)/12);
gain V     = V/127;

process (_,_,_,note_on _ N V)
          = put FREQ (freq N) ||
            put GAIN (gain V) ||
            put GATE 1 if V>0;
          = put GATE 0 if freq N = get FREQ;

midi_loop  = process (midi_get IN) || midi_loop;

audio_loop = write_audio_stream OUT
             (faust_compute SYNTH BUFSZ!0) ||
             audio_loop;

/*******************************************/

def POL = SCHED_RR, PRIO = 10;
realtime   = setsched this_thread POL PRIO;

synth      = writes "Hit <CR> to stop: " ||
             reads || ()
  where H1 = thread (realtime || midi_loop),
    H2 = thread (realtime || audio_loop);
```

Figure 4: Q script for the monophonic synth example

The second section of the code contains the definitions of the control and audio loop functions. It starts out with two helper functions `freq` and `gain` which are used to map MIDI note numbers and velocities to the corresponding frequency and gain values. The `process` function (not to be confused with the `process` "main" function of the Faust program!) does the grunt work of translating an incoming MIDI event to the corresponding control settings. In this simple example it does nothing more than responding to note on and off messages (as usual, a note off is just a note on with velocity 0). The example also illustrates how MIDI

```
import("music.lib");

// control variables

vol  = nentry("vol", 0.3, 0, 10, 0.01);

attk = nentry("attack", 0.01, 0, 1, 0.001);
decy = nentry("decay", 0.3, 0, 1, 0.001);
sust = nentry("sustain", 0.5, 0, 1, 0.01);
rels = nentry("release", 0.2, 0, 1, 0.001);

freq = nentry("freq", 440, 20, 20000, 1);
gain = nentry("gain", 1, 0, 10, 0.01);
gate = button("gate");

// simple monophonic synth

smooth(c) = *(1-c) : +~*(c);

voice   = gate : adsr(attk, decy, sust, rels) :
          *(osci(freq)+0.5*osci(2*freq)+
            0.25*osci(3*freq)) :
          *(gain : smooth(0.999));

process = vgroup("synth", voice : *(vol));
```

Figure 5: Faust source for the monophonic synth example

messages are represented as an "algebraic" data type in Q, and how the note and velocity information is extracted from this data using "pattern matching." In the case of a note on message we change the `FREQ` and `GAIN` of the single synth voice accordingly and then set the `GATE` variable to 1, to indicate that a note is playing. For a note off message, we simply reset the `GATE` variable to 0; in the DSP, this triggers the release phase of the synth's ADSR envelop.

The `process` function is invoked repeatedly during execution of `midi_loop`. The `audio_loop` function just keeps reading the audio output of the DSP and sends it to the audio output stream. The two loops are to be executed asynchronously, in parallel. (It is worth noting here that the necessary protection of shared data, i.e., the control variable references, is done automatically behind the scenes.)

The third section of the script contains the main entry point, the `synth` function which kicks off two real-time threads running the `midi_loop` and `audio_loop` functions and then waits for user input. The function returns a "void" `()` value as soon as the user hits the carriage return key. (At this point the two thread handles `H1` and `H2` are garbage-collected immediately and the corresponding threads are thus terminated automatically, so there is no need to

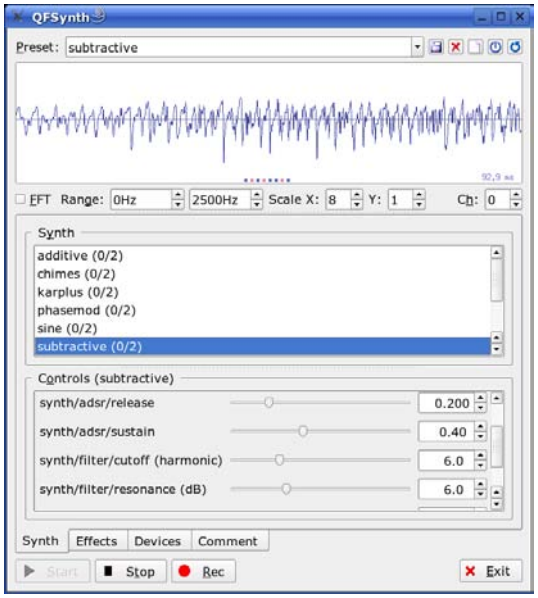

Figure 6: QFSynth program

explicitly cancel the threads.)

Of course the above example is rather limited in functionality (that shouldn't come as a big surprise as it is just about one page of Faust and Q source code). A complete example of a Faust-based polyphonic software synthesizer with GUI can be found in the QFSynth application (cf. Figure 6) which is available as a separate package from the Q website.

3.4 Q, Faust and SuperCollider

The Q-Faust interface provides a direct way to embed Faust DSPs in Q programs, which is useful for testing DSPs and for simple applications with moderate latency requirements. For more elaborate applications it is often convenient to employ a dedicated software synthesis engine which does the grunt work of low-latency control data and audio processing. This is where Q's OSC-based SuperCollider interface (Gräf, 2005) comes in handy. Using SuperCollider's Faust plugin interface, described in the next section, Faust DSPs can also be loaded into the SuperCollider sound server and are then ready to be operated from Q programs via OSC.

4 Faust and SuperCollider3

SuperCollider3 (McCartney, 2002) is a real-time synthesis and composition framework, divided into a synthesis server application (`scsynth`) and an object-oriented realtime language (`sclang`). Any application capable of sending OpenSoundControl (Wright et al., 2003) messages can control `scsynth`, one notable example being Q (section 3).

Correspondingly, support for plugins generated by Faust is divided into an interface to `scsynth` and `sclang`, respectively.

4.1 Interface to `scsynth`

In order to compile a Faust plugin for the SuperCollider3 synthesis architecture, you have to use the corresponding architecture file:

```
$ faust -a supercollider.cpp \
    -o noise.cpp noise.dsp
```

For compiling the plugin on Linux you can use the provided `pkg-config` specification, which is installed automatically when you pass `DEVELOPMENT=yes` to `scons` when building SuperCollider:

```
$ g++ -shared -o noise.so \
    `pkg-config --cflags libscsynth` \
    noise.cpp
```

The resulting plugin should be put in a place where `scsynth` can find it, e.g. into `~/share/SuperCollider/Extensions/Faust` on Linux.

Unit-generator plugins in SuperCollider are referenced by name on the server; the plugin generated by Faust currently registers itself with the C++ filename sans extension. In future versions of Faust the plugin name will be definable in the process specification itself.

4.2 Interface to `sclang`

Faust can produce an XML description of a plugin, including various meta data and the structural layout of the user interface.

This information is used by `faust2sc` in the Faust distribution to generate a SuperCollider class file, which can be compiled and subsequently used from within `sclang`.

For example,

```
$ faust -xml -o /dev/null noise.dsp
$ faust -xml -o /dev/null karplus.dsp
$ faust2sc -p Faust -o Faust.sc \
    noise.dsp.xml karplus.dsp.xml
```

generates a SuperCollider source file, that, when compiled by sclang, makes available the respective plugins for use in synth definitions.

Now copy the source file into `sclang`'s search path, e.g. `~/share/SuperCollider/Extensions/Faust` on Linux.

Since `scsynth` doesn't provide GUI facilities, UI elements in Faust specifications are mapped

to control rate signals on the synthesis server. The argument order is determined by the order of appearance in the (flattened) block diagram specification; audio inputs (named in1 ... inN) are expected before control inputs. The freeverb example plugin has the following arguments to the `ar` instance creation method when used from `sclang`:

```
in1 in2 damp(0.5) roomsize(0.5) wet(0.3333)
```

i.e. first the stereo input pair followed by the control inputs including default values.

4.3 Examples

Unsurprisingly plugins generated by Faust can be used just like any other unit generator plugin, although the argument naming can be a bit verbose, depending on the labels used in UI definitions.

Assuming the server has been booted, the "noise" example found in the distribution can be tested like this:

```
{ Pan2.ar(
  FaustNoise.ar(0.2),
  LFTri.kr(0.1) * 0.4)
}.play
```

A more elaborate example involves the "karplus" example plugin and shows how to use keyword arguments.

```
{
  FaustKarplus.ar(
    play: { |i|
      Impulse.kr(
        exprand(10/6*(i+1), 20)
        * SinOsc.kr(0.1).range(0.3, 1)
      )
    } ! 6,
    duration_samples: LFSaw.kr(0.1)
                      .range(80, 128),
    attenuation: LFPar.kr(0.055, pi/2)
                 .range(0.1, 0.4)
                 .squared,
    level: 0.05
  ).clump(2).sum
}.play
```

Note that the *trigger* button in the *jack-gkt* example has been replaced by a control rate impulse generator connected to the *play* input.

Rewriting the monophonic synth example from section 3.3 in SuperCollider is a matter of recompiling the plugin,

```
$ faust -a supercollider.cpp \
    -o synth.cpp synth.dsp
  $ g++ -shared -o synth.so \
      `pkg-config --cflags libscsynth` \
      synth.cpp
  $ faust -xml -o /dev/null synth.dsp
  $ faust2sc -p Faust -o FaustSynth.sc \
      synth.dsp.xml
```

and installing `synth.so` and `FaustSynth.sc` to the appropriate places.

The corresponding SynthDef just wraps the Faust plugin:

```
(
SynthDef(\faustSynth, {
  | trig(0), freq(440), gain(1),
    attack(0.01), decay(0.3),
    sustain(0.5), release(0.2) |
  Out.ar(
    0,
    FaustSynth.ar(
      gate: trig,
      freq: freq,
      gain: gain,
      attack: attack,
      decay: decay,
      sustain: sustain,
      release: release
    )
  )
}, [\tr]).send(s)
)
```

and can now be used with SuperCollider's pattern system:

```
(
TempoClock.default.tempo_(2);
x = Synth(\faustSynth);
p = Pbind(
  \instrument, \faustSynth,
  \trig, 1,
  \sustain, 0.2,
  \decay, 0.1,
  \scale, #[0, 3, 5, 7, 10],
  \release, Pseq(
    [Pgeom(0.2, 1.5, 4),
     4,
     Pgeom(0.2, 0.5, 4)],
    inf
  ),
  \dur, Pseq(
    [Pn(1/4, 4),
     15.5/4,
     Pn(1/8, 4)],
    inf
  ),
  \degree, Pseq(
    [1, 2, 3, 4, 5, 2, 3, 4, 5].mirror,
    inf
  )
).play(
```

