Partners and Sponsors

NUI MAYNOOTH
Ollscoil na hÉireann Má Nuad

Linux Weekly News
Sonokinetic
Linuxaudio.org
CiTu
Welcome everyone to LAC 2011 in Maynooth!

This year's conference offers yet again a sample of all things FLOSS and GNU/Linux. The ninth edition of the conference features twenty-five paper presentations, including a special session on Music Programming Languages and a keynote by Fons Adriaensen, who has been a strong supporter and participant of the conference since its early days at Karlsruhe. Since the beginnings as small developer's meeting, the conference has been extended to include tutorials, installations, concerts, club nights, all of which promotes a different side of libre audio software.

As the main organiser for this year's edition, I would like to thank my team for supporting the effort so well. In special, I would like to thank Robin Gareus, without whom we would definitely not have been able to put this year's show on the road; Frank Neumann, for organising all the paper submission process and peer-review, as well as advising on general aspects of the conference; Gordon Delap, John Lato and Eoin Smith, from the NUIM Music Department, for helping out in various organisational tasks. I would also like to thank the Research Support Office at the University for helping to search and secure external funding for the event. Many thanks also to all the presenters, in special the invited speakers, Yann Orlarey, John ffitch, Johannes Zmölnig, Vesa Norilo, Tim Blechmann and our keynote, Fons Adriaensen. Finally, the conference would not really work if it was not for the presence of such a wide group of participants, from various places around the world. We would like to thank everyone for making the effort to come to and participate in this year's event.

On a sadder note, I would like to note the passing away of Max Mathews, the father of Computer Music. Without Max's efforts much of the work that is celebrated at the LAC would not exist. These proceedings are dedicated to his memory.

We hope you have a pleasant stay in Maynooth!

Victor Lazzarini
Conference Organization
Victor Lazzarini
Robin Gareus

Live Streaming
John Lato
Jörn Nettingsmeier

Conference Design and Website
Robin Gareus

Concert Organisation and Sound
John Lato
Gordon Delap

Club Night
Eoin Smith

Paper Administration and Proceedings
Frank Neumann

Many Thanks to
Fons Adriaensen
Yann Orlarey
John ffitch
Johannes Zmölnig
Vesa Norilo
Tim Blechmann
The LAC team

...and to everyone else who helped in numerous places after the editorial deadline of this publication.
Review Committee

Fons Adriaensen  Kokkinizita.linuxaudio.org
Frank Barknecht  Footils.org, Cologne, Germany
Ico Bukvic  Virginia Tech, Blacksburg, VA, United States
Götz Dipper  ZKM/Zentrum für Kunst und Medientechnologie, Karlsruhe, Germany
Robin Gareus  Linuxaudio.org
Jari Kleimola  Aalto University, Finland
John Lato  National University of Ireland, Maynooth, Ireland
Victor Lazzarini  National University of Ireland, Maynooth, Ireland
Frank Neumann  Harman International, Karlsbad, Germany
Jörn Nettingsmeier  StackingDwarves.net, Germany
Marcelo Queiroz  University of São Paulo, Brazil
Martin Rumori  Institute of Electronic Music and Acoustics, Graz, Austria
Sampo Savolainen  Finland

Music Jury

Gordon Delap
Jesse Ronneau
John Lato
Victor Lazzarini
# Table of Contents

- **Two Recent Extensions to the FAUST Compiler**
  Karim Barkati, Dominique Fober, Stephane Letz, Yann Orlarey  
  p. 1

- **Introducing Kronos - A Novel Approach to Signal Processing Languages**
  Vesa Norilo  
  p. 9

- **Running Csound in Parallel**
  John ffitch  
  p. 17

- **New Clothes for Pure Data**
  IOhannes Zmöl nig  
  p. 23

- **Semantic Aspects of Parallelism for SuperCollider**
  Tim Blechmann  
  p. 29

- **Mimesis and Shaping**
  Joachim Heintz  
  p. 33

- **Particle synthesis - a unified model for granular synthesis**
  Øyvind Brandtsegg, Sigurd Saue, Thom Johansen  
  p. 39

- **Audio Plugin development with Cabbage**
  Rory Walsh  
  p. 47

- **Poing Impératif: Compiling Imperative and Object Oriented Code to Faust**
  Kjetil Matheussen  
  p. 55

- **An LLVM-based Signal-Processing-Compiler embedded in Haskell**
  Henning Thielemann  
  p. 61

- **OpenDAW - a reference platform for GNU/Linux Audio**
  Daniel James  
  p. 69

- **PulseAudio on Mac OS X**
  Daniel Mack  
  p. 75

- **Airtime: Scheduled Audio Playout for Broadcast Radio**
  Paul Baranowski  
  p. 81

- **A position sensing midi drum interface**
  Andrew Baxter  
  p. 85

- **Towards more effective mapping strategies for digital musical instruments**
  Patrick MyGlynn  
  p. 93

- **An LLVM bitcode interface between Pure and Faust**
  Albert Grä f  
  p. 99

- **Python for Audio Signal Processing**
  John Glover, Victor Lazzarini, Joseph Timoney  
  p. 107

- **An interface for Realtime Music Using Interpreted Haskell**
  Renick Bell  
  p. 115
<table>
<thead>
<tr>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>FluidSynth real-time and thread safety challenges</td>
<td>123</td>
</tr>
<tr>
<td>David Henningsson</td>
<td></td>
</tr>
<tr>
<td>Web-enabled Audio-synthesis Development Environment</td>
<td>129</td>
</tr>
<tr>
<td>Rory Walsh, Conor Dempsey</td>
<td></td>
</tr>
<tr>
<td>Low-Latency Audio on Linux by Means of Real-Time Scheduling</td>
<td>135</td>
</tr>
<tr>
<td>Tommaso Cucinotta, Dario Faggioli, Giacomo Bagnoli</td>
<td></td>
</tr>
<tr>
<td>Loudness measurement according to EBU R-128</td>
<td>143</td>
</tr>
<tr>
<td>Fons Adriaensen</td>
<td></td>
</tr>
<tr>
<td>Medusa - A Distributed Sound Environment</td>
<td>149</td>
</tr>
<tr>
<td>Flávio Luiz Schiavoni, Marcelo Queiroz, Fernando Iazzetta</td>
<td></td>
</tr>
<tr>
<td>Xth Sense: researching muscle sounds for an experimental paradigm of</td>
<td>157</td>
</tr>
<tr>
<td>musical performance</td>
<td></td>
</tr>
<tr>
<td>Marco Donnarumma</td>
<td></td>
</tr>
<tr>
<td>Composing a piece for piano and electronics on Linux</td>
<td>165</td>
</tr>
<tr>
<td>Lorenzo Franco Sutton</td>
<td></td>
</tr>
<tr>
<td>Concerto para Lanhouse</td>
<td>170</td>
</tr>
<tr>
<td>Giuliano Obici and Flávio Luiz Schiavoni</td>
<td></td>
</tr>
</tbody>
</table>
Two Recent Extensions to the FAUST Compiler

K. Barkati
Ircam – SEL Team
karim.barkati@ircam.fr

D. Fober
Grame
fober@grame.fr

S. Letz
Grame
letz@grame.fr

Y. Orlarey
Grame
orlarey@grame.fr

ABSTRACT

We present two recently introduced extensions to the FAUST compiler. The first one concerns the architecture system and provides Open Sound Control (OSC) support to all FAUST generated applications. The second extension is related to preservation issues and provides a mean to automatically compute an all-comprehensive mathematical documentation of any FAUST program.

1. INTRODUCTION

FAUST\(^1\) (Functional Audio Stream) is a functional, synchronous, domain specific language designed for real-time signal processing and synthesis. A unique feature of FAUST, compared to other existing languages like Max, PD, SuperCollider, etc., is that programs are not interpreted, but fully compiled.

One can think of FAUST as a specification language. It aims at providing the user with an adequate notation to describe signal processors from a mathematical point of view. This specification is free, as much as possible, from implementation details. It is the role of the FAUST compiler to provide automatically the best possible implementation. The compiler translates FAUST programs into equivalent C++ programs taking care of generating the most efficient code. The compiler offers various options to control the generated code, including options to do fully automatic parallelization and take advantage of multicore machines.