```
  protoEvent: (
    type: \set,
    args: [\trig, \freq, \release]
  )
)
)
```

5 Conclusion

Existing functional programming environments have traditionally been focused on non real-time applications such as artificial intelligence, programming language compilers and interpreters, and theorem provers. While multimedia has been recognized as one of the key areas which could benefit from functional programming techniques (Hudak, 2000), the available tools are not capable of supporting real-time execution with low latency requirements. This is unfortunate since real time is where the real fun is in multimedia applications.

The Faust programming language changes this situation. You no longer have to program your basic DSP modules in C or C++, which is a tedious and error-prone task. Faust allows you to develop DSPs in a high-level functional programming language which can compete with, or even surpass the efficiency of carefully hand-coded C routines. The SuperCollider Faust plugin interface lets you execute these components in a state-of-the-art synthesis engine. Moreover, using Q's Faust and SuperCollider interfaces you can also program the real-time control of multimedia applications in a modern-style functional programming language. Together, Faust, Q and SuperCollider thus provide an advanced toolset for programming DSP and computer music applications which should be useful both for practical application development and educational purposes.

References

Albert Gräf. 2005. Q: A functional programming language for multimedia applications. In *Proceedings of the 3rd International Linux Audio Conference (LAC05)*, pages 21–28, Karlsruhe. ZKM.

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

K. Karplus and A. Strong. 1983. Digital synthesis of plucked-string and drum timbres. *Computer Music Journal*, 7(2):43–55.

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

Y. Orlarey, D. Fober, and S. Letz. 2004. Syntactical and semantical aspects of Faust. *Soft Computing*, 8(9):623–632.

Matthew Wright, Adrian Freed, and Ali Momeni. 2003. OpenSound Control: State of the art 2003. In *Proceedings of the Conference on New Interfaces for Musical Expression (NIME-03)*, pages 153–159, Montreal.

Design of a Convolution Engine optimised for Reverb

Fons ADRIAENSEN
fons.adriaensen@skynet.be

Abstract

Real-time convolution has become a practical tool for general audio processing and music production. This is reflected by the availability to the Linux audio user of several high quality convolution engines. But none of these programs is really designed to be used easily as a reverberation processor. This paper introduces a Linux application using fast convolution that was designed and optimised for this task. Some of the most relevant design and implementation issues are discussed.

Keywords

Convolution, reverb.

1 Introduction

The processing power of today's personal computers enables the use convolution with relatively long signals (up to several seconds), as a practical audio tool. One of its applications is to generate reverberation — either to recreate the 'acoustics' of a real space by using captured impulse responses, or as an effect by convolving a signal with synthesised waveforms or virtual impulse responses.

Several 'convolution engines' are available to the Linux audio user. The *BruteFir* package[1] by Anders Torger has been well known for some years. More recent offerings are Florian Schmidt's *jack_convolve*[2] and *JACE*[3] by Fons Adriaensen.

While one can obtain excellent results with these programs, none of them is really designed to be an easy-to-use reverb application. Another problem is that all of them use partitioned convolution with uniform partition size, which means there is a tradeoff to be made between processing delay and CPU load (JACE allows the use of a period size smaller than the partition size, but this does not decrease the latency). While in e.g. a pure mixdown session a delay of say 100 ms could be acceptable, anything involving live interaction with performers requires much smaller latency.

The *Aella* package written by the author is a first attempt to create a practical convolution based 'reverb engine'. An alpha release[4] will be available at the time this paper is presented at the 4th Linux Audio Conference. In the following sections, some of the design and implementation issues of this software will be discussed.

2 Anatomy of natural reverb

If reverb is added as an effect then everything that 'sounds right' can be used. If on the other hand the object is to recreate a real acoustical space or to create a virtual one, then we need to observe how a natural reverb is built up, and how it is perceived by our hearing mechanisms.

Imagine you are in a concert hall listening to some instrument being played on stage. Provided the sound source is not hidden, the first thing you hear is the direct sound (DS). This will be followed after some milliseconds by the first reflections from the walls and ceiling. The sound will continue to bounce around the room and a complex pattern of reflections will build up, with increasing density and decreasing amplitude. This is shown in fig.1. Traditionally a reverb pattern is divided into an initial period of *early reflections* (ER) lasting approximately 50 to 80 ms, and a *reverb tail* (RT) that shows a quadratic increase in density and decays exponentially. From an acoustical point of view there is no clear border between the two regimes — the distinction is the result of psychoacoustic effects.

The early reflections, while being discrete, are not heard as a separate sound, rather they 'merge' with the direct sound. They provide our hearing mechanisms with important clues as to the direction and distance of the sound

[1] http://www.ludd.luth.se/ torger/brutefir.html
[2] http://www.affenbande.org/ tapas/jack_convolve
[3] http://users.skynet.be/solaris/linuxaudio
[4] http://users.skynet.be/solaris/linuxaudio/aella

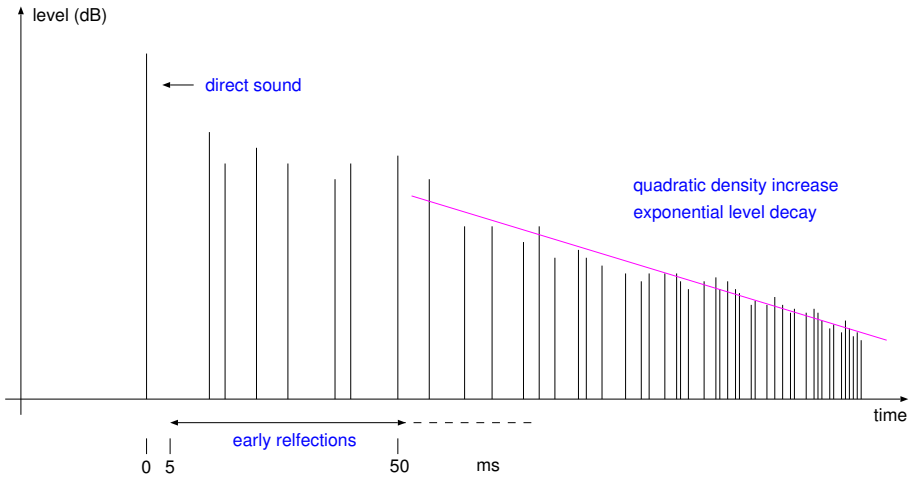

Figure 1: A typical natural reverb pattern

source, and the size and nature of the acoustical environment. The pattern of ER can either reinforce or contradict the results of other mechanisms used by our brain to detect the location of a sound source. For this reason it is important, when recreating an acoustical space, that the ER are consistent with the positioning of sound sources. For a traditional setting with musicians on a stage in front of the audience, at least three (left, centre, right) and preferably more sets of ER should be available to achieve a convincing result. More would be necessary in case the sound sources are all around the listeners.

In concert halls used for classical music, the lateral early reflections (from the side walls) seem to play an important part in how the 'acoustics' are appreciated by the listeners. This is often said to explain why 'shoe-box' concert halls such as the ones at the Amsterdam Concertgebouw or the Musikverein in Vienna are preferred over 'auditorium' shaped ones.

Early reflections very close to the direct sound (less than a few milliseconds) will often result in a 'comb filter' effect, and should be avoided. Discrete reflections that are too far behind the DS or too loud will be heard as a separate 'echo'. These echos occur in many acoustical spaces but not in a good concert hall.

In contrast to the ER, the reverb tail is clearly perceived as a separate sound. A natural reverb tail corresponds to a 'diffuse' sound field with no clear source direction. This doesn't mean that it has no spacial distribution — it has, and this should be reproduced correctly. Of course, the reverb tail will be different for each source (and listener) position, but in general we can not hear this difference — for the RT, only its statistical properties seem to matter. As a result, provided the early reflections are correct, it is possible to use a single reverb tail for all sources.

In most rooms, the RT will show an exponential decay over most of the time. This is not always the case: some spaces with more complex shapes and subspaces (e.g. churches) can produce a significantly different pattern.

3 Requirements for a convolution reverb engine

Taking the observations from the previous section into account it is now possible to define the requirements for a practical convolution based reverb engine.

3.1 Flexibility

In order to be as flexible and general-purpose as possible, the following is needed:

- The engine should allow to combine a number of ER patterns with one or more reverb tails. The number of each should be under control of the user. Separate inputs are required for each ER pattern.
- The relative levels of ER and RT, and the shape of the RT must be controllable. The latter can be used for effects such as e.g. 'gated reverb' that is cut off in the middle of the tail.
- The engine must be able to use a number of formats, from mono to full 3D Ambisonics. It should also support the use of different sample rates.

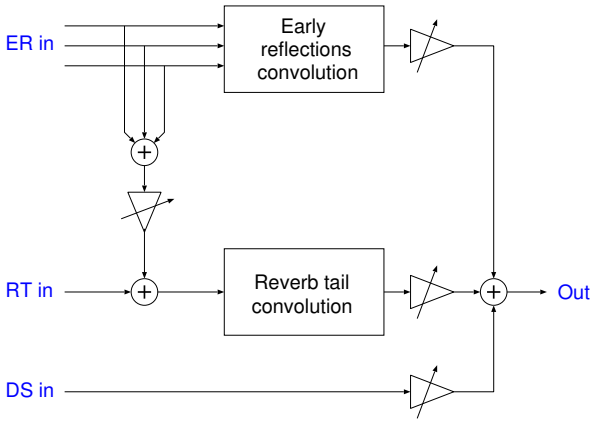

Figure 2: Aella audio structure.

This leads to the general structure shown in fig.2. The ER and RT inputs are always mono, while the DS input and the output will have the number of channels required for the selected format.

In most cases a reverb unit like Aella will be driven from post-fader auxiliary sends from a mixer application and in that case the DS input is normally not used. It is provided for cases where the reverb is used as in insert, e.g. for single channel effects, and to overcome some potential latency problems discussed in the next section.

3.2 Minimal processing delay

The reverb engine should ideally operate with no processing delay, and be usable with all period sizes when running as a JACK client. This turned out to be the most difficult requirement to satisfy. This is discussed in more detail in section 4.

There is even a requirement for *negative processing delay*. This occurs when the output of the reverb is sent back into the same JACK client that is driving it, creating a loop with one period time delay on the returned signal. It *is* possible to compensate for this: remember that the first few (5 to 10) milliseconds after the direct sound should not contain any ER in order to avoid coloration. So provided the period time is small enough, this 'idle time' can be absorbed into the processing delay, provided the DS path is not used. To enable this, Aella provides the option to take the feedback delay into account when loading a reverb pattern.

In case the period time is too long to do this, another solution is to route the direct sound through the reverb and accept a delay on all sound. Aella will always insert the proper delay (not shown in fig.2) into the DS path, depending on its configuration and the period size. Doing this also allows operation with larger processing delay, leading to lower CPU usage.

3.3 Ease of use

It's mainly the requirements from the previous two sections that make general purpose convolution engines impractical for day-to-day work. Having to take all of this into account and keep track of all impulse files, offsets, gains, etc. when writing a configuration script will rapidly drive most less technically inclined users away.

The solution is to automate the complete convolution engine configuration, using only parameters and options that make direct sense to e.g. a musical user. This is what Aella tries to achieve.

Aella first presents a menu of available reverb responses to the user. When one is selected, more information is provided, e.g. in case of a captured real impulse response some info is given on the original space, its reverb time, how the recording was made, etc. A new menu is presented showing the available ER and RT patterns for the selected reverb. The user selects the signal format, the patterns to be used, and some options that enable him or her to trade off CPU load against latency. Immediate feedback about processing delay is provided for the latter. Finally the users clicks the 'LOAD' button, and then all the complex partitioning and scheduling discussed in the next section is performed, and the impulse responses are loaded. The reverb is then ready for use.

Aella uses a single file for each reverb program, containing all the impulse responses (even for different sample rates and formats) and all the extra information that is required. Since this file has to contain binary data (the IR, in floating point format) anyway, and given the author's known aversion to things like XML, it should not come as a surprise that this is a binary file format. It is of course completely open, and (hopefully) flexible enough to allow for future extensions.

Presently these files are generated using a command line tool. In future versions of Aella this function may be integrated into the main program.

4 Using non-uniform partition sizes

The only way to obtain zero processing delay (in the sense that at the end of the process call-

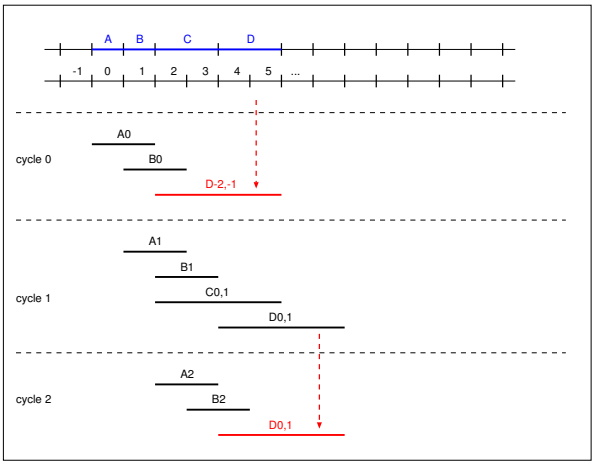

Figure 3: Equal load schema for (1,1,2,2) partitioning

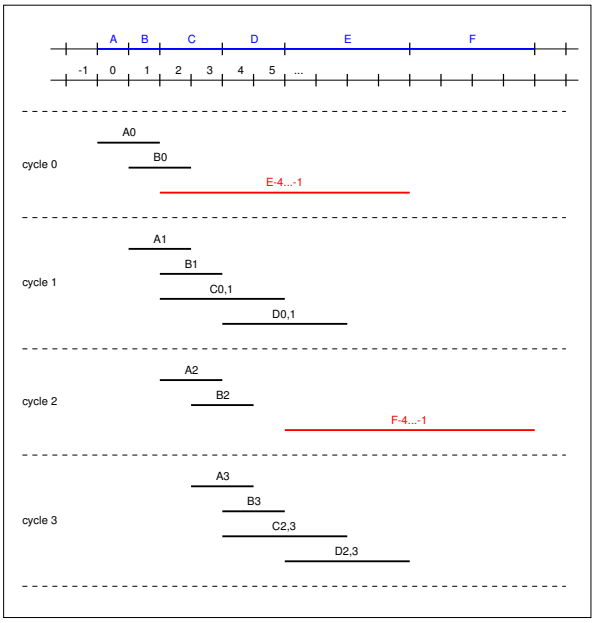

Figure 4: Equal load schema for (1,1,2,2,4,4) partitioning

back the output signal contains the input of the same callback convolved with the first samples of the impulse response) is to use a partition size equal to the period size. For small period sizes it is infeasible to compute the entire convolution using this partition size — the CPU load would be above the limit or unacceptable — so a scheme using a mix of sizes is required.

How to organise a non-uniform partition size convolution so as to obtain the same CPU load for all cycles is known to be a *hard problem*, in the sense that there is no simple algorithm, nor even a complicated one, that provides the optimal solution in all cases. It's an interesting research problem to say the least. One of the referenced papers (Garcia, 2002) will provide a good idea of the state of the art, and of the complexity of some of the proposed solutions.

The problem is further complicated if multiple inputs and outputs are taken into consideration (these can sometimes share the FFT and inverse FFT operations), and even more if the target platform is not a specialised DSP chip with predictable instruction timing but a general purpose PC, and things such as multitasking and cache effects have to be taken into account.

One of the most useful results was published by Bill Gardner as far as ten years ago (Gardner, 1995). Assuming the CPU load is dominated by the multiply-and-add (MAC) phases, and is proportional to the data size, a uniform load can be obtained if the partition sizes follow a certain pattern. Figure 3 provides the simplest example. Here we have four partitions A, B, C, and D with relative sizes 1, 1, 2, 2. The black lines depict the work that can be started in each cycle, for the indicated partition and cycle number(s). For example the line labelled 'C0,1' represents the output generated from the inputs of cycles 0 and 1 and partition C. In the odd numbered cycles we would have three times the work of the even numbered ones. But the computation related to partition D can be delayed by three cycles, so it can be moved to the next even cycle, resulting in an uniform load.

This schema can be generalised for partition sizes following the same doubling pattern. Figure 4 shows the solution for sizes proportional to 1, 1, 2, 2, 4, 4. The red lines in this figure correspond to the computations that have been moved.

There is a limit to what can be done in this way, as the underlying assumptions become invalid very for small period and large partition sizes.

Looking again at fig.4, it is clear that except when partition A is involved, the outputs are required only in the next or in even later cycles. This is even more the case for any later partitions (G, H, ... of relative size 8 or more). So part the work can be delegated to a separate thread running at at lower priority than JACK's client thread (but still in real-time mode). This will increase the complexity of the solution, as the work for later partitions needs to prioritised

in some way in order to ensure timely execution, but in practice this can be managed relatively easily.

In Aella, a mix of both techniques is employed. For the reverb tail convolution a large process delay can be accepted and compensated for by removing the zero valued samples at the start. It uses a minimum size of 1024 frames for the first partition, increasing to a maximum of 8K for later parts (due to cache trashing, nothing is gained by using larger sizes), and all work is done in a lower priority thread.

There is an unexpected potential problem related to moving work to a secondary thread. Imagine the average load is high (say 50%) and the largest partition size is 8K, i.e. there will be something like six such partitions per second. The audio processing by other applications will not suffer from the high load, as most of the work is being done at a lower priority. But the responsiveness of the system e.g. for GUI interaction, or even just for typing text in a terminal, will be severely impaired by a real-time thread executing for several tens of milliseconds at a time. So the work performed in the lower priority thread can not just be started 'in bulk' — it has to be divided into smaller chunks started by triggers from the higher frequency process callback.

For the early reflections convolution the situation is quite different, as it has to provide outputs without any delay. Depending on the period size, Aella will use a minimum partition length of 64 frames, and a schema similar to fig.4 for the first few partitions. Later ones are again moved to a lower priority thread.

The process callback will never *wait* for work done at lower priority to complete — it just assumes it will be ready in time. But it will *check* this condition, and signal a 'soft overrun' (indicated by a flashing light in Aella's GUI) if things go wrong.

For period sizes below 64 frames, Aella will buffer the inputs and outputs, and still try to get the best load distribution. The processing delay will increase due to the buffering, but any idle time before the first early reflection will be used to compensate automatically for the increased delay, in the same way as happens for the feedback delay.

The exact scheme used depends in a quite complicated way on the actual period size and on some user options related to processing delay. This was one of the more difficult to write parts of the application, much more than the real DSP code. The final result is that Aella will be able to operate without any processing delay and at a reasonable CPU load in most practical cases.

References

Guillermo Garcia. 2002. Optimal filter partition for efficient convolution with short input/output delay. *Audio Engineering Society Convention Paper 5660*. 113th AES Convention, Los Angeles October 2002.

William Gardner. 1995. Efficient convolution without input-output delay. *Journal of the Audio Engineering Society*, 43(3):127–136.

Sampled Waveforms And Musical Instruments

Josh Green
11321 Beauview Rd.
Grass Valley, CA, 95945 U.S.A.
josh@resonance.org

Abstract

Sampled Waveforms And Musical Instruments (SWAMI) is a cross platform collection of applications and libraries for creating, editing, managing, and compressing digital audio based instruments and sounds. These instruments can be used to compose MIDI music compositions or for other applications such as games and custom instrument applications. Discussed topics will include: common instrument file formats, Swami application architecture, Python scripted instrument editing, CRAM instrument compression, PatchesDB - a web based instrument database, and basic usage of Swami.

Keywords
Instruments, MIDI, Music

1 Introduction

It was decided early on in the development of Swami that the primary focus would be on providing a tool for editing wavetable based instruments. Wavetable instruments are composed of one or more "samples" of digital audio. While the original project goals have been expanded on, the focus remains the same. In the Linux audio and music paradigm of having many tools with well defined purposes interacting with each other, Swami provides a way to create and manage instrument sounds of specific formats. Useful types of applications to interface to the Swami application include: MIDI sequencers, audio sample editors, and JACK enabled applications. Wavetable synthesizers are also pluggable in the Swami architecture, although currently FluidSynth is the only supported soft synth. In addition, other applications may choose to interface to the underlying libraries for instrument file processing.

2 Instrument Formats

There are countless sample based instrument formats currently in use today. Some are commonly used, others obscure, some are based on open standards, some are proprietary. The goal of the Swami project is to support some of the more commonly used formats with a preference for open standards. Swami currently has varying levels of support for the following formats: SoundFont ®[1] version 2, DLS (DownLoadable Sounds), and GigaSampler ®[2]. Support for some of the popular Akai formats are planned for the future, and there is also some interest expressed in creating a new open instrument format. The rest of this section will be dedicated to describing the 3 currently supported formats mentioned (GigaSampler to a lesser extent).

2.1 Synthesis Model

All the currently supported instrument formats use digital audio as the basis of sound synthesis and have a fixed synthesis architecture. This means that there is a fixed number of synthesis components, each with a specific function. The synthesis components modulate various effects and parameters of the played sounds. Synthesis components and parameters include:

- Audio samples with loops
- Key splits and Velocity splits
- Envelopes
- LFOs (Low Frequency Oscillators)
- Filters (Low Pass Filter for example)
- Stereo panning
- Tuning (pitch of playback)
- Reverb and Chorus
- Effect Modulators (MIDI Controller, etc)

[1] SoundFont is a registered trademark of EMU Systems, Inc

[2] GigaSampler is a registered trademark of Nemesys Music Technology, Inc

2.2 Sample Loops

At the core of the synthesis model are the audio samples. The capability of having continuous sounds can be realized by looping a portion of a sample which also saves on storage space. An alternative is to have very long samples which satisfy the maximum expected note duration.

2.2.1 Key and Velocity Splits

When a MIDI note event is generated it contains the MIDI note # and velocity # (the speed at which the note was played, roughly translates to how hard the key was pressed). Key splits and velocity splits define a MIDI note range and velocity range respectively, for which a given audio sample will sound.

Due to the finite nature of digital sampled audio, a sample becomes more distorted the further from its original pitch it is played. While one could use a single sample for the entire MIDI note range, it is usually undesirable, since the sound will likely deviate far from the instrument or sound being reproduced. For this reason, an instrument or sound is often sampled at several different pitches and each one is given the note range (key split) which surrounds the notes closest to the original sampled pitch. Key splits are also used for percussion kits (usually one note per sound) as well as instruments with layered sounds.

Velocity splits are useful for playing back different sounds and/or different effect settings based on the speed a key is played.

2.2.2 Envelopes

Envelopes provide a simple and convenient way to modify an effect (such as volume) over the note playback cycle. The type of envelopes used in these formats are fixed 6 stage envelopes. Each stage is controlled by a value and are named: Delay, Attack, Hold, Decay, Sustain and Release. A brief description of a Volume Envelope: following the Delay period the Attack stage begins which controls the amount of time it takes for the sound to reach its maximum level, the volume is then held for the Hold time, followed by the Decay stage where the volume decreases to its Sustain level over a specified time and remains until the key is released which causes the volume to return to silence within the Release time interval.

2.2.3 LFOs

LFOs provide low frequency oscillation of effects. They are defined by a delay, frequency, and effect modulation level. They can be used to provide tremolo (volume), vibrato (pitch) and filter oscillations. The sine wave is the most commonly used oscillator waveform, but some formats allow for other types of waveforms as well.

2.2.4 Filter

The most common filter used is the Low Pass filter. This produces a “wah-wah” effect often used with guitars and electronic music. It consists of a filter cutoff frequency and Q (quality) value. As the filter cutoff decreases the frequencies above are attenuated. The Q parameter controls the intensity of the resonance at the cutoff frequency. The higher the Q the more pronounced the frequencies will be near the cutoff.

2.2.5 Modulators

Modulators provide a flexible mechanism for controlling effects in real time via MIDI controls or other parameters such as MIDI note or velocity value. SoundFont and DLS files both provide a similar model which allows one to map one or two controls to an effect using a math function (Linear, Concave, Convex, and Switch), direction, polarity and value amount to apply to the effect.