From a syntactic point of view FAUST is a textual language, but nevertheless block-diagram oriented. It actually combines two approaches: functional programming and algebraic block-diagrams. The key idea is to view block-diagram construction as function composition. For that purpose, FAUST relies on a block-diagram algebra of five composition operations (\(\cdot\), \(\sim\), \(<:\), \(\rightarrow\)).

For more details on the language we refer the reader to [1] [2]. Here is how to write a pseudo random number generator \(r\) in FAUST\(^2\):

\[
r = +(12345) \sim (1103515245);
\]

This example uses the recursive composition operator \(\sim\) to create a feedback loop as illustrated figure 1.

The code generated by the FAUST compiler works at the sample level, it is therefore suited to implement low-level DSP functions like recursive filters up to full-scale audio applications. It can be easily embedded as it is self-contained and doesn’t depend of any DSP library or runtime system. Moreover, it has a very deterministic behavior and a constant memory footprint.

The compiler can also wrap the generated code into an architecture file that describes how to relate the DSP computation to the external world. We have recently reorganized some of these architecture files in order to provide Open Sound Control (OSC) support. All FAUST generated applications can now be controlled by OSC. We will describe this evolution section 2.

Another recent addition is a new documentation backend to the FAUST compiler. It provides a mean to automatically compute an all-comprehensive mathematical documentation of a FAUST program under the form of a complete set of \(\LaTeX\) formulas and diagrams. We will describe this Self Mathematical Documentation system section 3.

2. ARCHITECTURE FILES

Being a specification language, FAUST programs say nothing about audio drivers nor GUI toolkits to be used. It is the role of the architecture file to describe how to relate the DSP module to the external world. This approach allows a single FAUST program to be easily deployed to a large variety of audio standards (Max/MSP externals, PD externals, VST plugins, CoreAudio applications, Jack applications, iPhone, etc.). In the following sections we will detail this architecture mechanism and in particular the recently developed OSC architecture that allows FAUST programs to be controlled by OSC messages.

---

\(^1\)http://faust.grame.fr

\(^2\)Please note that this expression produces a signal \(r(t) = 12345 + 1103515245 \ast r(t - 1)\) that exploits the particularity of 32-bits integer operations.
2.1 Audio architecture files

A FAUST audio architecture typically connects the FAUST DSP module to the audio drivers. It is responsible for allocating and releasing the audio channels and to call the FAUST `dsp::compute` method to handle incoming audio buffers and/or to produce audio output. It is also responsible for presenting the audio as non-interleaved float data, normalized between -1.0 and 1.0.

A FAUST audio architecture derives an audio class defined as below:

```cpp
class audio {
    public:
        audio() {} // constructor
        virtual ~audio() {} // destructor
    virtual bool init(const char*, dsp*) = 0;
    virtual bool start() = 0;
    virtual void stop() = 0;
};
```

The API is simple enough to give a great flexibility to audio architectures implementations. The `init` method should initialize the audio. At `init` exit, the system should be in a safe state to recall the `dsp` object state.

Table 1 gives the audio architectures currently available for various operating systems.

<table>
<thead>
<tr>
<th>Audio system</th>
<th>Operating system</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alsa</td>
<td>Linux, Mac OS X, Windows</td>
</tr>
<tr>
<td>Core audio</td>
<td>Mac OS X, iOS</td>
</tr>
<tr>
<td>Jack</td>
<td>Linux, Mac OS X, Windows</td>
</tr>
<tr>
<td>Portaudio</td>
<td>Linux, Mac OS X, Windows</td>
</tr>
<tr>
<td>OSC (see 2.3.2)</td>
<td>Linux, Mac OS X, Windows</td>
</tr>
<tr>
<td>VST</td>
<td>Mac OS X, Windows</td>
</tr>
<tr>
<td>Max/MSP</td>
<td>Mac OS X, Windows</td>
</tr>
<tr>
<td>Csound</td>
<td>Linux, Mac OS X, Windows</td>
</tr>
<tr>
<td>SuperCollider</td>
<td>Linux, Mac OS X, Windows</td>
</tr>
<tr>
<td>PureData</td>
<td>Linux, Mac OS X, Windows</td>
</tr>
<tr>
<td>Pure [3]</td>
<td>Linux, Mac OS X, Windows</td>
</tr>
</tbody>
</table>

Table 1. FAUST audio architectures.

2.2 GUI architecture files

A FAUST UI architecture is a glue between a host control layer (graphic toolkit, command line, OSC messages, etc.) and the FAUST DSP module. It is responsible for associating a FAUST DSP module parameter to a user interface element and to update the parameter value according to the user actions. This association is triggered by the `dsp::buildUserInterface` call, where the `dsp` asks a UI object to build the DSP module controllers.

Since the interface is basically graphic oriented, the main concepts are `widget` based: a UI architecture is semantically oriented to handle active widgets, passive widgets and widgets layout.

A FAUST UI architecture derives an `UI` class (Figure 2).

2.2.1 Active widgets

Active widgets are graphical elements that control a parameter value. They are initialized with the widget name and a pointer to the linked value. The widget currently considered are `Button`, `ToggleButton`, `CheckButton`, `VerticalSlider`, `HorizontalSlider` and `NumEntry`. A GUI architecture must implement a method `addXxx (const char* name, float* zone, ...)` for each active widget. Additional parameters are available for `Slider` and `NumEntry`: the `init` value, the `min` and `max` values and the `step`.

2.2.2 Passive widgets

Passive widgets are graphical elements that reflect values. Similarly to active widgets, they are initialized with the widget name and a pointer to the linked value. The widget currently considered are `NumDisplay`, `TextDisplay`, `HorizontalBarGraph` and `VerticalBarGraph`. A UI architecture must implement a method `addXxx (const char* name, float* zone, ...)` for each passive widget. Additional parameters are available, depending on the passive widget type.

2.2.3 Widgets layout

Generally, a GUI is hierarchically organized into boxes and/or tab boxes. A UI architecture must support the following methods to setup this hierarchy:

- `openTabBox (const char* l)`
- `openHorizontalBox (const char* l)`
- `openVerticalBox (const char* l)`
- `closeBox (const char* l)`

Note that all the widgets are added to the current box.
2.2.4 Metadata

The FAUST language allows widget labels to contain metadata enclosed in square brackets. These metadata are handled at GUI level by a `declare` method taking as argument, a pointer to the widget associated value, the metadata key and value:

```
depend( float*, const char*, const char*)
```

Table 2. Available UI architectures.

<table>
<thead>
<tr>
<th>UI</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>console</td>
<td>a textual command line UI</td>
</tr>
<tr>
<td>GTK</td>
<td>a GTK-based GUI</td>
</tr>
<tr>
<td>Qt</td>
<td>a multi-platform Qt-based GUI</td>
</tr>
<tr>
<td>FUI</td>
<td>a file-based UI to store and recall modules states</td>
</tr>
<tr>
<td>OSC</td>
<td>OSC control (see 2.3.1)</td>
</tr>
</tbody>
</table>

2.3 OSC architectures

The OSC [4] support opens the FAUST applications control to any OSC capable application or programming language. It also transforms a full range of devices embedding sensors (wiimote, smart phones, ...) into physical interfaces for FAUST applications control, allowing a direct use as music instruments (which is in phase with the new FAUST physical models library [5] adapted from STK [6]).

The FAUST OSC architecture is twofold: it is declined as a UI architecture and also as an audio architecture, proposing a new and original way to make digital signal computation.

2.3.1 OSC GUI architecture

The OSC UI architecture transforms each UI active widget addition into an `addnode` call, ignores the passive widgets and transforms containers calls (`openXxxBox, closeBox`) into `opengroup` and `closegroup` calls.

The OSC address space adheres strictly to the hierarchy defined by the `addnode` and `opengroup, closegroup` calls. It supports the OSC pattern matching mechanism as described in [4].

A node expects to receive OSC messages with a single float value as parameter. This policy is strict for the parameters count, but relaxed for the parameter type: OSC int values are accepted and casted to float.