2.3 Instrument Model

Instrument files are organized into different components (lets call them objects). The most intuitive model of instrument files is a tree of objects which have properties (i.e., parameters). The instrument file is the root object and has parameters such as Name, Author, Date, Copyright, etc. This object in turn contains child objects such as Samples, Instruments and Presets. The Sample objects are rather similar between formats in that they represent a single sample of audio with properties such as SampleRate, RootNote, FineTune, and possibly loop information. The Instrument objects consist of a set of child Region objects (Zones in SoundFont terminology) which each reference a Sample. These regions contain all the synthesis parameters to be applied to the referenced sample (key/velocity split ranges, tuning, envelopes, LFOs, etc). In the case of SoundFont files there is an additional level called Presets which group

together Instrument objects and allow offsetting of effect values for all referenced instruments. Instruments (Presets in the case of SoundFont files) define the unique MIDI bank and program number which each instrument should be mapped to, which are used for selecting it for play.

2.3.1 Feature Comparison

The synthesis models of SoundFont and DLS files are very similar. Both have a Volume Envelope, Modulation Envelope (Filter and Pitch), Modulation LFO (Pitch, Filter and Volume), Vibrato LFO (Pitch only), Low Pass Filter, Reverb, Chorus, Panning and Tuning parameters.

The file format on the other hand differs quite a bit. DLS is much more flexible and allows for custom additions. Samples consist of embedded WAV files, which adds a lot of flexibility as far as the audio format (although only 8 bit and 16 bit is defined for compliance). SoundFont files on the other hand are 16 bit audio only. DLS files also have support for multi-channel instruments (surround sound), whereas SoundFont is limited to stereo. In addition DLS uses 32 bit parameter values and SoundFont uses 16 bit.

GigaSampler files on the other hand have a completely different synthesis model. While the file structure is based on DLS they contain many proprietary extensions. Instruments are composed of dimensions which multiplex samples to ranges of a parameter (such as note, velocity, a MIDI controller, etc). This model is a convenient way of triggering different samples or effect parameters based on different inputs. GigaSampler has support for other specific features also like sample cross fading, different filter types, different LFO waveforms, is designed for sample streaming and some other nice features. The down side of this format is that it is not an open standard and many of the parameter ranges are quite small compared to the other formats.

SoundFont version 2	
Creator	E-mu Systems, Inc.
Pros	Open standard, popular, simple fixed synthesis model, flexible effect modulators.
Cons	16 bit mono or stereo audio only, format not very expandable.

DLS (DownLoadable Sounds)	
Creator	MIDI Manufacturers Association
Pros	Open standard (although v2 spec must be purchased), simple fixed synthesis model, flexible file format, large parameter ranges, adopted by MPEG4.
Cons	Not yet in wide use.

GigaSampler	
Creator	Nemesys Music Technology, Inc
Pros	Designed for streaming, dimension model is nice, cross fading, additional filter and oscillator types.
Cons	Proprietary format, small parameter ranges compared to other formats, more complex synthesis.

3 Swami Architecture

The Swami application consists of several components, including:

- libInstPatch (Lib Instrument Patch) – Object oriented instrument editing library.
- libSwami – All useful non GUI specific functionality can be found in this library.
- SwamiGUI – The GUI front end. Also implemented as a library to allow plugins to link against it.
- Plugins – FluidSynth for sound synthesis and FFTune for computer aided sample tuning.

The underlying libInstPatch and libSwami libraries are written in C and utilize the GObject library. This popular architecture was chosen so as to benefit from object oriented programming while still providing the most flexibility in language bindings. These two libraries are also not dependent on any graphical toolkit, simplifying their usage for non GUI applications.
The GUI is also written in C, in an object oriented fashion, and uses the GTK+ version 2 toolkit and GnomeCanvas. Of note is that GnomeCanvas is not dependent on Gnome and likewise neither is Swami.

3.1 libInstPatch

This library is responsible for loading, saving and editing instrument files. Its features include:

- Instrument files are stored in memory as trees of objects with type specific properties.
- Object conversion system which handles converting between file formats and objects. Also handles loading and saving instrument files and importing samples.
- Paste system for an intuitive method of adding objects to an instrument tree.
- Flexible sample data storage (in RAM, in a file, etc).
- Sample format conversion (bit width, mono/stereo, integer/float).
- Voice caches for fast lock free synthesis (only SoundFont synthesis model currently).
- CRAM instrument compression for supported formats and a generic decompresser implementation.
- Attempts to be multi-thread safe.

3.2 libSwami

Functionality which is not specific to instrument file editing or the GUI finds its way here. Features include:

- Plugin system for extending Swami.
- Wavetable driver object.
- MIDI device object for MIDI drivers.
- GValue based control networks.

Of particular interest is the GValue control networks. GValue is a structure used in the GObject library which acts as a wild card container. It can store an integer, float, enum, flags, string, GObject and more. Control networks are formed by connecting one or more SwamiControl objects together, values or events generated by a control object are sent to all connected objects. Examples of usage include, connecting GUI controls to an instrument parameter. When the parameter changes, all controls are notified with the new value. Another example usage is in routing MIDI events between the virtual keyboard and FluidSynth.

Also of note is the SwamiWavetbl object which forms the basis of adding support for a synthesizer such as the FluidSynth plugin.

3.3 SwamiGUI

The Swami Graphical User Interface is what many users will interface with when editing or managing their instruments. The GUI interface is also object oriented and makes heavy use of Model View Controller functionality (change a parameter all views update, as the user expects). The GUI layout is also flexible so that it may be subdivided at will and interface elements can be changed simply by drag and drop.

Many of the interfaces (virtual keyboard, sample loop editor, splits, etc) are implemented using the GnomeCanvas. Although not perfect, it has helped to create flicker free zoomable widgets without having to pay attention to a lot of details.

3.4 Existing Plugins

Current plugins include the FluidSynth software synthesizer and the FFTune plugin which uses the FFTW library (Fastest Fourier Transform in the West) to assist users in tuning their samples.

4 Python Binding

The 3 library components of Swami each have their own Python binding. The most useful of these is libInstPatch. Using this binding it is possible to create or modify instruments in a scripted fashion. When dealing with files containing many instruments it can become rather tedious to perform an editing operation across all instruments. With a bit of Python knowledge the tediousness of these tasks can be greatly reduced. This also adds a level of ease for users who wish to extend the functionality of Swami at runtime (versus having to write in C). Example uses include: writing custom instrument export or import routines, auto defining key splits based on the optimal note ranges, batch parameter setting or instrument renaming, etc.

5 CRAM Instrument Compression

CRAM (Compress hybRid Audio Media) is a format that was created specifically for compressing instruments. There are a few instrument compression formats in use today, but I found them to be either proprietary, lack cross

platform support or restricted to a specific instrument format. CRAM is not constrained to only instruments though and would likely be useful for other file formats containing binary and audio data as well. The file format uses EBML (created for the Matroska multimedia container format) which was chosen for its compact design and extendability.

Binary compressors such as gzip, bzip2, etc are generally poor at compressing audio. FLAC (Free Lossless Audio Codec) provides lossless audio compression and often achieves much better compression ratios than a standard binary compressor. CRAM utilizes FLAC for audio portions of an instrument file and gzip for the rest. Each audio sample is compressed individually, taking advantage of the knowledge of the bit width and number of channels. The binary is concatenated and compressed as a single stream of data. Instrument compressors must be aware of the structure of the file they are compressing but the decompresser on the other hand is generic, and sees the resulting output as simply interleaved binary and audio segments. CRAM supports multiple files in a single archive and can preserve file timestamps.

In the future support for lossy audio compression may be added to the CRAM format. While lossy audio is generally undesirable for music composition, it might be nice for previewing an instrument online or used as the instruments embedded with a MIDI file on some future web site.

One of the technical difficulties of lossy compression of samples is the phase relationship of the decompressed signal to the original. If the sample contains any loops, they may end up with audible clicks due to differences in the data at the loop end points. It has not been determined yet how pronounced this effect is with a codec such as Vorbis or what is the most effective way to deal with it.

6 PatchesDB

A web based instrument database written in Python. This project was started for the purpose of creating a free, concise, online instrument database.

The current implementation is called the Resonance Instrument Database and can be found at:
http://sounds.resonance.org

Features include:

- User accounts
- Upload and maintain your own instruments
- Support for all libInstPatch supported formats (Python binding used for import)
- CRAM used for compression
- Comment and rating system
- Browse by category, author or perform searches

Future plans include the ability to browse, download and synchronize instruments and information between a local Swami client and an online instrument database; and preview instruments by selecting an instrument, choosing notes or a predefined sequence in a web form and server synthesizes the sound and streams it via an Ogg Vorbis stream.

7 Future Plans

Here are some future ideas for Swami. Many of these are only in the idea stage and may not become reality, but I like to dream about them anyways :)

- SwamiSH – The Swami Shell. A command line interface which could be used instead of the GUI. Imagine editing large instruments remotely or providing a visually impared individual the chance to do some serious instrument editing.
- TCP/IP Jam Sessions – Multiple Swami clients connect together, each user chooses the instruments they would like to play, clients download instruments as necessary, streamed MIDI data is then exchanged (sequencers, MIDI equipment, etc). Users hear the same synthesized instruments with minimal bandwidth overhead. Might be better suited to LAN or dedicated low latency networks, but might be fun and expiremental otherwise too.
- Vector Audio Waveforms – The concept of waveforms described by splines is

appealing to me. Think Structure Vector Graphics, but for audio. Design a waveform from scratch or trace an existing sampled waveform. Resulting synthesis would not suffer from sampling errors or looping continuity issues. Automation of spline control points in a periodic fashion may lead to a new method of audio synthesis. By no means all encompasing, but perhaps useful in its own way.

- GStreamer plugin – Add wavetable instrument support to the GStreamer multimedia framework.
- DSSI interface – A Swami DSSI plugin would likely make integration with other software much smoother. DSSI is an API for audio processing plugins, particularly useful for software synthesis plugins with user interfaces.
- C++ Bindings – For those who prefer C++ but would like to use or interface to Swami or its underlying libraries.
- Open Instrument Format – It may be desirable to design a new flexible and open instrument format. Something along the lines of an XML file which references external audio sample files and has support for vector audio (for envelopes, LFO waveforms, etc). A flexible yet simple synthesis model might be desirable.

8 Resources

- Swami – http://swami.sourceforge.net
- FluidSynth – http://www.fluidsynth.org
- CRAM http://swami.sourceforge.net/cram.php
- Resonance Instrument Database http://sounds.resonance.org
- DSSI - http://dssi.sourceforge.net
- EBML - http://ebml.sourceforge.net

9 Thanks

- Peter Hanappe and Markus Nentwig for creating FluidSynth. I'm happy to be a contributor to that project and will try and be a better maintainer ;)
- Ebrahim Mayat for his continued efforts in testing Swami and FluidSynth on OSX.
- Keishi Suenaga for his recent work on building Swami on win32, it was a dirty job, but somebody had to do it.
- Takashi Iwai for his work on the AWE wavetable driver which sparked my interest in creating an instrument editor.
- Countless others who have provided a place to stay during my travels in Europe.

128 Is Not Enough - Data Structures in Pure Data

Frank Barknecht
GOTO10
Neusser Wall 2
D-50670 Köln,
Germany,
fbar@footils.org

Abstract

A lesser known feature of Miller Puckette's popular audio and media development framework "Pure Data" is the possibility to create user defined graphical data structures. Although the data structures are included in Pd for several years, only recently a sigificant number of users discovered and used this feature. This paper will give an introduction to the possibilities of Pd's data structures for composers and musicians and present several example applications.

Keywords

Pure Data, graphical score, multimedia, programming language, composition

1 Introduction

Since its introduction in 1996[1] Miller Puckette's[2] software environment Pure Data[3], or short Pd, has grown to become one the most popular open source applications amongst media artists. Today Pd not only supports the production of audio or midi data - with extensions like Gem or PDP it is also widely used in the field of video art and multimedia. While the user base of Pd literary is huge compared to most other related free software, one central feature of Pd is not in such a wide use, although it was one of the motivations to write Pd in the first place, according to (Puckette, 1996):

> Pd's working prototype attempts to simplify the data structures in Max to make these more readily combined into novel user-defined data structures.

"Novel user-defined data structures" is the key term here. Simple numbers, let alone just 128 of them as in the MIDI standard, do not provide a sufficient vocabulary for artists in any field. Moreover predefining a limited vocabulary at all is not flexible enough for unforseen and novel uses.

In (Puckette, 2002) Puckette further states about his motivation:

> The underlying idea is to allow the user to display any kind of data he or she wants to, associating it in any way with the display.

So the data structures Pd offers carry another property: They are *graphical* structures, that is, they have a visual representation as well.

Defining a structure for data alone is of not much use unless there are ways to access and change the stored data. For this task Pd includes several accessor objects, which will be explained below. It also is possible to edit the stored data through the graphical representation of a structure using mouse operations.

2 Defining data structures in Pd

The central object to create a structure definition in Pd is called `struct`. While a `struct` object theoretically can be created anywhere in a Pd patch, it generally is put into a Pd subpatch to be able to associate it with instructions for its graphical representation.

Every `struct` needs to be given a name and then one or more fields to carry its data. For example a structure defining a note event might look like this: `struct note float freq float vel float len`.

Besides fields for floating point numbers, a structure can also have fields of type `symbol` which stores a word, and `array`. The latter can be used to carry collections of structures defined elsewhere in Pd. Members of an `array` field have to be of the same type. An example for this could be a score structure, which holds several of our `note` structs in a "melody" and also gets a score-name field: `struct score`

[1] (Puckette, 1996)
[2] `www-crca.ucsd.edu/~msp/`
[3] `www.puredata.org`

`symbol score-name array melody note`. A fourth type that can be used in a `struct` definition is the `list` type, which similar to `array` can hold a collection of other structures, however these elements can be of different `struct` types.

Two field names are treated in a special way: `float x` and `float y` are used to specify the x- and y- coordinates of a structure's graphical representation. Such representations are specified by adding objects for drawing instructions to the same subpatch, that carries the structure definition. A very simple instruction is `drawnumber`, which just displays the value of the field given as its first argument: `drawnumber freq` will draw the current value of the freq-field of a structure. It also is possible to change that value using the mouse by click and drag.

3 Pointer Magic

The structure definition in Pd can be compared to `struct` in programming languages like C. This analogy is taken even further if we look at the way, instances of data structures are created and their data is accessed. Because this is done through pointers much like in C.

Pointers in Pd are a special data type. Other types would be float and symbol — also called "atoms", because they reference a single, atomic value — and lists made of several atoms. Pointers can be made to reference instances of `struct` structures. Pd has an object to hold these pointers which is called `pointer` as would be expected. To make a `pointer` object actually *point* to something in a Pd patch, it has to be told, where to find that something. Subpatches play an important role here.

3.1 Subpatches as named regions

Subpatches in Pd are commonly used to group related functionality and to make better use of the limited screen estate. However they also have a syntactic meaning, because they create a *named region* inside of a Pd patch. This region can be a target for various operations. Every subpatch in Pd can be accessed by sending messages to a receiver that is automatically created by prepending the subpatch's name with the string "pd-". An example for an operation supported by subpatches is `clear`, which deletes every object inside the target subpatch. A subpatch called "editor" thus can be cleared by sending the message `clear` to a sender called `pd-editor` as shown in Figure 1:

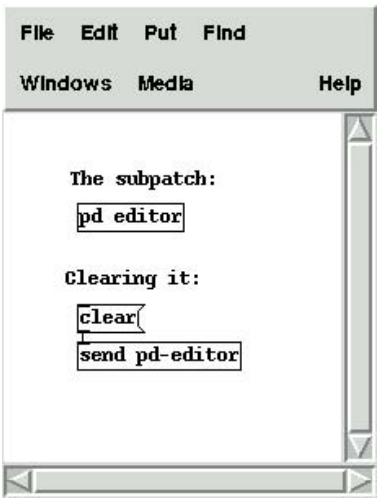

Figure 1: clearing a subpatch with a message

Now if a subpatch contains instances of data structures, these are organized as a linked list, which can be traversed using `pointer` objects. For this, `pointer` supports traversal operations initiated by (amongst others) the following messages:

- `bang`: output pointer to current element
- `next`: output pointer to next element
- `traverse pd-SUBPATCH`: position pointer at the start of the list inside the subpatch called "SUBPATCH".

3.2 Creating instances of data structures

Given a pointer to any position in a subpatch canvas, it is possible to insert new instances of structures using the `append` object. For this, `append` needs to know the type of structure to create and at least one of the fields to set.

Using our note example from above, one could use `append` like this: `append note freq`. For every field specified this way, the `append` object will generate an inlet to set the creation value of this field in the new instance. Additionally it will have a further, rightmost inlet, which has to be primed with a pointer, that specifies the position, after which the new structure instance should be inserted.

Supposing we have a subpatch called `editor` in our patch, we can get a pointer to the start of this subpatch by sending the message `traverse pd-editor` followed by `bang` to a `pointer` object, that itself is connected to `append`s rightmost inlet. Sending a number like 60 to the

leftmost inlet (the `freq` inlet) will then create a graphical instance of `struct note` inside the subpatch `editor`, whose frequency field is set to 60 and whose other fields are initialized to zero. For now, as we only used a `drawnumber freq` as single drawing instruction, the graphical representation is quite sparse and consists just of a number in the top-left corner of the subpatch.

3.3 Get and set

The current values of fields in this new instance of a `struct note` can be read using the `get`-object, which on creation needs to know, which `struct`-type to expect and which fields to read out. If a valid pointer is sent to a `get note freq vel` object, it will output the current value of the frequency and the velocity field stored at that pointer's position.

The opposite object is called `set` and allows us to set the fields inside an instance, that is specified by sending a pointer to the rightmost inlet of `set` first.

By traversing a whole subpatch using a combination of `traverse` and `next` messages sent to a `pointer` that is connected to `get`, reading out a whole "score" is easily done.

Accessing `array` fields is slightly more complicated, because it requires an intermediate step: First we need to `get` the array-field out of the initial pointer of the structure. The array field is itself represented by a pointer. This pointer however can be sent to the right inlet of an `element` object, that on its left inlet accepts an integer number to select the element inside of the array by this index number.

4 Drawings

Unless the subpatch containing the `struct` definition also has drawing instructions, the structure instances will be invisible, when they are created. Pd offers several graphical primitives to display data structures. `drawnumber` was already mentioned as a way, to draw a numeric field as a number. If the `struct` has float-typed fields called `x` and `y` this number also can be moved around in the subpatch in both dimensions. The x- and y-fields are updated according to the current position with the upper left hand corner of the subpatch window being at a point (0,0). The x-axis extends from left to right, the y-axis extends from *top to bottom* to make (0,0) stay at the upper left.

The various drawing primitives accept coordinates to specify their positions and dimensions as well, however these are calculated in a local coordinate system, whose (0,0)-point is translated to the point specified by the value of the fields `float x float y` in the `struct` definition. For example, using this definition `struct coord float x float y` we can use two `drawnumber` objects to draw `x` and `y` without overlapping by creating `drawnumber x 0 0` and `drawnumber x 0 15`. The y-field will be drawn 15 pixels below the x-field, as shown in figure 2 (which also shows colors and labels in `drawnumber`).

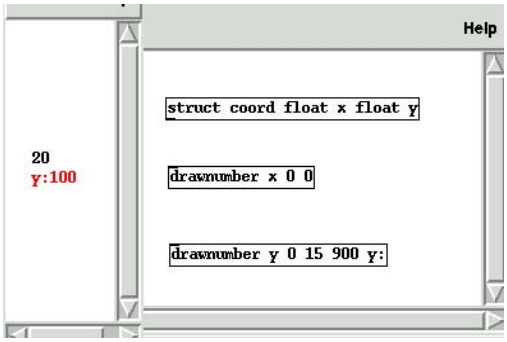

Figure 2: relative positioning

4.1 More drawing instructions

Further objects for drawning data are `drawpolygon`, `filledpolygon`, `drawcurve`, `filledcurve` and `plot`. They are described in their respective help patches. Here we will only take a look at `drawpolygon`, that is used to draw connected line segments. Like all drawing instructions it accepts a list of positional arguments to control its appearance and behavior. In the case of `drawpolygon` these are:

- optional flag `-n` to make it invisible initially
- alternatively a variable given by `-v VAR` to remotely control visibility
- Color specified as RGB-values.
- line-width in pixels
- two or more pairs of coordinates for the start and end points of the line segments.

The next instruction would draw a blue square of width `w`: `drawpolygon 9 1 0 0 w 0 w w 0 w 0 0` (Fig. 3)

The size of the square, that is, the variable `w` in this structure, can be changed using the mouse or with `set` operations. The current value of `w` always is accessible through the `get` object, see section 3.3.

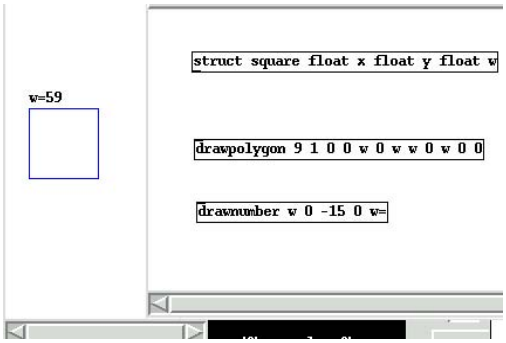

Figure 3: a blue square

4.2 Persistence

Saving a patch containing instances of data structures will also save the created data structures. Additionally it is possible to export the current contents of a subpatch to a textfile by sending the message `write filename.txt` to the subpatch's receiver (described in 3.1) and read it back in using a similar `read`-message.

Such a persistence file contains a textual description of the data structure templates and their current values, e.g. for our `square`-structure:

```
data;
template square;
float x;
float y;
float w;
;
;
square 136 106 115;
```

5 Data structures in action

Instead of explaining all the features in detail, that data structures in Pd provide, I would like to present some examples of how they have been used.

5.1 Graphical score

Pd's data structures most naturally fit the needs of preparing and playing graphical scores. In his composion "Solitude", north american composer and Pd developer Hans-Christoph Steiner used data structures to edit, display and sequence the score. The graphical representation also controls the sequencing of events. He describes his approach in a post to the Pd mailing list:[4]

> The experience was a good combination of visual editing with the mouse and text editing with the keyboard. The visual representation worked well for composition in this style. [My] biggest problem was finding a way to represent in the score all of the things that I wanted to control. Since I wanted to have the score generate the piece, I did not add a couple features, like pitch shifting and voice allocation control, which I would have liked to have.

Both the Pd patch (Fig. 4) and a recording of "Solitude" are available at the composer's website.[5]

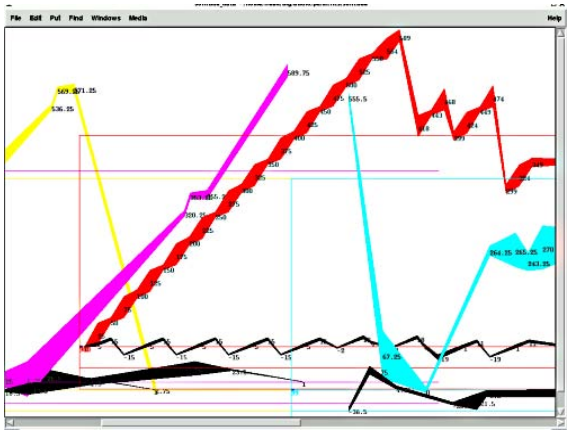

Figure 4: Score for "solitude" by Hans-Christoph Steiner

5.2 Interaction

"Solitude" changes the data stored inside a structure only during the compositional phase, but not in the performance of the piece. An example, where data structures are manipulated "on the fly" is the recreation of the classic video game "PONG" by the author[6], as shown in Fig. 5.

This simple piece uses the ratio of the current score in the game (1/2 in Fig. 5) to influence a fractal melody played in the background. The x-position of the ball is read out by a `get`-object to pan the stereo position of the melody.

5.3 GUI-building

Data structures also are useful to implement custom GUI elements for Pd. A collection of

[4] `lists.puredata.info/pipermail/pd-list/2004-12/024808.html`

[5] `at.or.at/hans/solitude/`

[6] `footils.org/cms/show/27`

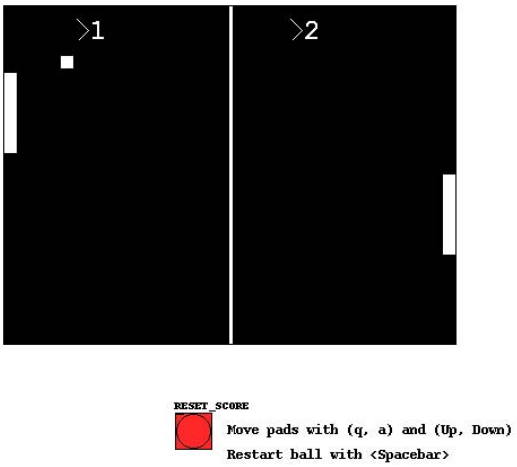

Figure 5: "PONG" by Frank Barknecht

these is currently built by Chris McCormick.[7] Figure 6 shows an envelope generator and a pattern seqencer of variable length, that can be reused several times in a patch.

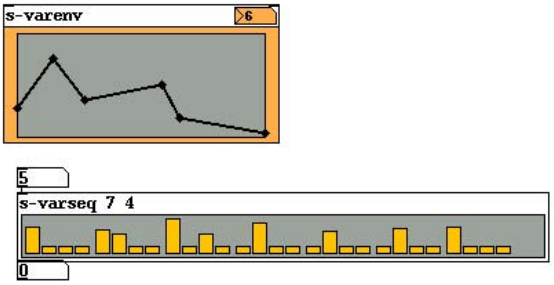

Figure 6: Two GUI objects by Chris McCormick

5.4 Visualisation

Many advanced concepts in computer music or digital art in general deal with rather abstract, often mathematical issues. Data structures can help with understanding these concepts by connecting the abstract with the visible world.

The german composer Orm Finnendahl created such interactive Pd patches using data structures to explain things like the sampling theorem or granular synthesis.

With his patches "pmpd-editor" and "msd-editor" (Fig. 7) the author of this paper wrote tools to explore particle systems (masses connected by springs) interactively. A user can create the topology for a particle system and

[7] mccormick.cx/viewcvs/s-abstractions/

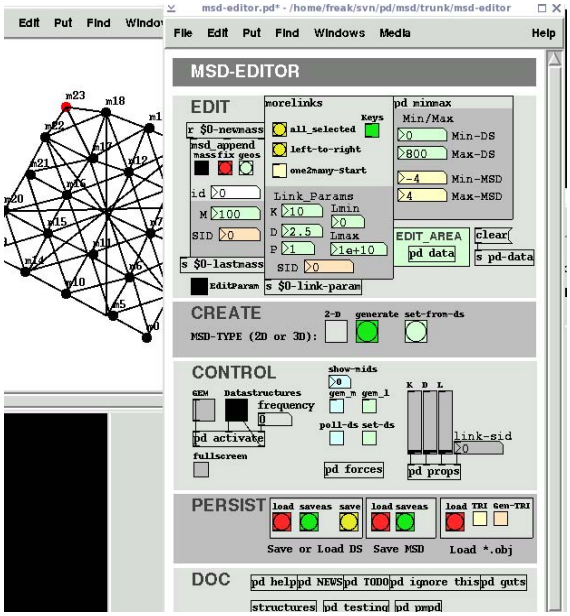

Figure 7: Editor for mass-spring-damper-topologies by Frank Barknecht

animate it directly inside the Pd patch. Various helper functions provide means for importing and exporting such topologies to be used in other applications as 3D-modellers for example. The particle systems designed with the editor can also generate control data to influence various synthesis methods. The editor is available in the CVS repository of the Pure Data developer community at `pure-data.sf.net`.

5.5 Illustration

Finally we get back to another original motivation for writing Pd in the first place. In (Puckette, 1996) Puckette writes:

> Pd's first application has been to prepare the figures for an upcoming signal processing paper by Puckette and Brown.

Almost a decade later, Puckette is still using Pd to illustrate paper, this time for his book project "Theory and Techniques of Electronic Music".[8]

All graphics in this book were made using Pd itself, like the one shown in Fig. 8.

6 Conclusion

This paper could only give a short introduction to Pd's data structures. As always they

[8] So far only available online at: `crca.ucsd.edu/~msp/techniques.htm`

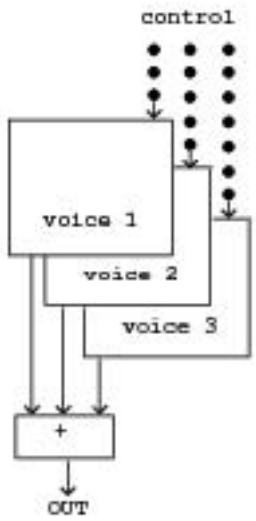

Figure 8: Illustration courtesy by M. Puckette from his book: "Theory and Techniques of Electronic Music"

are best learned by studying examples. Pd comes with documentation patches that can be edited and changed. Most patches that use data structures are collected in the directory "doc/4.data.structures" of the Pd documentation. The HTML-manual of Pd contains further information in chapter "2.9. Data structures".[9]

Pd's data structures are powerful tools that can greatly enhance the possibilities of Pd. In some areas they still are a bit awkward to use though. For example animating large numbers of data structures may influence audio generation and even lead to dropped samples and clicks. There also still are issues with providing a smooth framerate. In the author's view data structures thus cannot replace specialized extensions like Gem in this regard yet. If they should try to do so at all, remains an open question.

However problems like this can only be found and fixed, if more artists and musicians in the Pd community will actually use them—a classical chicken and egg problem. Thus it is hoped that this paper will create more interest in Pure Data's data structures.

[9] `www-crca.ucsd.edu/~msp/Pd_documentation/`

7 Acknowledgements

References

M. Puckette. 1996. Pure data: another integrated computer music environment. In *Proc. the Second Intercollege Computer Music Concerts*, pages 37–41.

M. Puckette. 2002. Using pd as a score language. In *ICMC Proceedings*, pages 184–187.

A Sample Accurate Triggering System for Pd and Max/MSP

Eric Lyon
Music - School of Arts, Histories and Cultures
The University of Manchester
Martin Harris Building
Coupland Street
Manchester, M13 9PL
United Kingdom,
eric.lyon@manchester.ac.uk

Abstract

A system of externals for Pd and Max/MSP is described that uses click triggers for sample-accurate timing. These externals interoperate and can also be used to control existing Pd and Max/MSP externals that are not sample-accurate through conversion from clicks to bangs.

Keywords

Computer Music, Drum Machines, Max/MSP, Pd, Sample Accurate Timing

1 Introduction

In the world of experimental electronic music, regular pulsation has often been frowned upon. During an exchange of ideas between Karlheinz Stockhausen and several younger electronic musicians (Witts 1995) Stockausen observed, "I heard the piece Aphex Twin of Richard James carefully: I think it would be very helpful if he listens to my work Song Of The Youth, which is electronic music, and a young boy's voice singing with himself. Because he would then immediately stop with all these post-African repetitions, and he would look for changing tempi and changing rhythms, and he would not allow to repeat any rhythm if it were varied to some extent and if it did not have a direction in its sequence of variations." Richard D. James responded from a different perspective, "I didn't agree with him. I thought he should listen to a couple of tracks of mine: "Didgeridoo", then he'd stop making abstract, random patterns you can't dance to."

2 The Need for Better Timing

The same year this interview was published, I attempted to use a pre-MSP version of Max to control a drum machine I had built in Kyma. This experiment also resulted in "random patterns you can't dance to," since the Max event scheduler at the time was good enough for certain kinds of algorithmic music, yet not stable enough for techno music. Ten years later, the event schedulers for both Max/MSP and Pd are much more stable, and are quite usable for some forms of music based on regular pulsation. However, their performance is still subject to variability based on factors such as the signal vector size and competition from control-level events. Furthermore, the scheduling systems of Max/MSP and Pd differ such that the timing behavior of similar patches can perform quite differently between the two systems. The Max/MSP event scheduler is prone to permanently drift from a sample accurate measurement of timing. The underlying Pd event scheduler is better than sample-accurate, though apparently at the cost of a higher likelihood of interruption of the audio scheduler, resulting in audible glitches. In both systems temporal accuracy of control-level events can drift freely within the space of a signal vector.

2.1 The Problem with Small Time Deviations

Even when the amount of deviation from sample accuracy is not clearly noticeable at a rhythmic level, it may still have undesirable musical effects. For example, a pulsation may feel not quite right when there are a few 10s of milliseconds of inaccuracy in the timing from beat to beat. Even smaller inaccuracies, though rhythmically acceptable, can still cause problems when sequencing sounds with sharp transients, since changes in alignment on the order of a couple of milliseconds will create different comb filtering effects as the transients slightly realign on successive attacks. This is a particularly insidious artifact since many users might not think to trace a spectral effect to a system timing flaw.

3 Sketch of a Solution

One way to sidestep the abovementioned problems is to program trigger scheduling at the sample level, rather than at the event level at

which bangs perform. This scheduling must be built into every external that is to benefit from sample accurate triggering. In order to be of much use, such externals must be able to easily synchronize with each other. I have developed a system of externals based on click triggers. The trigger signal contains a non-zero value at every sample where a trigger is to be sent, and zeros at all other samples. The click trigger can convey one additional piece of information to its receiver, such as desired amplitude.

4 Sample Accurate Metronomes

The centerpiece of the system is an external that coordinates multiple metronomes. It is called `samm~` (for sample accurate multiple metronomes). The first argument to `samm~` is the tempo in BPM, followed by a series of beat divisors that each define the metronome speed for a corresponding outlet. For example, the arguments 120 1 2 3 7 would activate four outlets, all beating at 120 BPM, the first at a quarter note, the second at an eighth note, the third at an eighth note triplet and the fourth at a sixteenth note septuplet. Any of these parameters can have fractional components, and the beat divisors may be less than 1.0 (resulting in beat durations greater than a quarter note). A click trigger from `samm~` is always a sample with the value 1.0. The tempo can be varied during performance while preserving proportional relations among all beat streams.

4.1 Alternative Metronome Specifications

For convenience, several different methods are provided for specifying metronome tempi . A new set of beat divisors may be specified with the message "divbeats." Beat durations may be specified directly in milliseconds with the message "msbeats." Beats may be specified in samples (useful if tempi need to be built around a soundfile in terms of its length in samples) with the message "sampbeats." Finally beat durations may be specified with ratio pairs (with the denominator representing a division of a whole note) using the message "ratiobeats." The message "ratiobeats 1 4 3 16 5 28" specifies relative beat durations of respectively a quarter note, a dotted eighth note and five septuplets. Fractions may be employed to represent more complex ratios, though it is probably simpler in that case to represent such ratios with decimal numbers and use the "divbeats" message.

5 Pattern Articulation

The beat streams from `samm~` can be patterned into arbitrary rhythms with another external called `mask~`. This external stores a sequence of numbers, which are sent out in cyclic series in response to click triggers. An initial series is given as a set of arguments to mask~. For example, the arguments 1 0 0 0 1 0 0 0 1 0 0 0 1 0 1 0 could serve to define a rhythmic pattern for a single instrument in a drum machine, in this case perhaps a kick drum. Since zeros cannot trigger an attack, any zero in a `mask~` pattern will convert an incoming beat to a rest. Since any non-zero number can serve as a trigger, the attacks need not all have value "1" but could specify different amplitudes instead. Multiple `mask~` instances could control different parameters of the same event, all sample-synched. Since `mask~` patterns can be of any size (up to 1024 members), different sized `mask~` patterns will cycle in and out of phase with each other, which is desirable in a poly-metric scheme. It is also possible for two `mask~` patterns of the same size to be out of sync with each other if, for example, one `mask~` was created later in the design of a given patch. This loss of sync is usually not desirable. Thus, `mask~` provides an option whereby the input is interpreted not as triggers, but rather as index numbers used to iterate through the `mask~` pattern. Using the same indexing clicks (generated from another `mask~` of course) guarantees that all patterns so controlled remain locked in phase. Any `mask~` external can hold a large number of different patterns which may be stored and recalled during performance.

6 Sample Accurate Synthesizers

Sample accurate externals are provided for sound production through both sampling and synthesis. `adsr~`, an external that is already part of my Web-published Max/MSP external set LyonPotpourri (Lyon 2003)is an ADSR envelope generator. I have retrofitted `adsr~` to respond to click triggers, interpreting the value of the click as the overall amplitude of the envelope. Any software synthesis algorithm that uses `adsr~` as an envelope can now be triggered with sample accuracy.

7 Sample Accurate Samplers

A sample playback external called `player~` is provided, which plays back a sample stored in a buffer (for Max/MSP) or an array (for Pd). The

two parameters to `player~` are amplitude and playback increment, sent as signals to the first and second inlets respectively. Amplitude is always a click trigger. Under normal conditions, playback increment is a signal that can be manipulated during performance. An alternative static increment mode is provided, called by the 'static_increment' message, in which the playback increment is also received as a click trigger, which persists throughout the note playback without variation.

7.1 Polyphonic Nature of player~

One important feature was added for convenience - `player~` is polyphonic (up to 16 voices at present, though this limit may be opened up to the user as a parameter in a later version). An inconvenient feature of `groove~`, `tabosc4~`, et. al. is that if a note is currently active when a new playback trigger arrives, the current note is instantly truncated which often creates discontinuities. In `player~`, all currently active notes continue playing to the end of their buffers, even as new attack triggers arrive. This is much more convenient than having to create a `poly` structure for every member of a drum machine. This is also the reason for the static_increment mode. In static_increment mode multiple instances of playback can proceed at different playback increments, which is quite handy for creating polyphony from a single sample.

8 Putting the Pieces Together - a Simple Drum Machine

Now let's look at an example of how the externals described thus far can be combined. (See Figure 1.) A `samm~` unit with a tempo of 120 BPM creates two beat streams, the first dividing the quarter by two (eighth-notes) and the second dividing the quarter by four (sixteenth-notes). Two `player~` objects play back samples stored in two arrays. The bdbuf `player~` takes its metronome from the eighth-note beat stream. Its attack/amplitude pattern is stored in the `mask~` object directly above it. The increment is fixed at 1.0, taken from a `sig~` object. The output is scaled and sent to the DACs.

8.1 Polyrhythmic Sequencing

The hihat structure is slightly more complicated than that of the bass drum. The beat stream is sixteenth-notes in all cases. The duration of the attack/amplitude pattern is one beat, rather than the four beats of the bass drum pattern. But a second pattern with a

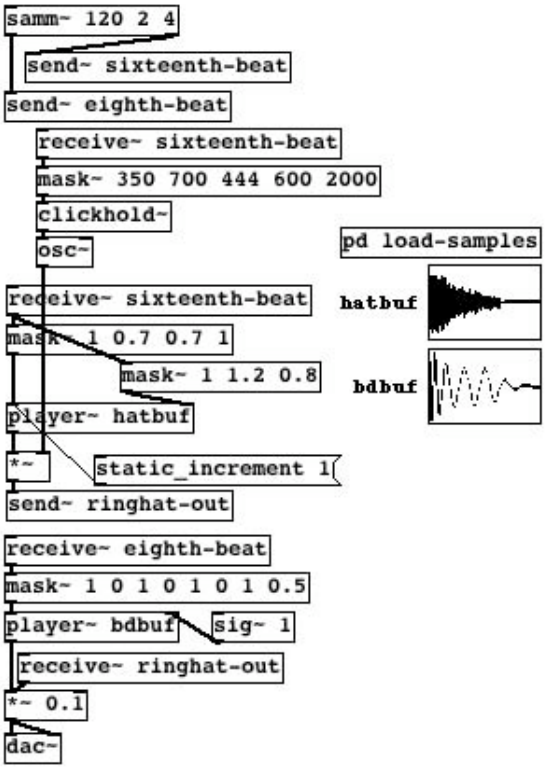

Figure 1: A two voice drum machine.

periodicity of three sixteenth-notes controls the increment from a second `mask~` object routed to the second (increment) inlet of the hatbuf `player~`. A third rhythmic level is added as the hihat output is ring-modulated by the sine wave output of an `osc~` object. The `osc~` is controlled by a frequency pattern with a periodicity of five sixteenth-notes. A custom object, `clickhold~`is inserted between the `mask~` and the `osc~` to sample and hold each click as it comes in, resulting in the sustained frequency signal required by `osc~`. As the three different patterns go in and out of phase with each other, a composite 15-beat pattern emerges. More complex polyrhythmic arrangements are easily imagined, especially to control various parameters of synthesis algorithms rather than the relatively simple sample playback.

8.2 You Call *That* a Drum Machine?

It is quite clear that the Pd patch shown in Figure 1 does not look anything like a conventional drum machine. Of course it is possible to use some graphical objects to wire up an interface that that looks more like a drum machine and serves as a front end, generating patterns for the `mask~` objects. But this sort of tidy front

end would also limit our possibilities. The very looseness of the scheme of distributed `mask~` objects suggest more fluid ways of thinking about drum patterns, and manipulating them during performance.

9 dmach~ - an Integrated Drum Machine External

The combined use of `samm~` and `mask~` can create arbitrarily complex rhythms. However certain kinds of rhythms are somewhat inconvenient to specify under this model. Consider a 4/4 bar pattern where the first beat is divided into 16th notes, the second beat into triplets, and the last two beats divided into eighth-note quintuplets. A representation of this pattern requires three different beat streams and three different mask~ objects. In order to address this problem, a proof-of-concept external called `dmach~` has been designed. `dmach~` contains an internal clock, and stores user-specified patterns. The patterns are sent as outlet pairs; `dmach~` is designed to send both attack patterns and increment patterns. These patterns can be recalled at will during the performance. The current pattern is looped until such time as a new one is recalled. Patterns are stored with the 'store' message and recalled with the 'recall' message. The current pattern is played to completion before a new pattern is loaded, thus guaranteeing a sample-synced performance. The last outlet of `dmach~` sends a click at the start of each pattern playback, making it easy for the user to build a sequencer for stored patterns.

9.1 Pattern Specification for dmach~

Pattern specification in `dmach~` allows for arbitrary bar sizes and arbitrary subdivisions within the bar. Patterns are entered into `dmach~` with the 'store' message. The first two parameters are the pattern number and the number of beats in the bar. The pattern number is an integer which will be used to recall the pattern. The number of beats is defined as the number of quarter notes in a bar. A specification of 4 creates a 4/4 bar. A 3/8 bar would be specified with a bar duration of 1.5. Following are a series of beat patterns, each one targeted toward a different instrument. The first parameter is the designated instrument. (Instrument 0 has its beat stream come out of the first outlet, and its increment stream out of the second outlet of `dmach~`.) Next is a beat duration representing a segment of the bar. If the entire beat

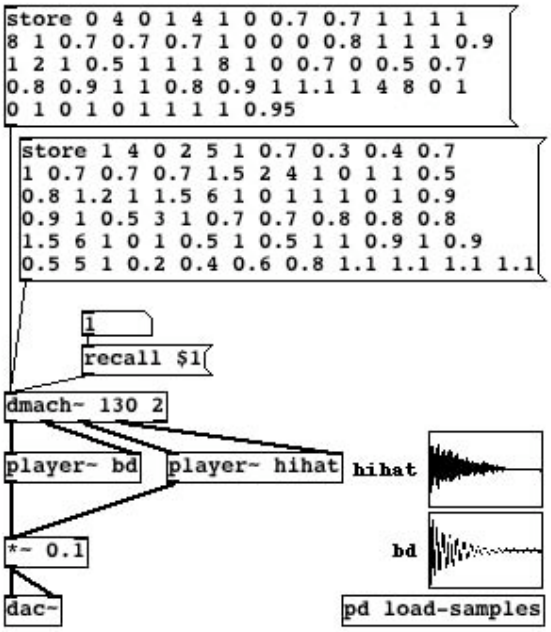

Figure 2: A `dmach~` two voice drum machine.

pattern shares a single subdivision of the beat, then this segment would simply be the same as the bar duration. However since an arbitrary number of segments can be specified (on condition that they eventually add up to precisely the bar duration), an arbitrary rhythmic subdivision of the bar is possible, which could be quite complex. Following the segment duration is the segment subdivision factor. This subdivision must be an integer. Following this is an attack pattern that must have the same number of elements as the subdivision factor just specified. Following the attack pattern is the increment pattern. The increment pattern has the same number of elements as non-zero attacks in the attack pattern. In other words, increments are specified only for attacks, not for rests. Additional segments are identically specified until the duration of the bar is filled. This process is repeated for every instrumental beat/increment stream required for the pattern. Data is only required for instruments that play in a given pattern. A pattern with no instrument data functions as a bar of rest.

9.2 Usability of dmach~

As mentioned above, `dmach~` is a proof-of-concept external. Complete error checking on user input to `dmach~` is a thankless task that has not yet been undertaken, thus it is presently easy to crash `dmach~` with bad data, taking

down Pd or Max/MSP with it. Given the intricacy of the data format, it is recommended that a preprocessor be used to generate the data from a more user-friendly interface than typing raw data by hand. It is interesting to consider what might be a suitable graphical interface to design patterns for `dmach~` though such considerations are beyond the scope of this paper. The complexity of pattern specification for `dmach~`, while potentially burdensome, is also necessary in order to obtain full rhythmic flexibility. Indeed this flexibility goes considerably beyond what is possible with most commercial drum machines. However as mentioned above, the user can be buffered from this complexity with a suitable interface, at the cost of some loss of flexibility in pattern design. As can be seen in Figure 2, the data complexity is localized in the 'store' messages, so the patching structure for wiring up a drum machine in Pd with `dmach~` is somewhat simpler than in the earlier example with multiple `mask~` objects.

9.3 Relative Inflexibility of dmach~

While it is convenient to bind increment patterns to attack patterns in dmach~ this arrangement is somewhat inflexible. The user might prefer not to control increment, or to control increment out of sync (or even randomly) in which case the additional outlets for increment become superfluous, as does the burden of specifying increment pattern in the 'store' messages. On the other hand, one might well wish to simultaneously control parameters other than or in addition to increment, such as pan location, filter parameters, ring modulation frequency or multiple synthesis parameters, if a software synthesizer is being driven by a particular `dmach~` beat stream.

9.3.1 Increasing Flexibility for dmach~

A simple method to increase the flexibility of `dmach~` would use the second beat stream outlet to send attack index numbers which could then be used to control multiple `mask~` objects. This would give full flexibility, though the pattern data would in most cases be spread over multuple `mask~` objects. In some cases this could be an advantage since individual data streams could be changed independently during performance. A more complicated solution would allow the user to specify the structure of a given `dmach~` object through its creation arguments, such that a given beat pattern could have an arbitrary number of outlets in addition to its attack pattern outlet. This would keep all the pattern data in a single 'store' message. However the complexity of maintaining data in this form, along with the possibility of eventually bumping up against the hard limit on the number of atoms allowed in a Max/MSP or Pd message box, might make this solution unwieldy in practice. However a sufficiently flexible graphical interface that could create and manage the data in the 'store' messages with arbitrary structure, might make this approach worth pursuing. As mentioned above, `dmach~` is still a prototype object, which is not yet ready for prime-time. However `dmach~` does raise interesting questions about structuring control data within the sample accurate triggering system under discussion.

10 Interoperation with Sample Inaccurate Externals

Many useful externals exist in Pd and Max/MSP which do not currently provide sample accurate response to triggers. In order to utilize such externals in the system presented here, they must be triggered with a bang synchronized to the incoming click trigger. Max/MSP makes this quite easy to do via the `edge~` external which sends bangs on changes from one to zero. Since `edge~` is only available for Max/MSP, a custom external called `click2bang~` has been designed to send out a bang in response to an incoming click. The bang can only be accurate to within the size of the signal vector. However the receiver can be isolated in a sub-patch with a signal vector size set to 1 by the `block~` object, forcing that part of the patch back to sample accuracy.

11 Clicks and Continuity

While click triggers are conceptually simple and thus easy to work with in designing patches, they do have one disadvantage. Once sent there is no further information on the progress of a triggered process until the next click is received. For the types of data discussed here this is not a problem. However certain continuous processes such as filter sweeps might need to be correlated to the progress of a given time span. For most practical purposes a `line` or `line~` object triggered by an appropriate message (itself triggered by a `click2bang~`) will suffice. However it would be fairly easy to design an external that outputs precise continuous data in response to click triggers. We might call such an external `clickline~` which would receive a target value

and the duration over which to interpolate from a stored initial value to the target value, with all data sent as clicks.

12 Conclusions and Future Work

The click trigger model has proved easy to implement, useful for designing rhythmic patches in Pd and Max/MSP and enables a degree of timing precision for rhythmic events that is not generally practical for Pd and Max/MSP. I plan to incorporate this model into any future externals I design. In the spirit of generosity, I will suggest future work for others as well as for myself. There has been increased interest in the Max/MSP community for sample-accurate timing. Some more recent (and recently updated) Max/MSP object such as `sfplay~` and `techno~` employ sample-accurate triggering, albeit generally using more complicated methods than the click triggering system described here. It would be nice to see a unified sample-accurate triggering system employed to encompass the many Pd and Max/MSP externals that could benefit from it, such as `tabplay~` and `groove~`. Third party developers of externals might also find this model useful for any of their objects that involve triggering. Finally, all of the work described here is based on steady pulses. However it would be interesting to develop metronomes that implement arbitrary tempo curves, which would also output click triggers. This would allow for sample-accurate exploration of a very different class of rhythms.

13 Acknowledgements

Thanks to Peter Castine for a brief Internet discussion on the numerical accuracy of double floating point variables, confirming my calculation that `samm~` will keep all of its metronomes sample-synched indefinitely for any practical musical performance, given current human lifespans. Thanks also to the two anonymous LAC reviewers for helpful comments and suggestions on the first draft of this paper.

References

Lyon, Eric. *LyonPotpourri, a Collection of Max/MSP externals.* 2003. Dartmouth College. Accessed 3 Jan. 2006. <http://arcana.dartmouth.edu/~eric/MAX/>.

Witts, Dick. "Advice to Clever Children/Advice from Clever Children." *The Wire* Nov. 1995: 33+.

Scripting Csound 5

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

Abstract

This article introduces a scripting environment for Csound 5. It introduces the Csound 5 API and discusses it use in the development of a TclTk scripting interface, TclCsound. The three components of TclCsound are presented and discussed. A number of applications, from simple transport control of Csound to client-server networking are explained in some detail. The article concludes with a brief overview of some other Csound 5 language APIs, such as Java and Python.

Keywords: Music Programming Languages; Scripting Languages; Computer Music Composition.

1 Introduction

The Csound music programming system (Vercoe 2004) is currently the most complete of the text-based audio processing systems in terms of its unit generator collection. Csound hails from a long tradition in Computer Music. Together with cmusic (Moore 1990), it was one of the first modern C-language-based portable sound compilers (Pope 1993), when it was released in 1986. Due to its source-code availability, first from the cecelia MIT ftp server and then from the DREAM site at Bath, it was adopted by composers and developers world-wide. These brave new people helped it to develop into a formidable tool for sound synthesis, processing and computer music composition. Its latest version, Csound 5 (ffitch, 2005) has close to one thousand opcodes, ranging from the basic table lookup oscillator to spectral signal demixing unit generators.

Many important changes have been introduced in Csound5, which involved a complete redesign of the software. This resulted not only in a better software, from an engineering perspective, but in the support for many new possible ways of using and interacting with Csound.

An important development has been the availability of a complete C API (the so-called 'Host API', which was, in fact, already partially present in earlier versions). The API can be used to instantiate and control Csound from a calling process, opening a whole new set of possibilities for the system.

2 The Csound 5 API

The Csound 5 Host API allows the embedding of the audio processing system under other 'host' software. Effectively, Csound is now a library, libcsound, that can provide audio services, such as synthesis and processing, for any application. This allows for complete control over the functioning of the audio engine, including transport control, loading of plugins, inter-application sofware bus, multithreading, etc.. A 'classic' csound command-line program can now be written based only on a few API calls:

```
#include <csound.h>
int main(int argc, char **argv) {

int result;
CSOUND *cs; /* the csound instance */

/* initialise the library */
csoundInitialize(&argc, &argv, 0);
/* create the csound instance */
cs = csoundCreate(NULL);
/* compile csound code  */
result = csoundCompile(cs, argc, argv);
/* this is the processing loop */
if(result) while(csoundPerformKsmps(cs)==0);
/* destroy the instance */
csoundDestroy(cs);
return 0;
}
```

The Csound API can be used in many applications; the development of frontends is the most obvious of these. A good example of its application is found on the csoundapi~ Class, which provides a multi-instantiable interface to Csound 5 for Pure Data. The Csound API is the basis for TclCsound (Lazzarini 2005), a Tcl/Tk extension, discussed in the next section.

3 TclCsound

The classic interface to csound gives you access to the program via a command-line such as

```
csound -odac hommage.csd
```

This is a simple yet effective way of making sound. However, it does not give you neither flexibility nor interacion. With the advent of the API, a lot more is possible. At this stage, TclCsound was introduced to provide a simple scripting interface to Csound. Tcl is a simple language that is easy to extend and provide nice facilities such as easy file access and TCP networking. With its Tk component, it can also handle a graphic and event interface. TclCsound provides three 'points of contact' with Tcl:

1. a csound-aware tcl interpreter (cstclsh)
2. a csound-aware windowing shell (cswish)
3. a csound commands module for Tcl/Tk (tclcsound dynamic lib)

3.1 The Tcl interpreter: cstclsh

With cstclsh, it is possible to have interactive control over a csound performance. The command starts an interactive shell, which holds an instance of Csound. A number of commands can then be used to control it. For instance, the following command can compile csound code and load it in memory ready for performance:

```
csCompile -odac hommage.csd -m0
```

Once this is done, performance can be started in two ways: using `csPlay` or `csPerform`. The command

```
csPlay
```

will start the Csound performance in a separate thread and return to the cstclsh prompt. A number of commands can then be used to control Csound. For instance,

```
csPause
```

will pause performance; and

```
csRewind
```

will rewind to the beginning of the note-list. The csNote, csTable and csEvent commands can be used to add Csound score events to the performance, on-the-fly. The `csPerform` command, as opposed to `csPlay`, will not launch a separate thread, but will run Csound in the same thread, returning only when the performance is finished. A variety of other commands exist, providing full control of Csound.

3.2 Cswish: the windowing shell

With Cswish, Tk widgets and commands can be used to provide graphical interface and event handling. As with cstclsh, running the cswish command also opens an interactive shell. For instance, the following commands can be used to create a transport control panel for Csound:

```
frame .fr
button .fr.play -text play -command csPlay
button .fr.pause -text pause -command csPause
button .fr.rew -text rew -command csRewind
pack .fr .fr.play .fr.pause .fr.rew
```

Similarly, it is possible to bind keys to commands so that the computer keyboard can be used to play Csound.

Particularly useful are the control channel commands that TclCsound provides. For instance, named IO channels can be registered with TclCsound and these can be used with the invalue, outvalue opcodes. In addition, the Csound API also provides a complete software bus for audio, control and string channels. It is possible in TclCsound to access control and string bus channels (the audio bus is not implemented, as Tcl is not able to handle such data). With these TclCsound commands, Tk widgets can be easily connected to synthesis parameters.

3.3 A Csound server

In Tcl, setting up TCP network connections is very simple. With a few lines of code a csound server can be built. This can accept connections from the local machine or from remote clients. Not only Tcl/Tk clients can send commands to it, but TCP connections can be made from other sofware, such as, for instance, Pure Data (PD). A Tcl script that can be run under the standard tclsh interpreter is shown below. It uses the Tclcsound module, a dynamic library that adds the Csound API commands to Tcl.

```
# load tclcsound.so
#(OSX: tclcsound.dylib, Windows: tclcsound.dll)
load tclcsound.so Tclcsound
set forever 0

# This arranges for commands to be evaluated
proc ChanEval { chan client } {
if { [catch { set rtn [eval [gets  $chan]]}
err] } {
puts "Error: $err"
} else {
puts $client $rtn
flush $client
}
}

# this arranges for connections to be made
```

```
proc NewChan { chan host port } {
puts "Csound server: connected to $host on port
$port ($chan)"
fileevent $chan readable [list ChanEval $chan
$host]
}

# this sets up a server to listen for
# connections
set server [socket -server NewChan 40001]
set sinfo  [fconfigure $server -sockname]
puts "Csound server: ready for connections on
port [lindex $sinfo 2]"
vwait forever
```

With the server running, it is then possible to set up clients to control the Csound server. Such clients can be run from standard Tcl/Tk interpreters, as they do not evaluate the Csound commands themselves. Here is an example of client connections to a Csound server, using Tcl:

```
# connect to server
set sock [socket localhost 40001]

# compile Csound code
puts $sock "csCompile -odac hommage.csd"
flush $sock

# start performance
puts $sock "csPlay"
flush $sock

# stop performance
puts $sock "csStop"
flush $sock
```

As mentioned before, it is possible to set up clients using other software systems, such as PD. Such clients need only to connect to the server (using a netsend object) and send messages to it. The first item of each message is taken to be a command. Further items can optionally be added to it as arguments to that command.

3.4 A Scripting Environment

With TclCsound, it is possible to transform the popular text editor e-macs into a Csound scripting/performing environment. When in Tcl mode, the editor allows for Tcl expressions to be evaluated by selection and use of a simple escape sequence (ctrl-C ctrl-X). This facility allows the integrated editing and performance of Csound and Tcl/Tk code.

In Tcl it is possible to write score and orchestra files that can be saved, compiled and run by the same script, under the e-macs environment. The following example shows a Tcl script that builds a csound instrument and then proceeds to run a csound performance. It creates 10 slightly detuned parallel oscillators, generating sounds similar to those found in Risset's *Inharmonique*.

```
load tclcsound.so Tclcsound

# set up some intermediary files
set orcfile "tcl.orc"
set scofile "tcl.sco"
set orc [open $orcfile w]
set sco [open $scofile w]

# This Tcl procedure builds an instrument
proc MakeIns { no code } {
global orc sco
puts $orc "instr $no"
puts $orc $code
puts $orc "endin"
}

# Here is the instrument code
append ins "asum init 0 \n"
append ins "ifreq = p5 \n"
append ins "iamp = p4 \n"


for { set i 0 } { $i < 10 } { incr i } {
append ins "a$i  oscili iamp,
         ifreq+ifreq*[expr $i * 0.002], 1\n"
}

for { set i 0 } {$i < 10 } { incr i } {
    if { $i } {
         append ins " + a$i"
    } else {
         append ins "asum = a$i "
    }
}

append ins "\nk1 linen 1, 0.01, p3, 0.1 \n"
append ins "out asum*k1"

# build the instrument and a dummy score
MakeIns 1 $ins
puts $sco  "f0 10"

close $orc
close $sco

# compile
csCompile $orcfile $scofile -odac -d -m0

# set a wavetable
csTable 1 0 16384 10 1 .5 .25 .2 .17 .15 .12 .1

# send in a sequence of events and perform it
for {set i 0} { $i < 60 } { incr i } {

    csNote 1 [expr $i * 0.1] .5 \
      [expr ($i * 10) + 500] [expr 100 + $i *
10]

}

csPerform

# it is possible to run it interactively as
# well
csNote 1 0 10 1000 200
csPlay
```

The use of such facilities as provided by e-macs can emulate an environment not unlike the one found under the so-called 'modern synthesis systems', such as SuperCollider (SC). In fact, it is possible to run Csound in a client-server set-up, which is one of the features of SC3. A major advantage is that Csound provides about three or four times the number of unit generators found in that language (as well as providing a lower-level

approach to signal processing, in fact these are but a few advantages of Csound).

3.5 TclCsound as a language wrapper

It is possible to use TclCsound at a slightly lower level, as many of the C API functions have been wrapped as Tcl commands. For instance it is possible to create a 'classic' Csound command-line frontend completely written in Tcl. The following script demonstrates this:

```
#!/usr/local/bin/cstclsh
set result 1
csCompileList $argv
while { $result != 0 } {
set result csPerformKsmps
}
```

This script is effectively equivalent to the C program shown in section 2. If saved to, say, a file called csound.tcl, and made executable, it is possible to run it as in

```
csound.tcl –odac hommage.csd
```

4 OTHER LANGUAGE WRAPPERS

It is very likely that many users will prefer to run Csound from their programming environment of choice. For these, C++, Lisp (using CFFI), Java and Python are other available languages. Access to the Csound library is provided by SWIG-generated wrappers.

4.1 Python and Java examples

The way these languages interact with the Csound 5 library is very similar to the C API. A major difference is that they are require to import the Csound module (based on a 'native' library module), called `csnd`. In Python, the csnd.py and csnd.pyc files, distributed with Csound, hold the Python API, whereas in Java, the csnd.jar archive holds the csnd package.

Generally, for a straight performance, the steps involved are:

1. Create a Csound instance:

Java:
```
cs = new Csound();
```

Python:
```
cs = csnd.csoundCreate(None);
```

2. Compile Csound code:

Java:
```
cs.Compile("hommage.csd");
```

Python:
```
csnd.csoundCompile(cs, 2,
       ['csound', 'hommage.csd'])
```

3. Run a processing loop:

Java:
```
int result;
while(result == 0)
   result = cs.PerformKsmps();
```

Python:
```
while result == 0:
   result =csoundPerformKsmps(cs);
```

4. Clean up, ready for a new performance:

Java:
```
cs.Reset();
```

Python:
```
csnd.csoundReset(cs);
```

The Java and Python wrappers open up many new possibilities for using Csound programmatically. There are, though, a few aspects of the C API, which are very C-specific and do not translate into Java or Python. However, these should not make any impact on the majority of the applications that the system might have.

5. Conclusion and Future Prospects

Csound is a very comprehensive synthesis and musical signal processing environment. The additional facilities provided in its latest version have brought it up-to-date with more modern software engineering concepts. Its existence as a library has enabled a variety of new uses and added new 'entry-points' into the system. It is very likely that such enhancements will also spur further features, esp. when the new parser for the language, under development by John ffitch, is introduced. It will then be possible to develop other ways of interacting with the system, such as alternative synthesis languages, interpreters and further facilities for networking and distributed operation.

5 References

John ffitch. On the design of Csound 5. *Proceedings of the 2005 Linux Audio Conference*,ZKM, Karlsruhe, Germany.

Victor Lazzarini. The TclCsound frontend and language Wrapper. *Sounds Electric 2005*. NUI, Maynooth, Ireland.

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).*

Barry Vercoe et Al. 2005. *The Csound Reference Manual*.
http://www.csounds.com/manual/html/index.html

Linux and music out of the box

Hartmut NOACK
www.linuxuse.de/snd
Max-Steinke-Strasse 23
13086 Berlin,
BRD,
zettberlin@linuxuse.de

Abstract

The perception of Linux audio software amongst so-called "average musicians" with no "geek"background is not as positive as it could be. Many musicians still think that there is not enough usability and too little Linux software available to produce an album. This proposal introduces design studies for a user interface dedicated to a Linux Audio Workstation (LAW) that I intend to build based on standard PC hardware. The interface intends to integrate all the main virtues of Linux audio software in a logical and self explanatory way, without encumbering their flexibility, to provide the same level comfort and intuitive operation as offered by suites such as Steinberg's Cubase. There will be no install-CD to be downloaded, as offered by Distros like Demudi or Agnula, but only some useful scripts and LFS-style documentation to enable people to rebuild the box itself if they wish to do so.

Keywords

usability, integration, suite, interface, workstation

1 Introduction

Musicians often state that they don't know very much about computers, that they only want to use these boxes and that they have neither the time nor motivation to learn how a computer or software works. Proprietary software vendors try to adapt to such attitudes by designing all-in-one applications, such as Steinberg's Cubase, with simple-looking interfaces and automatic setup assistants that leave power usage and fine tuning to the experienced users and hide possible options and - of course - the source code from the "end user" to keep their products under control and consistent.

This is not the way that Linux audio can go, since it is open source. Free software authors, as well as the distributors, need to develop other ways to achieve usability and a trustworthiness that meets the needs of productive use. This can be done if the whole audio workstation is developed and built as an integrated combination of software **and** hardware components, that fit together and are Linux compatible down to the last screw.

Some may ask:

> is this still possible without charging the users for software ?

we say: It is! - if we find a way to charge the users for integration and support... it is our belief that the time is nigh to make people invest in free audio software development. People like Paul Davis, Taybin Rutkin or Werner Schweer should be rewarded for their work. We believe that a reasonable part of the revenue that could be generated with oss-based audio workstations should go directly to the developers of the most important software for these boxes - especially if these developers do not yet receive any reasonable financial support.

To make sure that the system as a whole works flawlessly enough for studio use, it will be necessary to setup and test each box by hand before shipping. We did so in Suse 9.3 to build the first prototype - on the second box, that we plan to build in June, it will be done with Ubuntu including packages from the Ubuntu studio metadistribution. The project does not intend to add another distribution, toolbox or file format to the already existing mass of experiments in Linux audio - we only want to help others using all the great stuff that is already there. So all scripts will be native, average bash-syntax and XML as used in xfce4. The LAW itself will be built, set up and tuned by hand, the scripts will be usable in every Linux-environment that has xfce4 and the incorporated applications.

2 plans and sketches

Most users coming from proprietary tools, such as Steinberg Nuendo or SEKD Samplitude, have a rather ambivalent impression of the system when looking at Linux solutions. On the one

hand they like tools such as Jamin or snd, but on the other hand they are disappointed by the lack of features that they have commonly come to expect in proprietary applications. The concept that a complex system can be built out of many little tools that provide each other with functions is not very common amongst users of proprietary systems[1].
So to add the comfort known from commercial suites to the Linux audio tools, primarily means integrating all the little, cute and powerful free tools to make them work properly together, without limiting their flexibility.

What have we left behind in our Fat32 partitions?

Applications such as Steinberg's NUENDO can be used to record a complex album without learning more then half a dozen keyboard commands (though one can use hundreds of them - if you wish...) i.e.: you only have to learn to understand the metaphors for things that you already know from hardware mixers, recorders and FX boxes, to make NUENDO work for you - you do not need to know about computers, because the GUI makes it look as if it where some space age version of the hardware that you already know.[2].

Under the hood one could find a framework made of engines, plug-in-interfaces and routing mechanisms, quite similar to jack and ladspa, the main difference is indeed the GUI: you don't need to think about it: everything in one place and all needed functions labelled more or less descriptively in more or less logical menus. Right click - double click and sometimes CTRL or Shift - that's it. There are 2 prerequisites for this type of comfort:

1. Every part of the suite works with the given drivers and its interface fits with the rest.
2. The parts do not conflict with each other

Both can be achieved with free components under Linux, but it takes a lot of effort to set it up and there is still no integrated user interface that allows truly intuitive work with the full power of available possibilities and without unwanted surprises. The ingredients of applications such as NUENDO are available for Linux and PC hardware properly supported by Linux is also available. So the first steps to build a Linux Audio Workstation would be:

1. To design a hardware setup for a reasonable price, that fits perfectly together and is completely Linux proof.
2. To develop a set of scripts and consistent templates and presets that allow the user to switch on the full force of a jack-environment with a single click.

Commercial developers have a major advantage over Linux hackers: the proprietary Systems (i.e. MS Windows XP and Mac OSX) are consistent, applications can be monolithic and survive for years with few changes. Testing and development is easier with less diversity. If Linux software is installed as a binary package and automatically set up, then Linux will not be as stable and consistent as is required. Many developers work on Fedora or Debian and most of them massively alter/tune their systems - so that users who try to to run preconfigured applications on out-of-the-box systems from Novell or Mandriva will be confronted with the unpredictable effects of diversity. Since we cannot relay on consistency - we need to *use* the diversity and freedom of free software to prove that free development can lead to the same and even better functionality as known from commercial vendors. Whereas the basic system can safely be installed automatically, the important applications (jackit, ardour, muse and around 12 more) have to be compiled at the box with sane ./configure-flags and the whole system setup needs to be examined, adapted and thoroughly tested by humans. To limit the effort to a level that can be reached with a small financial budget and in reasonable time, there can be only 1 PC setup and maybe 1 or two laptop machines. The latter will be compromised versions because they must more often serve as office/web-machines than a PC built as a dedicated audio workstation.
To those who want to use our system without purchasing or rebuilding the whole box, we recommend using Ubuntu Linux dapper drake to install the needed components. We have also started a tutorial in the wiki of http://www.audio4linux.de, that describes every step to set up the stuff on Ubuntu dapper

[1] there is indeed buzz and of course PD for MS Windows - but those who know these systems well are not in desperate need for clickable wrapper-scripts in Linux....

[2] Steinberg even tries to resemble the look of real equipment by using bitmaps that look rusty and scratched - just like the stomp boxes you are familiar with...

drake, and also provides links to download the needed files.
The main goal of the native approach is stability. No bleeding-edge components, no x86_64-CPU and no beta/cvs versions of software, as far as this can be avoided. The Workstation should start out of the box up to kdm/gdm with no configuration needed (while still retaining all the configuration possibilities that the user may wish for). The user will have the opportunity of choosing between 3-4 preconfigured setups:

- Audio workstation (preset-configuration with factory support, guaranteed functionality, limited configurability)
- Audio workstation experimental (same preset-configuration as above, but with full configurability, limited support and no guarantee - this would also be available for download.)
- PC Workstation (Standard home computer with all the power of a Linux desktop system, Internet, office, graphics/layout etc.)
- Rescue system (with direct access to scripts, that reset configurations and replay backups - this may be offered at the boot screen)

Sometimes the question arises, do we make this with KDE or GNOME? Since both systems are used to provide typical desktop automation mechanisms (office, pim etc) they are not optimal for an audio workstation. An exception to this is the "PC Workstation". Whereas this may work with KDE, the audio systems and the rescue system were set up with Fluxbox, which appeared to us to be the best compromise between leanness and configurability/comfort. Experimenting with other WMs finally led us to the decision to switch to fxce. It is leaner then KDE or GNOME and as scriptable as Fluxbox, but also comes with some very useful features such as multiple menus and bitmap icons for a more descriptive interface. However, all the scripts, templates and presets can be used in any Desktop environment - templates and presets rely on fitting versions of the respective audio applications and the scripts only require bash ... KDE-base must be also installed, since the wrapper scripts utilise kdialog to display messages and konqueror is used to show HTML help pages.

2.1 What exists already and how we are different

There are both integrated hardware solutions with Linux and CD and/or metadistros available for installation out there. The hardware systems are designed and priced for semi-professionals and we don't know of any Linux-based audio solution that is also a decent Desktop PC. Our aim is to provide a box that can be built for about 700,- EUR and that can also serve as a Desktop/Internet computer.
The installable Distros such as Demudi, jacklab or CCRMA all have one thing in common: they work with binaries, they do little to integrate the apps and they leave the choice and setup of the hardware to the user. All these people still do great work and it is definitely not our intention to replace any of them, but rather to collaborate with them.
We are not trying to build our own little world, but wish to incorporate things that are already there and glue them together in a useful way. As mentioned before, we primarily deliver simple wrapper scripts, presets, templates and samples. These will work on any distro that has the software installed, which is called by these scripts and can handle these presets etc. On the box that we ship, the same things will run like a charm (no kidding - our concept allows intense testing of the very machine that will be delivered - so unwanted surprises will be seldom...) - if one wants to use the stuff in a similar but different environment, adaptations may be needed.

We follow a paradigm that favours a grassroots style growth of the project. So we have built a standard desktop PC with an Intel PIV 2.8 CPU 1024MB DDR-RAM and a Terratec EWX 24/96 audio card - rather average real-world hardware available for less then 700,- .

After about 8 months of intense testing and experimentation the box is now configured as follows:

- Suse 9.3 Linux system with default kernel 2.6.11.4-20a and alsa 1.0.9 out of the box
- jackit 0.100.0, ardour 0.99,. Muse 0.7.2, Hydrogen 0.9.2 compiled on the machine with Suse's gcc and libs
- recent rezound and snd wave-editors also compiled on the box
- a plethora of synths and smaller tools from Suse-RPM-repositories

We tested the setup by using it to record several multitrack sessions, composing and manipulating music with midi-driven softsynths (neither real keyboards nor external midi hardware so far) and by editing and optimising a noisy-tape-to CD - job for the complete "Ring" by Richard Wagner (plus several similar smaller jobs for string quartet and for rehearsal tapes). The setup works well and stable and provides everything we need.
The issues regarding compatibility/stability are solved so far (though it would not be wise, to actually *guarantee* full stability for all needed programs under all conditions...)

2.1.1 ... and what we are working on at the moment

We have begun to build the previously mentioned framework of helping elements, consisting of 4 major components:

- XFCE WM configuration scripts, which allow access to all needed features in a logical and comfortable manner
- several scripts and hacks to load complex scenarios and to provide additional help text
- a set of templates for all major applications that also involve the collaboration between them
- about 300 free licensed presets, sounds and patterns

Starting setups of several collaborating applications *could* be trivial - if all involved elements were aware of each other. Today we still face the problem of certain softsynths that can be attached to jackd via their own command line, and others that need to be called up via tools such as *jack connect*. These are not serious obstacles of course, but there is no reason not to address smaller annoyances as well as great todos. The communication between the several processes is particularly critical and often leads to ruin - this should be remarked amongst developers and distributors.

The website http://www.linuxuse.de/snd offers downloads of sample scripts and templates plus some help texts. More help is to be found at http://www.audio4linux.de and we are also working on an extensive user manual that can serve as an introduction to Linux audio. On the final System, in addition to the help provided by the programmers, there will be also be 3 levels of extra help for the users:

- kdialog popups explaining things that happen, ask the user for needed interaction and point to more detailed help.
- HTML help files for *every* application that explain the basics and how the whole system works. It will be possible to start scripts directly from these files, which will be shown in KDE's Konqueror. (Security people may be not that enthusiastic about the help system...)
- an online forum (PHPBB plus a simple wiki) with daily appearance of at least one developer/help author.

3 next steps

I would like to present a wish list to the developers and to the user community as well. Developers should improve their documentation and users should start to read it.... . Back in 2003 we had a set of experimental stuff that could be used but was limited by some crucial weaknesses, especially the lack of stability. Today we have the tools complete and solid enough to start to discover how to deploy them as perfectly as possible. To do this, there must be a vivid community of more than just about 100 users[3]

We have combined Ardour, Muse, Rezound, Hydrogen, AMS, ZynaddSubFX and about 20 tools such as Ladspa, qalsatools etc. into a setup that is powerful and usable for us as experienced Linux users, and we have made a feature-freeze from the 1st of January 2006 until the 1st of June 2006, to use the time to make the setup powerful and usable for everyone that wants to deal with music and computers.

We will then offer the complete Linux Audio Workstation, which can be used to give soundtech people and musicians a chance to find out that Linux audio is ready to run for everybody.

4 Conclusions

We believe that free audio software can be an important, powerful way to make Linux visible to the public and thus to make the very concept of collaborative, open and free production of software a success. We not only believed that

[3] http://www.frappr.com/ardourusers lists 80 Ardourusers today, Taybin Rutkin pointed the users at the ardour-mailing lists to this page in November 2005...

Linux audio is somewhat usable now and could have a niche - we believe that within 2-3 years it can develop into a superior solution for creative sound people in the same manner as Linux has become a success in motion picture animation/CGI - production.
This can be done if it it becomes easier to use, more logically integrated, more stable and more consistent **without** hiding anything of its great opportunities from the user.

At LAC I would like to present our approach to making Linux audio usable for everyone to developers and users as well.

5 Acknowledgements

Our thanks go to ... Paul Davis ., Werner Schweer, Nasca O. Paul and to all the other great Linux audio developers out there, and to the people at www.audio4linux.de, www.mde.djura.org (Thac and Ze) , ccrma.stanford.edu/planetccrma and all the other websites, mailing lists and BB's where Linux audio springs to life...

The 64 Studio distribution – creative and native

Daniel James
64 Studio Ltd.
PO Box 37
Freshwater,
PO40 9ZR,
Great Britain
daniel@64studio.com

Free Ekanayaka
64 Studio Ltd.
PO Box 37
Freshwater,
PO40 9ZR,
Great Britain
free@64studio.com

Contents

1. The 64-bit question
2. Package selection
3. Why Debian?
4. The business model
5. Challenges
6. Conclusion
7. Acknowledgements
8. References
9. Appendix: Packages in version 0.6.0

Abstract

This paper describes the high integration of proprietary software for the creative desktop, and the effort involved in creating a free software alternative which will run natively on the latest 64-bit x86 hardware. It outlines the author's reasons for creating a 64-bit distribution based on Debian, the packages selected, the business model of the 64 Studio company and the challenges for future development.

Keywords

64-bit, Debian, content creation

Introduction

If we take a step back from pure audio software for a moment, and look at the state of creative desktop computing tools more generally, it's obvious that there has been a lot of consolidation among the proprietary software vendors in the last couple of years. For example Yamaha swallowed up Steinberg, Adobe bought Syntrillium (the creators of Cool Edit), Avid bought Digidesign and Apple bought Logic. Adobe's 'partnership' with Macromedia became a takeover, and now the company positions its extensive range of multimedia applications as the 'Adobe Platform'. What this means is that regardless of the hardware or operating system in use, the mainstream creative desktop of the near future is likely to represent a highly integrated set of non-free applications from a very small number of vendors. We can expect these proprietary applications to be well tested for performance, reliability and usability.

We believe that it will be very difficult for GNU, Linux and other free software to compete for users on the multimedia desktop unless it can achieve a similar level of integration and polish. Without a significant user base, it becomes difficult for free software to maintain the hardware support that it needs. Reports indicate that it is becoming progressively more difficult to obtain full specifications for video card driver development, and several of the most popular high-end audio interfaces, particularly the FireWire models not running BeBoB, remain without the prospect of a free software driver. We aim to deliver a viable and sustainable creative platform based on free software, and partner with hardware manufacturers to ensure the availability of fully-supported components and peripherals.

1 The 64-bit question

Since any software project takes a while to get to a mature stage, when we launched a new multimedia distribution last year, we decided to concentrate on the kind of desktop systems which we believe will be common among creative users in the future.

We're interested in 64-bit x86 for two main reasons - the improvements in memory architecture, allowing commodity machines to have many gigabytes of RAM, and the opportunity to drop support for legacy PC hardware. From the point of view of a distribution, any technology that narrows down the field of potential hardware combinations is a great advantage. We don't have to support ISA bus sound cards and we don't have to care if the binaries won't run on a 486.

That may sound a little harsh for owners of older hardware, but there will be plenty of 32-bit GNU/Linux distributions around for some time, and the relentless downward spiral in the cost of newer technologies looks set to make using any hardware older than a year or two quite counter-productive. For example, the HP laptop which we are giving this presentation on is an entry-level model from a department store in the UK. It has a 1.6GHz AMD Turion 64-bit processor and 1GB RAM as standard. It cost less than a far slower generic white-box PC of a couple of years ago, and it probably uses a great deal less energy too.

We've had native 64-bit Linux on the Alpha and the Itanium for years, but these architectures never reached the mainstream desktop. SGI has an Itanium2 based GNU/Linux desktop product aimed at the creative market, but it costs US $20,000 per machine. Compared to Windows or any other operating system, GNU/Linux clearly had a head start on x86_64, and you can choose from a range of natively compiled desktop distributions for the hardware. Unfortunately for the creative user, all of these are aimed at the general purpose computing audience. It's impossible to be all things to all people, and what's good for the so-called 'consumer' is rarely right for the content creator.

2 Package selection

For example, typical distributions use Arts or ESD to share the sound card between applications, while most GNU/Linux musicians would want to use JACK - admittedly more complex, but far more powerful. I (Daniel) was once asked what was so difficult about JACK that means it isn't found as the primary sound server in any mainstream GNU/Linux distribution. I don't think it is difficult to use, but for the time being it still requires a patched kernel, and some knowledge of sample rates and buffers. Many non-musical users just want to be able to throw audio at any sample rate to the sound card, and could care less about real-time priority.

In addition, the creative user's default selection of applications would be very different to - for example - a sys-admin. Even gigantic distributions like Debian don't package all of the specialist tools needed for media creation, and the integration between packages is often less than perfect. So the goal of 64 Studio is to create a native x86_64 distribution with a carefully selected set of creative tools and as much integration between them as possible.

Today, we have free software applications covering many of the creative disciplines other than audio or music, including 2D and 3D graphics, video, and publishing for the web or print. Unfortunately media creation, when compared with media 'consumption', remains a niche activity, even on Linux. This niche status is reflected in the fact that none of the mainstream Linux distributions work particularly well 'out of the box' for media creation - but to be fair, Windows XP or OS X also require many additional packages to be installed before their users can realise the full creative potential of their chosen platform.

Of course specialist Linux audio distributions already exist, including AGNULA/DeMuDi, Planet CCRMA, dyne:bolic and Studio to Go!, with a good level of integration for music-making. But all of these other audio distributions are x86 only so far, and there are few specialist distributions in the other creative fields. Ratatouille, a Knoppix-based distribution designed for animators, is one exception.

Switching to native 64-bit software doesn't necessarily realise an instant and obvious improvement in performance on the same hardware, but we think that if we create a native platform, then application developers can begin to realise the benefits of 64-bit processor

optimisation and an improved memory architecture. Even in the short term, it makes more sense than building i386 binaries.

But there's a problem with specialist distributions. Since they have relatively few users, they usually end up being maintained by a single person. External project funding, whether from the state or a venture capitalist, is often unreliable in the long term, and can steer the agenda of the distribution away from that of the users.

Since we believe maintaining a niche distribution is simply too much work for a volunteer to be expected to do, we set up a company to pay developers to create and maintain the system using the Custom Debian Distribution framework. You may know of Free's work on CDD from recent releases of the AGNULA/DeMuDi distribution. Most of the packages in 64 Studio come from the Pure 64 port of Debian testing, with some from Ubuntu, some from DeMuDi and some custom built.

3 Why Debian?

A more obvious choice might be Red Hat, given that many of the high end (which is to say expensive) proprietary tools used in Hollywood studios and elsewhere are sold as binary-only Red Hat packages. However, the split between Red Hat Enterprise and Fedora Core presents serious problems for any derived distribution. On the one hand, you could rebuild Red Hat Enterprise from source as long as you removed all Red Hat trademarks, but that's a lot of extra work - and you'd have to follow Red Hat's agenda for their distribution, which you couldn't have any input to. We doubt that you'd get much goodwill from Red Hat for 'improving' their distribution either.

On the other hand, you could build a distribution on top of Fedora Core. It's broadly Red Hat compatible, and there are the beginnings of a community process taking place - although it's still far more centrally controlled than genuine grass-roots distributions. The key problem with this approach is that Fedora Core is not designed or built to actually be used. We can say this with some confidence because I (Daniel) was able to ask Michael Tiemann, former Red Hat CTO and now vice president of open source, this question myself. Fedora Core remains a technology preview for Red Hat Enterprise, and the Fedora Project has absolutely no commitment to stability or usability. If Red Hat wants to try a major update to see what breaks, it can.

Debian does have a commitment to stability, and a bona-fide community process. There are other reasons for favouring Debian over Red Hat, not least of which is the long-established support in Debian for seamless upgrades with apt-get, since on the creative desktop we'll be upgrading continuously. The work of the Debian Pure 64 port team is of a very high quality, not to mention that of all the many Debian package maintainers.

We recognise that whatever packages we put into 64 Studio, users will want some of the packages that we haven't included - so being able to use thousands of binaries straight from the Pure 64 port without modification is a major advantage. Because we're sticking very closely to Debian with the 64 Studio design, users can install any application from Pure 64 simply by enabling an additional apt source. This includes most of the well-known applications with the exception of OpenOffice.org, which just won't build natively on x86_64 yet.

In fact, 64 Studio is not so much a distribution based on Debian as a Debian remix. Free is in the process of becoming a Debian Developer, so we will be able to contribute our improvements back directly - where they are Debian Free Software Guidelines compliant. However, we do benefit from the flexibility of not being an official part of Debian. For example, the Debian project has decided that it does not want to package binary audio interface firmware, which is required to be loaded by the driver for the interface to work. That's fair enough, and we understand the reasons for their decision, but it's a major problem if you own that kind of interface, because it won't work out of the box.

This kind of hardware - effectively reprogrammable on the fly with a new firmware blob - is only going to become more common, and

not just for audio interfaces. So for the sake of our users, we have to support it. Otherwise, free software will become significantly harder to use than proprietary equivalents - and that's not a future we want to see. As end users, we couldn't modify our sound cards when they were pure hardware, so we don't think it's any worse that they now require a binary upload. At least now there is the possibility of creating our own firmware and uploading that instead, which we didn't have before.

Our first alpha release was based on Fluxbox, because this window manager places minimal demands on system resources, and is very quick to learn, since there isn't much to it. However, we have since switched to a stripped-down Gnome install. Again, this is because we don't want to make free software too difficult for people who are used to proprietary tools. This doesn't mean that we will dumb down the interface or clone the look of other platforms, but - for example - we can expect new users to assume that the program launching menu is in the lower left corner of the screen. There are also expectations about drag and drop file management, or GUI-based system configuration tools. Fluxbox is very fast, but it's an extra thing to get used to on top of everything else.

4 The business model

Since we want to pay developers to work on 64 Studio, part of making the distribution sustainable is creating a viable business model based on free software. The maintainers of the 64 Studio distribution are fundamentally in an editorial role, selecting the most appropriate software from the many thousands of packages available, and putting it into a convenient snapshot. Since the software is free software, it would be churlish of us to demand that people pay us to do this, but if we provide something of value then it should be worth a modest (and completely optional) subscription. We believe Red Hat's compulsory subscription model has cost its Enterprise distribution a lot of potential users. Apart from being ethically questionable in the context of software contributed to the distribution at zero cost, as a systems manager at a well-known studio with hundreds of Red Hat desktops put it, "Why should we have to pay for support every year whether we need it or not?"

Community support often meets or exceeds the quality that proprietary software vendors provide, but people tell us that it's reassuring to have some paid-for support available as an option. Sometimes our questions are just too ordinary to interest people on a mailing list or forum, or at the other end of the scale they can require patience and time-consuming research to answer. It can sometimes be difficult to get the help you need when you're up against a project deadline. We believe that by covering one kind of desktop user really well, we can provide detailed support for the people that need it at a reasonable cost. For the people that don't need support, or are planning large deployments where per-seat licences would be prohibitive, it's still free software - and we're not going to lock people into support contracts in order for them to access updates either.

We also offer the 64 Studio codebase as a development platform for OEMs building multimedia products on x86_64 hardware. We believe this enables these companies to reduce their development costs and time-to-market. We are considering producing a server edition of the distribution in future that would combine a fast and simple install with pre-configured services, so that a workgroup file server or a streaming media server could be set up in a few minutes - and these services would work right away with 64 Studio desktop machines of course. In the longer term, we hope that 64 Studio will go beyond packaging and integration work to contribute directly to application development, particularly where 'missing links' are identified.

5 Challenges

There are a number of challenges we still have to face. The first is following the rapid pace of kernel development. In version 0.6.0 we were using Linux 2.6.13 with Ingo Molnar's real-time pre-emption code and a few other patches. At one time these patches didn't build on x86_64 at all, and as far as we knew, we were the only native 64-bit distribution using them at the time. The

indications from our beta testing community are that this combination works really well for audio with full pre-emption enabled, the most aggressive setting. For the time being we are using the realtime-lsm framework to give real-time priorities to non-root users, because we know it works. We may switch to rlimits in the future, as this code has been merged into the mainline kernel for some time now.

Another challenge we have to deal with is the Debian community process. We are not in a position to demand anything from the Debian developers, we can only suggest and encourage. If there's a real roadblock within Debian, we have the option to create a custom package, but obviously that's something we'd rather not do.

A third challenge is the issue of support for proprietary formats within free software. At the level of encoding and decoding, we think the best solution we've seen is the GStreamer plugin collection, which as far as we can tell meets the requirements of free software licences regarding linking, and also the legal requirements of the patent holders. It's simply not sustainable to expect users to locate and download libraries of dubious legal status, and install these by themselves. Apart from any ethical problems, it's impossible to support users properly in that situation. In addition, using these libraries is likely to be out of the question for an institutional user, such as a college.

At the level of project interchange, for example moving a complex project from Ardour to ProTools, there does seem to be a move among proprietary audio applications towards support for AAF, the Advanced Authoring Format. Free software must support this kind of high-level project compatibility format, otherwise it doesn't stand a chance of gaining a significant user base in this area. When we talk to people in the music industry, it's almost a mantra that 'everyone mixes in ProTools'. We're not aware of any free software audio application that supports ProTools format import or export directly, but at least with AAF we have the chance of finding a middle way.

6 Conclusion

64 Studio is available for download as an .iso image, and the distribution is seamlessly upgradeable with apt-get of course. We'd be more than pleased to hear your test reports and suggestions for the distribution - you can help us make free software the creative desktop of choice.

7 Acknowledgements

We would like to thank all the free software developers who make building a distribution like 64 Studio possible.

8 References

64 Studio homepage
http://64studio.com/

The 64-bit x86 architecture
http://en.wikipedia.org/wiki/AMD64

Debian Pure 64 port
http://amd64.debian.net/

Advanced Authoring Format
http://aaf.sourceforge.net/

9 Appendix

Some of the packages included in 64 Studio release 0.6.0:

CD
sound-juicer
cdrdao
dvd+rw-tools
gcdmaster

Graphics
gimp
inkscape
blender
gphoto2
gtkam
gtkam-gimp
gthumb
yafray
dia
libwmf-bin
ktoon
pstoedit
sketch
imagemagick
perlmagick
xsane

Internet
gftp
bluefish
linphone
gaim
gnomemeeting

JACK
jackeq
jack-rack
jamin
meterbridge
qjackctl

Audio
alsa-base
alsa-firmware
alsa-source
alsa-tools
alsa-tools-gui
alsa-utils
flac
speex
swh-plugins
tagtool
tap-plugins
vorbis-tools
totem-gstreamer

Base
bittornado-gui
bittorrent
gnome-system-tools
ia32-libs
nautilus-cd-burner
vorbis-tools
vorbisgain

Office
abiword-gnome
abiword-help
abiword-plugins
abiword-plugins-gnome
gnumeric
gnumeric-doc
gnumeric-plugins-extra

Publishing
scribus
evince

Notation
noteedit

Recording
ardour-gtk
ardour-session-exchange
audacity
timemachine

Sequencing
hydrogen
rosegarden4
muse
seq24

Synthesis
ams
amsynth
linuxsampler
qsampler
qsynth
vkeybd

Video
kino
libtheora0
dirac

Kernel
kernel-image-2.6.13-1-multimedia-amd64-generic
realtime-lsm

footils – Using the *foo* Sound Synthesis System as an Audio Scripting Language

Martin RUMORI
Klanglabor, Academy Of Media Arts
Peter-Welter-Platz 2
D-50676 Cologne
Germany
rumori@khm.de

Abstract

foo is a versatile non-realtime sound synthesis and composition system based on the Scheme programming language (Eckel and González-Arroyo, 1994; Rumori et al., 2004; Rumori, 2005). It is mainly used for sound synthesis and algorithmic composition in an interactive *type-render-listen*-loop (the musician's *read–eval–print*-loop) or in conjunction with an editor like the inferior mode of *emacs*. Unlike with other sound synthesis languages, *foo* programs are directly executable like a shell script by use of an *interpreter directive*. *foo* therefore allows for writing powerful sound processing utilities, so called *footils*.[1]

Keywords

foo, scheme, scripting, algorithmic composition, sound utilities

1 Introduction

Scripting has played a major role in the development of computer systems since the early days. Scripting often means being a *user* and a *programmer* at the same time by accessing functions of applications or operating systems in an automated, "coded" way rather than interactively.

A major design principle of the UNIX operating system is to create several simple applications or tools which are suitable for exactly one purpose and to combine them in a flexible way to implement more complex functionalities. This is possible through the well-known UNIX concepts of pipes and file redirections. A powerful command line interpreter, the *shell*, allows for accessing these concepts both in interactive mode as well as in so called *shell scripts*. It is quite easy to generalize an interactive shell command for using it in a script and vice versa. In fact, the UNIX shell programming language has started to blur the distinction between *user* and *programmer*.

[1]Use of the term *footils* by courtesy of Frank Barknecht, see http://www.footils.org

UNIX shell scripts are often used for recurring custom tasks closely related to the operating system itself, such as system administration, maintenance and file management. Apart from that, there are many scripting languages for special purposes, such as text processing (awk, Perl). Scripts written in one of these languages can be seamlessly integrated with UNIX shell scripting by means of the so called *interpreter directive* at the beginning of a script (as documented in `execve(2)`):

```
#!/usr/bin/perl
```

Those scripts appear and behave like any other UNIX program or shell script and thus get *scriptable* itself. This property of being "recursive" makes shell scripting so powerful.

2 Scripting and computer music

2.1 Standalone applications

In the field of computer music, composers often deal with graphical standalone applications, such as Ardour or Pd, or with dynamic languages in an interactive fashion, such as SuperCollider. While these tools are very powerful in terms of harddisk recording, sound synthesis or composition (like Perl for text processing), they do not integrate in the same way with the operating system's command line interface as textual scripts (unlike Perl for text processing). In most cases, however, this is not necessary or desirable.

Pd can be launched without showing its GUI. the patch to be executed can be given at the command line, including initial messages to be sent to the patch. SuperCollider's language client, *sclang*, may be fed with code through its standard input which then is interpreted. This level of shell automation is already sufficient for tasks such as starting a live session or an interactive sound installation.

2.2 Soundfile utilities

A major task when using a computer for any task is file maintenance. This is especially true for dealing with soundfiles. Common tasks include conversion between different soundfile types, changing the sample rate or the sample format, separating and merging multichannel files, concatenating soundfiles, removing DC offset or normalizing. As for sorting or archiving text files, this kind of tasks are often applied to many soundfiles at a time and therefore should be scriptable.

In order to accomplish those tasks, a lot of command line utilities are available which fully integrate which shell scripting in the abovementioned sense. Examples for such utilities are *sndinfo* and *denoi* included in Csound (Boulanger, 2005), the tools bundled with *libsndfile* *sndfile-info*, *sndfile-play* and *sndfile-convert*, or *sndfile-resample* from *libsamplerate* (de Castro Lopo, 2006). Another example is the well known *sox* program, which also allows for some effects processing (Bagwell, 2005).

Those tools can be called from a shell script in order to apply them comfortably to a large number of files. The following *bash* script might be used to remove mains power hum from a number of soundfiles specified on the command line:

```
#!/bin/bash

SOX=sox
FREQUENCY=50 # european origin
BANDWIDTH=6
SUFFIX="_br"

while getopts ":f:b:s:" OPTION; do
  case $OPTION in
    f ) FREQUENCY=$OPTARG ;;
    b ) BANDWIDTH=$OPTARG ;;
    s ) SUFFIX=$OPTARG ;;
  esac;
done

shift $(($OPTIND - 1))

for INFILE; do
  OUTFILE=‘echo $INFILE | sed -r -e \
      "s/^(.*)(\.[^\.]*)$/\1${SUFFIX}\2/g"‘
  $SOX $INFILE $OUTFILE \
    bandreject $FREQUENCY $BANDWIDTH;
done
```

While using command line soundfile tools this way might be quite elegant, they still can only run as they are. It is not possible to directly access and manipulate the audio data itself from inside such a script.

2.3 Scriptable audio applications

Apart from that, there are operations on soundfiles which are much closer related to the artistic work with sound itself, such as filtering or effects processing of any kind, mixing, arranging or simply “composing” based on algorithms and/or underlying sound material. While requesting scripting capabilities is evident for doing file related tasks mentioned above, the latter procedures are mostly done inside single standalone applications or sound synthesis systems.

Attempts have been made to open the processing scheme of the audio data to a script interface. One approach was realized in the *Computer Audio Research Laboratory (CARL) Software Distribution* at *CRCA (CME)* since 1980 (Moore and Apel, 2005). The *CARL* system consists of several independent small UNIX programs for reading and writing soundfiles, as well as sound synthesis, effects processing and analyzing audio data. They communicate with each other via ordinary UNIX pipes. This way it is possible to generate a kind of signal processing patches as in Pd, but by means of an arbitrary shell scripting language:

```
$ fromsf infile.ircam | \
    filter bandreject.fir | \
    tosf -if -os outfile.ircam
```

This approach is quite smart, as it allows for using the UNIX command line for audio processing in the same way as for text processing. It is even possible to set up parallel processing pipes with the `para` program. since the shell language is not powerful enough for expressing those parallel pipes, this program has to use its own syntax, which unfortunetaly causes some deformation to the aesthetical integrity of the *CARL* approach.

Another approach was implemented by Kai Vehmanen in his *ecasound* application (Vehmanen, 2005). *ecasound* allows for creating flexible so called signal processing *chains*. The parameters for these chains are specified via command line options or from files containing the chain rules. Therefore *ecasound* is fully scriptable from the commandline or from inside shell scripts.