Two additional messages are defined to provide FAUST applications discovery and address space discoveries:

- the `hello` message: accepted by any module root address. The module responds with its root address, followed by its IP address, followed by the UDP ports numbers (listening port, output port, error port). See the network management section below for ports numbering scheme.
- the `get` message: accepted by any valid OSC address. The `get` message is propagated to every terminal node that responds with its OSC address and current values (value, min and max).

Table 3. OSC support in FAUST applications architectures.

<table>
<thead>
<tr>
<th>Audio system</th>
<th>Environment</th>
<th>OSC support</th>
</tr>
</thead>
<tbody>
<tr>
<td>Linux</td>
<td>GTK, Qt</td>
<td>yes</td>
</tr>
<tr>
<td>Jack</td>
<td>GTK, Qt, Console</td>
<td>yes</td>
</tr>
<tr>
<td>PortAudio</td>
<td>GTK, Qt</td>
<td>yes</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>Qt</td>
<td>yes</td>
</tr>
<tr>
<td>CoreAudio</td>
<td>Qt, Console</td>
<td>yes</td>
</tr>
<tr>
<td>Jack</td>
<td>Qt</td>
<td>yes</td>
</tr>
<tr>
<td>PortAudio</td>
<td>Qt</td>
<td>yes</td>
</tr>
<tr>
<td>Windows</td>
<td>Qt</td>
<td>yes</td>
</tr>
<tr>
<td>iOS (iPhone)</td>
<td>Cocoa</td>
<td>not yet</td>
</tr>
</tbody>
</table>

Example:

Consider the `noise` module provided with the FAUST examples:

- it sends `/noise 192.168.0.1 5510 5511 5512` in answer to a `hello` message,
- it sends `/noise/Volume 0.8 0. 1.` in answer to a `get` message.

The OSC architecture makes use of three different UDP port numbers:

- 5510 is the listening port number: control messages should be addressed to this port.
- 5511 is the output port number: answers to query messages are send to this port.
- 5512 is the error port number: used for asynchronous errors notifications.

When the UDP listening port number is busy (for instance in case of multiple FAUST modules running), the system automatically looks for the next available port number. Unless otherwise specified by the command line, the UDP output port numbers are unchanged.

A module sends its name (actually its root address) and allocated ports numbers on the OSC output port on startup. Ports numbers can be changed on the command line with the following options:

```
[-port | -outport | -errport] number
```

The default UDP output streams destination is `localhost`. It can also be changed with the command line option

```
-dest address
```

Where address is a host name or an IP number.

2.3.2 OSC audio architecture

The OSC audio architecture implements an audio architecture where audio inputs and outputs are replaced by OSC messages. Using this architecture, a FAUST module accepts arbitrary data streams on its root OSC address, and handles this input stream as interleaved signals. Thus, each
incoming OSC packet addressed to a module root triggers a computation loop, where as much values as the number of incoming frames are computed.

The output of the signal computation is sent to the OSC output port as non-interleaved data to the OSC addresses /root/n where root is the module root address and n is the output number (indexed from 0).

For example:

consider a FAUST program named split and defined by:

```plaintext
process = _ <=: _-
```

the message

```
/split 0.3
```

will produce the 2 following messages as output:

```
/split/0 0.3
/split/1 0.3
```

The OSC audio architecture provides a very convenient way to execute a signal processing at an arbitrary rate, allowing even to make step by step computation. Connecting the output OSC signals to Max/MSP or to a system like INScore, featuring a powerful dynamic signals representation system [7], provides a close examination of the computation results.

#### 2.4 Open issues and future works

Generally, the labeling scheme for a GUI doesn’t result in an optimal OSC address space definition. Moreover, there are potential conflicts between the FAUST UI labels and the OSC address space since some characters are reserved for OSC pattern matching and thus forbidden in the OSC naming scheme. The latter issue is handled with automatic characters substitutions. The first issue could be solved using the metadata scheme and will be considered in a future release.

Another issue, resulting from the design flexibility, relies on dynamic aggregation of multiple architectures covering the same domain: for example, it would be useful to embed both a standard and the OSC audio architecture in the same module and to switch dynamically between (for debugging purposes for example). That would require the UI to include the corresponding control and thus a mechanism to permit the UI extension by the UI itself would be necessary.