```
$ ecasound -i:infile.aiff -o:outfile.aiff \
    -efr:50,6
```

Ecasound allows for building up parallel processing chains at the same grammatical level of

the scripting language used for simpler tasks. *ecasound* does not only operate on soundfiles, but may also record, process and play audio in realtime, optionally using controller data such as MIDI.

Both the *CARL* tools and *ecasound* are examples for tools which allow for directly accessing the audio processing scheme. by calling them from inside a shell script, similar to the *sox* example above, one is able to create very sophisticated sound processing utilities.

3 Using *foo* for writing audio processing scripts

foo's approach for allowing scripting is different from the abovementioned ones. *foo* is neither a closed standalone application nor a utility especially designed for scripting. *foo* is a sound synthesis and composition environment for non-realtime use based on the dynamic, general purpose programming language Scheme.

3.1 History of *foo*

foo was developed by Gerhard Eckel and Ramón González-Arroyo in 1993 at ZKM, Karlsruhe, for the *NeXTStep* platform. The low-level *foo kernel* is written in Objective-C, while the higher level parts are written in Scheme.

Starting from 2002, *foo* was ported to the Linux platform by the author using the *GNUStep* framework (Fedor et al., 2005), a free *OpenStep* implementation. The project was registered at *SourceForge* in 2003. In 2004, *foo* was ported to Mac OS X, where it runs natively using the *Cocoa* (formerly *OpenStep*) framework.

Also in 2004, *foo* was partitioned into *libfoo*, which contains the signal processing primitives, and *elkfoo*, which consists of the interface to the Elk Scheme interpreter (Laumann and Hocevar, 2005). This should make a possible future transition to a different Scheme implementation easier. For easier packaging and cross-plattform-building, *foo* got an *autotools* build system in the same year.

3.2 Major concepts of *foo*

Them main purpose of *foo* is to provide a high quality, highly flexible sound synthesis and music composition system.

foo provides signal processing primitives written in Objective-C which can be accessed from inside the Scheme environment. Unlike CLM (Schottstaedt, 2005) or Csound, *foo* does not distinguish between *instruments* and *events (score)*. Nevertheless, it is easily possible to express a Csound-like semantics of orchestra and score with *foo*, or the concept of *behavioral abstractions* as in Nyquist.

By means of the *foo* primitives, static signal processing patches can be generated and executed. Temporal relationships are expressed in hierarchical time frames which are relative to the surrounding one.

Higher level concepts, such as envelopes or musical processes, are entirely implemented in Scheme in the *control* library by Ramón González-Arroyo, which is part of *foo*.

This openness allows for using *foo* for very different tasks: apart from algorithmic composition based on highly abstracted Scheme constructs as found in the *control* library, it is also possible to use *foo* on the *kernel* level for simple tasks like converting soundfiles, extracting or merging channels, or effects processing.

Like CLM, *foo* is usually used interactively by entering and evaluating Scheme expressions, which construct signal processing patches or render them into soundfiles. It is also common to use *foo* in conjunction with an editor, such as the *inferior*-mode of *emacs*. Since Scheme is an interpreted language (at least in the implementation used so far), *foo* programs can also made directly executable from the shell command line prompt.

This allows for writing versatile "shell" scripts which are not bound to the capabilities of a specific application like *sox*, but rather can benefit from the full power of a generic programming *and* sound synthesis language. Writing foo scripts ("footils") also differs from approaches such as *ecasound* in that there is no distinction anymore between the calling language (shell) and the audio processing language (*ecasound* chain rules).

3.3 Making *foo* scripting comfortable

Several issues had to be solved in order to make *foo* programs directly executable as scripts while not affecting the interactive use of *foo*. In the following, some of these issues are described.

3.3.1 Understanding the *interpreter directive*

According to the manpage of `execve(2)`, a script can be made executable by adding an *interpreter directive*:

> execve() executes the program pointed to by filename. filename

must be either a binary executable, or a script starting with a line of the form `#! interpreter [arg]`. In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, which will be invoked as `interpreter [arg] filename`.

Since the original *elk* executable did not accept a scheme file as a direct argument, a different startup executable for *foo* has been written. This was already done in the first version of *foo*. A *foo* script can be build by adding a line like

```
#!/usr/local/bin/foo
```

at its first line.

3.3.2 Load stages, packages and script arguments

The startup procedure of the *foo* sound synthesis program follows a multi-stage approach:

- start the interpreter executable and evaluate command line options directed to the interpreter itself (heap size, etc.)
- hand over control to the scheme interpreter by loading the toplevel scheme file, load *foo* primitives into the interpreter, evaluate scheme stage command line options, load *foo* packages
- if a script file was specified on the command line, build the command line left over for the script and execute it, else enter the interactive read-eval-print-loop

This load procedure indicates a problem which arises when specifying options to the *foo* executable:

```
$ foo --unload control
Usage: foo [options] [arguments]
...

$ foo -- --unload control
```

The first invocation of *foo* fails, because the option `--unload` is not understood by the scheme interpreter's command line parser. In order to make sure it "reaches" the scheme initialization stage, it has to be "quoted" with `--` to bypass the *elk* interpreter.

Invoking *foo* with the option `--unload control` will prevent the control library from being loaded into *foo* at startup. This might be suitable for scripts which do not need this package in order to speed up the initialization process. Therefore the interpreter directive for such a script should read:

```
#!/usr/local/bin/foo -- --unload control
```

Another problem occurs at this point: `execve(2)` apparently does not tokenize multiple initial arguments given in an interpreter directive into several arguments but passes them as a whole as one argument to the interpreter.

In order to be able to parse multiple options given in the interpreter directive, the *foo* executable contains a hack which tries to tokenize `argv[1]` into several arguments according to a certain heuristic, constructs a corrected argument vector and re-executes itself.

3.3.3 Command line parsing

foo scripts have to be able to access the part of the invoking command line following the script-file argument in order to understand script options. Command line parsing therefore is a common task in *foo* scripts.

In order to make command line parsing easier for script authors, a scheme library `cmdline` is included with *foo*. It features alternative options such as long- and shortopts, options with or without parameters, different possibilities of specifying multiple parameters to options, and automatic help message generation:

```
#!/usr/local/bin/foo -- --unload control

(require 'cmdline)

(let*
  ;; equiv-opts-list | mandatory? | \
                  with-params? | help-string
  ((option-list
    '((("--help" "-h") #f #f "this help screen")
      (("--outfile" "-o") #t #t "output file")
      (("--type" "-t") #f #t "file type")
      (("--sformat" "-s") #f #t "sample format")
      (("--srate" "-r") #f #t "sample rate")
      (("--channels" "-c") #f #t "channels")))
   ;; show help message
   (help
    (lambda ()
      (format #t "~a: script foo~%"
        (car (foo:script-args)))
      (format #t "usage:~%")
      (format #t "~a~%"
        (cmdline:help-message option-list))
      (exit))))

  ;; help requested?
  (if (cmdline:option-given?
        (foo:script-args) option-list "--help")
```

```
      (help))

  ;; commandline valid?
  (if (not (cmdline:cmdline-valid?
         (foo:script-args) option-list #t))
      (help)))
```

This script will produce the following output when invoked with no arguments:

```
$ ./script.foo
(cmdline:validate) \
  mandatory option missing: --outfile
./script.foo: script foo
usage:
   --help, -h             this help screen
   --outfile, -o <args>   output file
   --type, -t <args>      file type
   --sformat, -s <args>   sample format
   --srate, -r <args>     sample rate
   --channels, -c <args>  channels
multiple <args>: --opt <arg1> --opt <arg2> \
              or --opt <arg1,arg2,...>
```

Other functions of the `commandline` scheme library not shown in this example include reading the parameter lists of specific options or getting the remaining arguments of the command line, e. g. file arguments.

3.3.4 Interacting with the outer world: `stdin` and `stdout`

Reading from standard input and writing to standard output from *foo* scripts is important if executed inside a pipe. Imagine calling a *foo* script like this:

```
$ find . -name '*.wav' | script.foo
```

This will mean reading a file list from standard input, which can be accomplished with standard scheme functions. The following code reads the files from standard input into the list `files` and the number of files into `num-files`:

```
(let*
  ((files
    (do ((file-list '()
                    (cons last-read file-list))
         (last-read))
        ((begin
           (set! last-read (read-string))
           (eof-object? last-read))
         (reverse file-list))))
   (num-files (length files)))

  ...)
```

Writing to standard output is done similarly through standard scheme functions.

3.4 *footils*

footils is a collection of *foo* scripts written so far by Gerhard Eckel and the author. The aim of *footils* is to provide a set of powerful and robust scripts for the musician's everyday use. *footils* currently consists of the following scripts:

fsconvert convert soundfiles

fssrconv do a samplerate conversion on soundfiles

fsextract extract channels from multichannel files

fsfold fold a soundfile over itself and normalize

fskilldc remove DC from soundfiles

fsmono2stereo create stereo file from mono file

fsquadro2stereo create stereo file from quadro file

fsnorm normalize soundfiles

fsregion extract a temporal region from soundfiles

fsreverse reverse soundfiles in time

fstranspose transpose soundfiles

fscat concatenate soundfiles

These scripts are currently refactored and integrated with the *foo* distribution.

4 Conclusions

Several issues of scripting in the field of computer music have been considered. It turned out that with most current software it is not possible to write scripts which are seamlessly accessible from the UNIX command line and at the same may benefit from the power of a fully featured programming and sound synthesis language.

It has been shown that writing scripts with *foo* might be able to close this existing gap.

5 Acknowledgements

I would like to thank Gerhard Eckel and Ramón González-Arroyo for developing *foo* and for starting the development of scripts for the *foo* system.

Many thanks to Frank Barknecht for his kind permission to use the term *footils* for the *foo* script collection.

References

Chris Bagwell. 2005. Homepage of sox. *http://sox.sourceforge.net*.

Richard Boulanger. 2005. Official homepage of csound. *http://www.csounds.com*.

Erik de Castro Lopo. 2006. Homepage of libsndfile, libsamplerate etc. *http://www.mega-nerd.com*.

Gerhard Eckel and Ramón González-Arroyo. 1994. Musically salient control abstractions for sound sound synthesis. *Proceedings of the 1994 International Computer Music Conference*.

Adam Fedor et al. 2005. The official gnustep website. *http://ww.gnustep.org*.

Oliver Laumann and Sam Hocevar. 2005. Homepage of Elk Scheme. *http://sam.zoy.org/projects/elk/*.

F. Richard Moore and Ted Apel. 2005. Homepage of CARL software. *http://www.crca.ucsd.edu/cmusic/*.

Martin Rumori, Gerhard Eckel, and Ramón González-Arroyo. 2004. Once again text and parentheses – sound synthesis with foo. *Proceedings of the 2004 International Linux Audio Conference*.

Martin Rumori. 2005. Homepage of foo sound synthesis. *http://foo.sf.net*.

Bill Schottstaedt. 2005. The clm home page. *http://ccrma.stanford.edu/software/clm*.

Kai Vehmanen. 2005. Homepage of ecasound. *http://www.eca.cx/ecasound/*.

Ontological Processing of Sound Resources

Jürgen Reuter
Karlsruhe, Germany,
http://www.ipd.uka.de/~reuter/

Abstract

Modern music production systems provide a plethora of sound resources, e.g. hundreds or thousands of sound patches on a synthesizer. The more the number of available sounds grows, the more difficult it becomes for the user to find the desired sound resource for a particular purpose, thus demanding for advanced retrieval techniques based on sound classification. This paper gives a short survey of existing approaches on classification and retrieval of sound resources, discusses them and presents an advanced approach based on ontological knowledge processing.

Keywords

classification of sounds, sound resource lookup, ontologies, OWL

1 Introduction

State-of-the-art music production systems consist of a computer-centered, heterogeneous network of hardware and software modules with typically huge banks of sound resources. Modern hardware synthesizers or tone modules often have banks with hundreds or thousands of different sounds. Producing electronic music therefore means to select among synthesizer or tone modules, as well as to select sounds from each module. Modules not only (if at all) provide factory presettings, but typically also reserve much memory space for lots of user patches that may be tweaked manually or loaded via MIDI, thereby even increasing the number of available sounds. The music producer's task of selecting a sound thus becomes an increasingly complex challenge.

For example, imagine a composer who has already in mind a vague idea of the electronic sounds that should be used for a new piece of music. In an older piece a couple of years back in time, there was a bass line with a bass sound that also should fit well for the new piece. But what synthesizer was used to produce this sound? Even if the synthesizer is known, which one of the hundreds or thousands of sound patches was used? If it was a user patch, where was the patch stored? Even if the sound can be located, over which specific MIDI port and channel can the sound be addressed? Unfortunately, on many synthesizers, sound patches are ordered in a fairly chaotic fashion, especially, if they do not fit into the GM sound map. In the worst case, the composer has to scan through thousands of sound patches to find the desired one. What is required, is the possibility to search for a particular sound.

Searching for a file in a file system is conceptually fairly straight forward, given the name of the file (or part of it), or the file type, or some content that is known to appear in the file. In contrast, searching for a sound is much more challenging. First of all, while all files in a file system can be iteratively accessed by browsing through the directory hierarchy, there is, as of this writing, no central registry for all sound resources that are available on the system. Rather, every synthesizer has its own approach of managing sound patches. Secondly, while files can be searched for by their name or type or content, defining useful search criteria for sounds is difficult. Finally, searching for near matches means to have a system that allows for defining proper metrics of sound comparison.

In the above example of looking for a bass sound, listing all available bass sounds would already fairly reduce the number of sound resources that have to be further checked. If the bass sound can be qualified even more specific, the search could be even more effective. In this article, we examine and discuss multiple approaches for classifying and describing sounds. We present a prototype design and implementation of a sound classification and description framework based upon ontological technology. We show how this framework enables us to search for specific sounds. Finally, we discuss the impact of further pursuing this approach on

Linux audio development.

1.1 Preliminaries

There is a mismatch between the classical, rather mathematical notion of the term sound and the common conception of sound as viewed by most musicians and music listeners. While the classical definition solely focuses on the core wave shape of a periodic signal, most people perceive aperiodic characteristics also as part of a sound. Among such aperiodic characteristics are vibrato, noise content, reverb or echo content, but also irregularities of the harmonic spectrum such as non-equidistant partials or partials that vary in pitch or amplitude. For the remainder of this article, we therefore extend the classical definition by also incorporating such aperiodic properties into the notion of sound.

1.2 Paper Outline

We start with a short survey of how various systems currently address, if at all, the sound selection problem (Section 2). Then we discuss the approaches in order to reveal commonly used strategies (Section 3). Based upon this discussion, we develop an ontological framework in order to solve the sound selection problem in a unified way (Section 4). We demonstrate the usefulness of our system by giving some examples of how the user can benefit from the system (Section 5). The work presented here has a significant impact on Linux audio development in general and on construction of software synthesizers in particular, which is left for further investigation (Section 6). We complete our journey with a concluding summary of our results (Section 7).

2 Related Work

We give a short survey on the history of sound classification, from acoustic instrument taxonomies and organ disposition over grouped categories in the MIDI standard to what recent synthesizers provide. This survey is not at all meant to be complete, but establishes some central ideas for sound classification that we will discuss and further develop in the subsequent sections.

2.1 Instrument Taxonomies

Classification of acoustic instruments has a long tradition. Figure 1 shows an example taxonomy of selected acoustic instruments as it can be found in this or similar form in standard music literature. Note that such taxonomies are typically based on how an instrument technically *works* rather than how it *sounds*. Still, if two instruments work similarly, they often sound similarly. Eventually, however, a small change in construction may result in a tremendous change in sound.

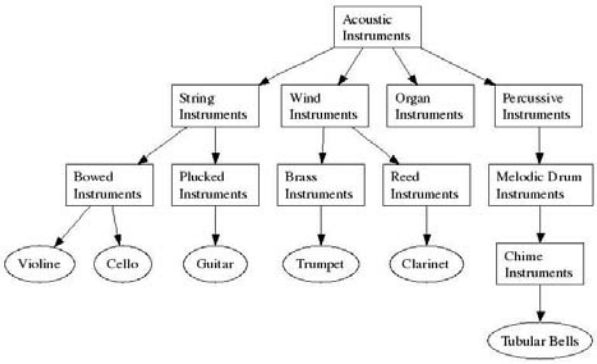

Figure 1: A Taxonomy of Selected Acoustic Instruments

Also note, that, traditionally, the realization of the sound source of the instrument is more important for classification than e.g. that of the body. For example, a saxophone has a reed mouthpiece and therefore is considered to be a reed instrument regardless of its metallic body, while the so-called Indonesian trumpet is blown like a trumpet and therefore considered as brass instrument, regardless of its wooden body.

2.1.1 Dispositional Approach

The situation is slightly different for the (acoustic or electric) organ, which has the ambition of uniting many instruments (organ registers) into a single apparatus. While, at least for the acoustic organ, there is also a technical classification of pipes based on how they *work* (e.g. labial or stopped pipes), organ registers are often named after well-known acoustic instruments (e.g. flute, trumpet, saxophone), i.e. how they *sound*. Indeed, the organ's naming of registers is maybe the oldest example for a categorization of sounds: it assists the organ player in looking up a sound. This is especially important since each organ has an individual, more or less rich set of sounds, and that way, a guest organ player can quickly get familiar with a foreign organ. Remarkably, already Supper(Supper, 1950) notes that the rules, which underly the disposition of an organ, are of hierarchical kind. We will resume this idea, when presenting a framework for describing and look-

ing up sounds (cp. Section 4).

2.2 Grouping

The instrument patch map of the General MIDI (GM) Level 1 Standard(MIDI Manufacturers Association, 2005) defines 128 instruments that are partitioned into sixteen categories (cp. Fig. 2).

Program	Family
1-8	Piano
9-16	Chromatic Percussion
17-24	Organ
25-32	Guitar
33-40	Bass
41-48	Strings
49-56	Ensemble
57-64	Brass
65-72	Reed
73-80	Pipe
81-88	Synth Lead
89-96	Synth Pad
97-104	Synth Effects
105-112	Ethnic
113-120	Percussive
121-128	Sound Effects

Figure 2: GM Level 1 Sound Categories

Originally, the motivation for specifying an instrument patch map was driven by the observation that a MIDI file which was produced on some MIDI device sounded totally different when reproduced on a different MIDI device because of incompatible mappings from MIDI program numbers to sounds. Therefore, in the early days, MIDI files could not be easily exchanged without patching program change commands. Hence, the main purpose of the GM instrument patch map was to specify a fixed mapping from MIDI program numbers to sounds. Given the existing MIDI devices of the time when the GM standard was created, a set of 128 prototype sounds, so-called instruments, was specified and assigned to the 128 MIDI program numbers. A GM compatible device has accordingly to provide sounds that match these prototype sounds. Still, the GM standard explicitly leaves the specification of prototype sounds fuzzy and thus encourages device implementors to take advantage of space for variations of an actual MIDI device. Hence, when playing a MIDI file among different GM compatible devices, there will be typically an audible difference in quality or style, but the overall impression of the performance is expected to remain.

The GM instrument patch map specifies prototype sounds that were popular on mainstream MIDI devices at that time. Remarkably, most sounds in the map represent acoustic or electroacoustic instruments as used in classical or popular music. They are grouped roughly following the classical taxonomies of instruments (cf. Section 2.1).

Only the four categories *Synth Lead*, *Synth Pad*, *Synth Effects* and *Sound Effects* contain a collection of sounds that allude to specific sounds that had evolved in electronic music and were widely used since then. The former two allude to a qualitative categorization (*lead*, *pad*), while the latter two (*effects*) allude to the intended purpose of use.

Due to the extraordinary relevance of drum sounds in temporary music, the GM standard also defines a drum map that assigns basically fixed-pitch drum sounds to MIDI pitch numbers. Having a fixed pitch (unless pitch-bended or otherwise tuned), drums constitute a separate category of their own. Within this category of drums, however, there is no further categorization perceivable, except, maybe, that those drums that represent a standard drummer's hardware are grouped together in the lower part of the map, while Latin percussion and ethnic drums are mostly assigned to upper pitches. In this sense, the drum map itself maybe considered to be ordered according to the style (i.e. intended use or purpose) of the drums.

2.3 Banking

More recent MIDI devices break the limit of 128 program numbers by introducing *sound banks*: with the bank select MSB/LSB controller channel messages, a MIDI channel can be directed to switch to a different bank of sounds. In order to remain GM compatible, each bank should itself conform to the GM instrument patch map, but may provide a different style of sounds (e.g. "bright", "resonant", "slow", "fast decay"). Unfortunately, some manufacturers added this way also such sounds, that do not really fit to the GM instrument patch map. (Not only) therefore, the GM Level 1 Guidelines(Lehrman and Massey, 1998) discourage the use of banks at all on GM Level 1 compatible devices. We put on record that adding new sounds to an existing system of fixed categories may lead to difficulties.

2.4 Tagging

The Virus TI synthesizer(Access Music, 2004) has a function for tagging each sound patch with up to two values of a predefined set of 21 group identifier. These identifiers are:

Acid Arpeggiator Bass Classic Decay
Digital Drums EFX FM Input Lead
Organ Pad Percussion Piano Pluck
String Vocoder
Favorites 1 Favorites 2 Favorites 3

Figure 3: Supported Tags of the Virus TI

By tagging a sound with one of these identifiers, the sound is declared to be a member of a respective group of sounds. Interestingly, if we look at the group names, we can identify exactly the same categorization principles that we already met before:

- Identifiers like *Acid*, *Bass*, *Classic*, *Digital*, *Drums*, *Lead*, *Organ*, *Pad*, *Percussion*, *Piano*, *Pluck* and *String* suggest groups based upon similarity to sounds that the user is assumed to already know. Note that some of the identifiers such as *Drums* or *Percussion* denote a rather wide field of sounds.
- The identifier *EFX* (for sound effects) presumably denotes a group of sounds classified by its typical purpose (namely a sound effect rather than e.g. a musical instrument).
- Identifiers such as *Arpeggiator*, *Decay*, *FM*, *Input* and *Vocoder* allude to how the sound is created.
- The three *Favorites* groups finally can be considered as generic groups for which the user individually specifies the exact semantics.

2.5 Parametric Categorization

State-of-the-art sound hardware provides sets of parameters that are used to define sound patches by mainly specifying how to create the sound. This approach suggests to categorize sounds based on the values of such parameter sets. However, the size and structure of the parameter sets differs widely across particular devices.

The MIDI specification defines a few controllers for controlling e.g. vibrato, envelope and a selected set of filters. Most of these controllers have post-processing characteristics, which is of interest in particular for sample-based tone generators. In contrast, the parameter sets of synthesizers are typically much bigger and broader than those of tone generators, since they affect also the core generation of sound. For example, synthesizers often provide complex networks of oscillators, filters, and controllers with numerous possibilities of parameterization. Unfortunately, most synthesizers have sound parameters that are specific for each device individually. Even worse, a categorization based on large and complex parameter sets makes the categorization itself complex.

Due to the plethora of existing methods of synthesis, it seems doubtful that there will ever be agreement on a comprehensive standard set of sound parameters. Yet, more recent scientific work suggests new parameters that look like candidates for standardization. Maier et. al.(Maier et al., 2005) characterize sounds by quantitative properties that can be directly computed from the acoustic signal. To describe sounds, they compute for example the amount of disharmonic spectrum peaks. Conceptually somewhat related is the observation of Nasca(Nasca, 2005), that in his software synthesizer ZynAddSubFX, controlling the bandwidth of each harmonic offers a powerful approach to create realistic, warm sounds. Observations like those of Maier and Nasca suggest that such parameters are good candidates for providing a proper model of the way sounds are perceived by human beings.

3 Discussion

From the survey in the previous section, we may conclude the following observations:

Sounds may be categorized by

- their similarity to a prior known set of sounds. This approach complies with a composer's way of thinking, if the composer qualitatively specifies sounds (e.g. a soft, bright, crystal sound).
- their purpose of use. This approach complies with a producer's way of thinking if the producer has a targeted application of sound (e.g. a phone ring).
- the way they are created. This approach complies with a sound engineer's way of thinking when creating a new sound patch with his or her favorite synthesizer (e.g. a square wave vibrato modulated sawtooth sound with flanger).

Regarding the structure of categorization, we may note:

- Categories may have a hierarchical structure, thus creating a taxonomy.
- It is difficult to specify orthogonal categories. That means, in general a sound may be a member of multiple categories.
- Since there are most likely always sounds remaining that do not fall into any existing category, it is useful to have generic categories to be specified by the user that capture the remaining sounds.

The Virus' tagging approach may be used to associate a sound to (at most two) categories. However, tagging does not at all consider categories as a hierarchy, unless we support *deductive* tags: Assume, that we consider all drum sounds to be percussive. Then, if a sound is tagged "drum", it should implicitly also be tagged "percussive". This way, we may specify a hierarchy of tags. The hierarchical taxonomy of acoustic instruments is a good candidate for creating a hierarchy of tags.

4 The Sound Resources Ontology

Similar to the Virus TI, we follow the approach of tagging sounds with tags that aim to characterize qualitative attributes of the sound. For a tagging-only description and looking up of sounds, a simple relational database approach is sufficient. However, we would like to group sounds in a hierarchical manner and potentially give tags a deductive semantics as described in the previous section. Therefore, we prefer a framework with deductive capabilities based on ontological technologies.

4.1 OWL Knowledge Bases

Ontologies are an appropriate means for describing hierarchical structures of *classes of individuals* (also called *concepts*) in a flexible way, based on description logic. The *Web Ontology Language OWL*(Miller and Hendler, 2004) with its three sub-languages *OWL-Full*, *OWL-DL* and *OWL-Lite* has emerged as the maybe most important standard for description logic languages. For the remainder of this article, we consider OWL-DL, which specifies description logic semantics that is a decidable fragment of first-order logic. In contrast to *rule-based logic* such as Prolog or Datalog, the *description logic* of OWL-DL focuses on features such as hierarchical concepts, *properties* (i.e. binary relations between pairs of individuals or an individual and a data value), and property and cardinality *restrictions*. OWL can be expressed in XML-based *RDF*(Miller et al., 2004) syntax, which we use as source file format. The entire ontological description, regardless whether stored in memory or on disk, and regardless in which language specified, is usually referred to as the so-called *knowledge base*. Similar to a database, a knowledge base typically may be updated, and its current content may be queried (cp. Fig. 4).

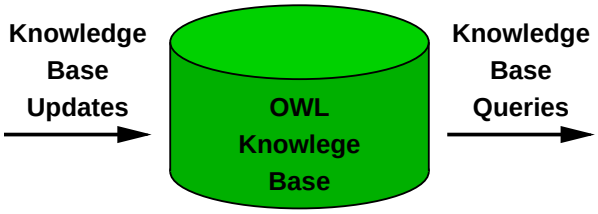

Figure 4: Updating and Querying an OWL Knowledge Base

The initial knowledge base is created from a file specified in RDF. The current version is available at: `http://www.ipd.uka.de/~reuter/ontologies/lad/sound/sound-registry.owl`

4.2 Ontology Design

Following the discussion in Section 3, our ontology contains a concept `Sound` that serves as common super class for all particular sounds. Respecting the categories of GM Level 1 devices, our ontology defines a subclass `GMSound` that disjointly divides into the 16 GM categories, each represented by a concept of its own. At the same time, `GMSound` also divides into (generally overlapping) groups that correspond to the different `SoundQuality` individuals. Each `SoundQuality` individual represents a tag of those in Fig. 3 or of a few others, that have deliberately been added, inspired by the GM categories. That way, we basically have two independent hierarchies of sounds, thus giving the user more choices in querying or browsing for a particular sound. The ontology also features a concept `SoundResource`. Each individual of this class represents a resource that hosts `Sound` individuals. An example for a `SoundResource` individual is a particular MIDI synthesizer. The ontology also models a `SoundPort` concept with the subclass `ALSAPort` such that for each `SoundResource` in-

dividual, a port can be looked up in order to access the resource. A `SoundSubscriber` finally may allocate any number of `Sound` individuals, such that the number of available sounds left can be tracked. Property constraints are deployed to bind GM sounds to MIDI program numbers.

5 Evaluation

To demonstrate the usefulness of our approach, we walk through a short sample tour on exploring the space of sounds, using *The Protégé Ontology Editor and Knowledge Acquisition System*(Crubézy et al., 2005) for visualization (cp. Fig. 5). This free, open source application from Stanford University provides a graphical user interface for viewing and editing ontologies. Note that there are a lot of OWL related tools on the net(Miller and Hendler, 2004); in this section, we just use Protégé for illustration purposes. One could also take some OWL reasoner with API, for example Jena (Hewlett-Packard Development Company, LP, 2005), and develop appropriate command line tools or dedicated interactive applications for exploring the space of sounds. However, in this section we chose Protégé for the purpose of illustrative visualization.

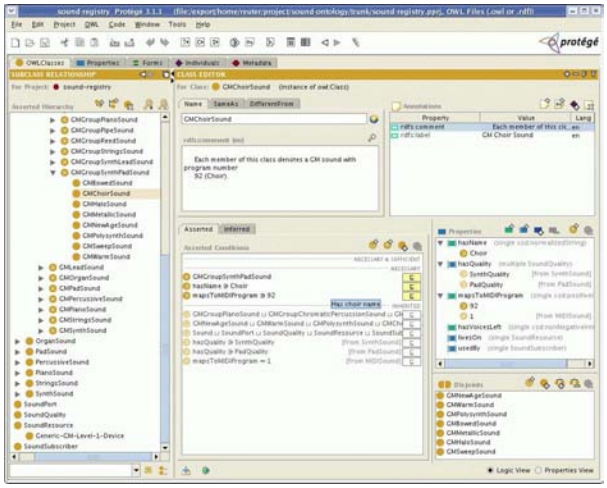

Figure 5: The Sound Registry Ontology viewed with Protégé

5.1 Querying for a Sound

We start querying for a sound by specifying properties that the sound must fulfill. In the illustrated example (Fig. 6), we ask for a sound that fulfills the two sound qualities "Synth" and "Bass" and, additionally, lives on the "MU-50" sound resource. Note that properties have a well-defined domain and range, such that Protégé lets us select e.g. the sound quality only from the actual list of available sound qualities (rather than accepting any arbitrary individual or data value).

Figure 6: Querying for a Sound on the MU-50 with "Synth" and "Bass" Qualities

Protégé returns a result consisting of three sounds that match our constraints (Fig. 7). We find the "Generic-GM-BassAndLead" sound most interesting and double-click on it to find out more about it.

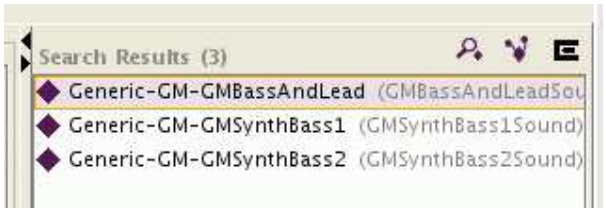

Figure 7: The Search Results

A proper window pops up (cp. Fig. 8). In this window we can see that the sound indeed fulfills our three constraints. Moreover, we learn to know that this sound maps to MIDI program 88. Imagine that we were not using Protégé, but a dedicated application embedded e.g. into a sequencer software; then the software could exploit the MIDI channel value to, for example, set the MIDI program of the currently selected track. We also notice the `rdfs:comment` field with a detailed annotation regarding this sound. Finally, in the field `hasQuality`, we can see, that this sound not only fulfills the qualities "Synth" and "Bass" as required in our query, but also the quality "Lead". In order to look, what this quality means, we double-click on "LeadQuality".

Again, a proper window pops up (cp. Fig. 9). This window shows a description of the "Lead" quality in the field `rdfs:comment`, such that we learn to know even more characteristics of the sound than what we actually required in our query.

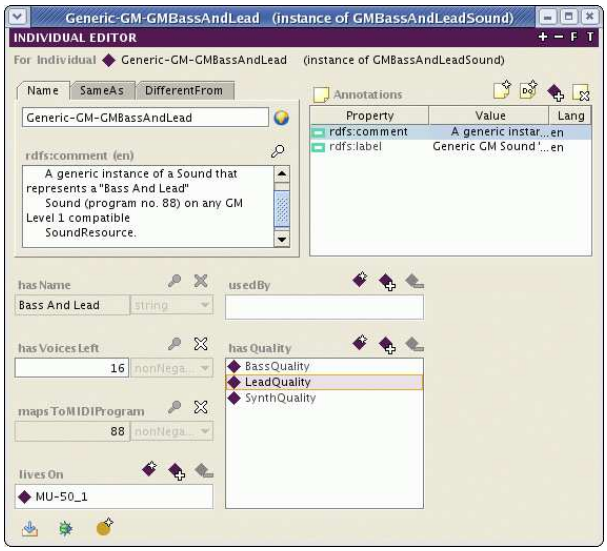

Figure 8: Viewing the Generic GM "Bass And Lead" Sound Properties

Figure 9: Viewing the "Lead" Sound Quality Properties

5.2 Browsing the Hierarchy of Sounds

Searching for sounds is also possible by just browsing through the concept hierarchy. Protégé displays the concept hierarchy as a tree that can be expanded or collapsed at your choice (cp. Fig. 10). Note that, due to possible multiple inheritance of concepts in OWL, a concept may appear multiple times in the tree. For example, the `GMOrganSound` concept appears two times in the tree, once as subclass of `GMSound`, and another time as subclass of `OrganSound`. Individuals of the concept `Sound` appear on the leaf nodes of the tree and can be viewed in more detail when selecting the appropriate leaf.

6 Future Work

Our prototype ontology focuses on the description of a generic GM Level 1 device as an example sound resource. While we provide a general ontological framework for virtually any kind of sound resource, we currently do not provide a

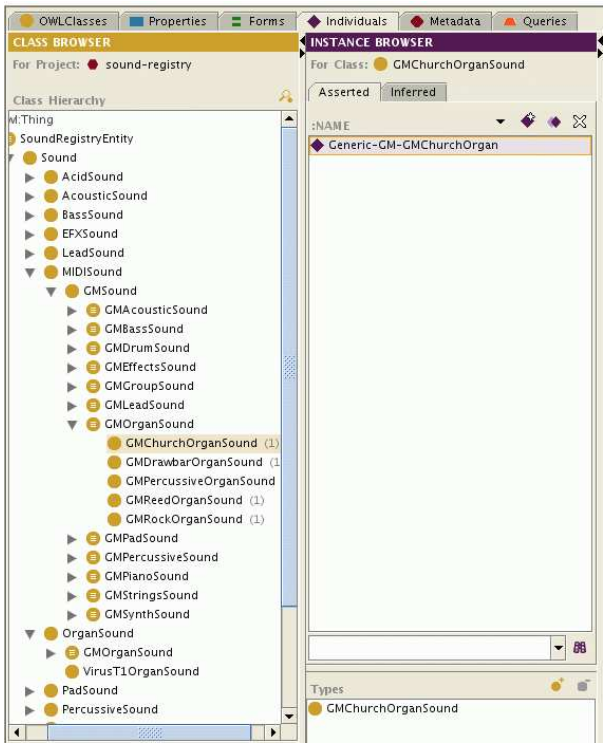

Figure 10: Browsing through the Concept Hierarchy of Sounds

description of any specific sound resource. The task of providing ontological descriptions for individual sound resources remains open for discussion. After all, ontological descriptions are desired for both, external hardware synthesizers as well as software synthesizers running under Linux. This work is in particular meant to initiate discussion on and fostering the development of proper standards.

6.1 Impact on ALSA Developers

The ontological description should be accessible to virtually all users of the audio infrastructure. Since ALSA(Jaroslav Kysela et al., 2006) has established as the default sound architecture on most current Linux distributions, responsibility for provision and maintenance of the ontological description as well as a for providing a query and management API should probably fall to ALSA. ALSA developers may want to develop and establish a proper infrastructure and API. In fact, ALSA developers could try to standardize the ontological framework as well as the query and management API in an interoperable manner. Ontological descriptions could then be provided independently of a particular operating system. This way, ALSA developers could pave the way for manufacturers of

external hardware starting to provide ontological descriptions of their hardware's sound resources by themselves. All operating systems would greatly benefit from such a development.

6.2 Impact on Linux Audio Application Developers

Just like external hardware providers, developers of software synthesizers under Linux should provide ontological descriptions of their synthesizers' sound resources, following the standards to be provided by ALSA developers.

Editing sound patches typically will affect the ontological description. For example, given a software synthesizer that lets the user create new sounds, the software could enable the user to describe the sound with tags, e.g. by displaying check-boxes or a multiple selection list with items for each tag. ALSA developers may want to standardize a default set of tags. Given such tags and other user settings, the software synthesizer should be capable of generating on the fly a corresponding ontological description of the sound.

If Linux audio developers feel that it is of too much burden for software synthesizers to create ontologies, ALSA developers may alternatively develop a sound resource query API, that each software synthesizer should implement. The ontological description of all software synthesizers could then be created and managed completely by ALSA.

7 Conclusion

We showed that in complex systems with a plethora of sound resources, characterizing sounds e.g. by classification is an essential task in order to efficiently look up a particular sound. Our historical survey on sound classification elucidated the importance of this task.

We have demonstrated the feasibility of deploying ontological technology for describing and looking up sounds and sound resources. We developed a terminological knowledge base that serves as an ontological framework, and we created a generic GM Level 1 device as facts knowledge that serves as an example on how to use our framework.

While we focus on the technical realization of the OWL-DL based framework, so far we leave open how to integrate this framework into the operating system. If Linux audio developers feel that looking up sounds and sound resources is worth being solved in a uniform way under Linux, further discussion on the integration with applications, ALSA and maybe other parts of the operating system will be required.

References

Access Music. 2004. Virus TI Totally Integrated Synthesizer. URL: http://www.access-music.de/events/11-2004/virusti_basics.php4.

Monica Crubézy, Mark Musen, Natasha Noy, and Timothy Redmond et.al. 2005. The Protégé Ontology Editor and Knowledge Acquisition System, August. URL: http://protege.stanford.edu/.

Hewlett-Packard Development Company, LP. 2005. Jena Semantic Web Framework, October. URL: http://jena.sourceforge.net/.

Jaroslav Kysela et al. 2006. Advanced Linux Sound Architecture – ALSA. URL: http://www.alsa-project.org/.

Paul D. Lehrman and Howard Massey. 1998. *General MIDI System Level 1 Developer Guidelines*. MIDI Manufacturers Association, Los Angeles, CA, USA, July. URL: http://www.midi.org/about-midi/gm/gmguide2.pdf.

Hans-Christof Maier, Franz Bachmann, Michael Bernhard, Geri Bollinger, and Adrian Brown. 2005. Prisma - music. URL: http://www.prisma-music.ch/.

MIDI Manufacturers Association. 2005. MIDI Specifications. URL: http://www.midi.org/about-midi/specshome.shtml.

Eric Miller and Jim Hendler. 2004. Web Ontology Language (OWL), February. URL: http://www.w3.org/2004/OWL/.

Eric Miller, Ralph Swick, and Dan Brickley. 2004. Resource Description Framework (RDF), February. URL: http://www.w3.org/RDF/.

Paul O. Nasca. 2005. Zynaddsubfx – an open source software synthesizer. In *Proceedings of the 3rd International Linux Audio Conference (LAC2005)*, pages 131–135, Karlsruhe, Germany, April. Zentrum für Kunst und Medientechnologie (ZKM). URL: http://lac.zkm.de/2005/papers/lac2005_proceedings.pdf.

Walter Supper. 1950. *Die Orgeldisposition*. Bärenreiter-Verlag, Kassel und Basel.