#### 3. SELF MATHEMATICAL DOCUMENTATION

Another recent addition to the FAUST compiler is the Self Mathematical Documentation developed within ASTREE, an ANR funded research project (ANR 08-CORD-003) on preservation of real-time music works involving IRCAM, GRAME, MINES-PARISTECH and UJM-CIEREC.

The problem of documentation is well known in computer programming at least since 1984 and Donald Knuth’s claim [8]: “I believe that the time is ripe for significantly better documentation of programs [...]”

A quarter-century later, general purpose programming languages can use doxygen, javadoc or others Literate Programming tools. But computer music languages lack integrated documentation systems and preservation of real-time music works is a big issue [9].

The self mathematical documentation extension to the FAUST compiler precisely addresses this question for digital signal processing (unfortunately not yet the asynchronous and more complex part). It provides a mean to automatically compute an all-comprehensive mathematical documentation of a FAUST program under the form of a complete set of \LaTeX\ formulas and diagrams.

One can distinguish four main goals, or uses, of such a self mathematical documentation:

1. **Preservation**, i.e. to preserve signal processors, independently from any computer language but only under a mathematical form;
2. **Validation**, i.e. to bring some help for debugging tasks, by showing the formulas as they are really computed after the compilation stage;
3. **Teaching**, i.e. to give a new teaching support, as a bridge between code and formulas for signal processing;
4. **Publishing**, i.e. to output publishing material, by preparing \LaTeX\ formulas and SVG block diagrams easy to include in a paper.

The first and likely most important goal of preservation relies on the strong assumption that *maths will last far longer than any computer language*. This means that once printed on paper, a mathematical documentation becomes a *long-term preserveable* document, as the whole semantics of a DSP program is translated into two languages independent from any computer language and from any computer environment: the mathematical language, mainly, and the natural language, used to structure the presentation for the human reader and also to precise some local mathematical items (like particular symbols for integer operations). Thus, the mathematical documentation is self-sufficient to a programmer for reimplementing a DSP program, and shall stay self-sufficient for decades and probably more!

#### 3.1 The faust2mathdoc Command

The FAUST self mathematical documentation system relies on two things: a new compiler option --mathdoc and a shell script faust2mathdoc. The script first calls `faust --mathdoc`, which generates:

- a top-level directory suffixed with ”-mdoc”.
- 5 subdirectories (`cpp/, pdf/, src/, svg/, tex/`),
- a \LaTeX\ file containing the formulas,
- SVG files for the block diagrams;

then it just finishes the work done by the FAUST compiler,

- moving the output C++ file into `cpp/`,
- converting all SVG files into PDF files,
- launching `pdflatex` on the \LaTeX\ file,
3.2 Automatic Mode

The user has the possibility to introduce in the FAUST program special tags to control the generated documentation. When no such tags are introduced, we are in the so-called automatic mode. In this case everything is automatic and the generated PDF document is structured in four sections:

1. “Mathematical definition of process”
2. “Block diagram of process”
3. “Notice”
4. “Faust code listings”

3.2.1 Front Page

First, to give an idea, let’s look at the front page of a mathematical documentation. Figure 3 shows the front page of the PDF document generated from the freeverb dsp FAUST program (margins are cropped).

The header items are extracted from the metadatas declared in the FAUST file:

```plaintext
declare name "freeverb";
declare version "1.0";
declare author "Grame";
declare license "BSD";
declare copyright "(c)GRAME 2006";
```

The date of the documentation compilation is inserted and some glue text is added to introduce each section and the document itself. So, in addition to the mathematical language, the document also relies on the natural language, but one can legitimately expect it to last far longer than any current computer language.

3.2.2 Mathematical definition of process

The first printed section contains whole mathematical definition of process. Obviously, the computation of the formulas printing is the most important part of the mathematical documentation.

To handle a LATEX output for the mathematical documentation, instead of using a simple pattern matching substitution, the FAUST compiler has been extended from within, by reimplementing the main classes, in order to print a normalized form of the equations. This means that like the standard C++ output of the compiler, the LATEX output is computed after the compilation of the signal processors, thus benefiting from all simplifications and normalizations that the FAUST compiler is able to do.

Some printed formulas are shown on Figure 3 (from the freeverb dsp file) and Figure 4 (from HPF dsp, a high-pass filter), as they appear in the corresponding generated PDF documents.

On Figure 3, one can see the definition of three kinds of signals, while on Figure 4 one can see two other kinds, and these are exactly the five families of signals that are handled:

- “Output signals”,
- “Input signals”,
- “User-interface input signals”,
- “Intermediate signals”,
- “Constant signals”.

In fact, the documentator extension of the FAUST compiler manages several kinds of signals and makes a full use of FAUST signal tagging capabilities to split the equations.
4. Intermediate signals $p_i$ for $i \in [1, 8]$ and $r_1$ such that

\[
\begin{align*}
p_i(t) &= k_1 \cdot \max(0, u_{s1}(t)) \\
p_2(t) &= \cos(p_1(t)) \\
p_3(t) &= 2 \cdot p_2(t) \\
p_4(t) &= 0.5 \cdot \sin(p_1(t)) \\
p_5(t) &= (p_2(t) - 1) \\
p_6(t) &= (1 + p_3(t)) \\
p_7(t) &= 0.5 \cdot p_6(t) \\
p_8(t) &= \frac{1}{1 + p_4(t)} \\
r_1(t) &= p_9(t) \cdot (x_1(t-1) \cdot (0 - (p_8(t) - x_1(t)))) + p_7(t) \cdot x_1(t) + p_8(t) \cdot r_1(t-2) + p_1(t) \cdot r_1(t-1) \\
\end{align*}
\]

5. Constant $k_1$ such that

\[
k_1 = \frac{6.28318530717959}{fs}
\]

**Figure 4.** Some printed formulas.

This is very important for human readability’s sake, or else there would be only one very long formula for process! No need to make things so complex, the documentation pushes this idea a step further than the five main signal families, using letters and numeric indices to name the left member of each subequation.

The indices are easy to understand: on Figure 3 for example, mentions like “$y_1(t)$”, “$y_2(t)$” and “Input signals $x_i$ for $i \in [1, 2]$” clearly indicates that the freeverb block diagram has two input signals and two output signals, i.e. is a stereo signal transformer.

The letter choice is a bit more complex, summarised in Table 4.

### 3.2.3 Fine mathematical automatic display

#### 3.2.4 Block diagram of process

The second section draws the top-level block diagram of process, i.e. a block diagram that fits on one page. The appropriate fitting is computed by the FAUST compiler part that handles the SVG output.

Figure 1 shows the block diagram computed from the noise.dsp file (a noise generator). By default, the top-level SVG block diagram of process is generated, converted into the PDF format through the svg2pdf utility (using the 2D graphics Cairo library), entitiled and inserted in the second section of the documentation as a floating LaTeX figure (in order to be referenceable).

### 3.2.5 Notice

The third section presents the notice, to enlighten the documentation, divided in two parts:

- a common header (shown on Figure 6);
- a dynamic mathematical body (an example is shown on Figure 7, from the capture dsp file).

<table>
<thead>
<tr>
<th>Letter</th>
<th>Signal Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>$y(t)$</td>
<td>Output signal</td>
</tr>
<tr>
<td>$x(t)$</td>
<td>Input signal</td>
</tr>
<tr>
<td>$u_0(t)$</td>
<td>User-interface button input signal</td>
</tr>
<tr>
<td>$u_r(t)$</td>
<td>User-interface checkbox input signal</td>
</tr>
<tr>
<td>$u_s(t)$</td>
<td>User-interface slider input signal</td>
</tr>
<tr>
<td>$u_u(t)$</td>
<td>User-interface numeric box input signal</td>
</tr>
<tr>
<td>$u_y(t)$</td>
<td>User-interface bargraph output signal</td>
</tr>
<tr>
<td>$p(t)$</td>
<td>Intermediate parameter signal (running at control rate)</td>
</tr>
<tr>
<td>$s(t)$</td>
<td>Intermediate simple signal (running at control rate)</td>
</tr>
<tr>
<td>$r(t)$</td>
<td>Intermediate recursive signal (depends on previous samples $r(t-n)$)</td>
</tr>
<tr>
<td>$q(t)$</td>
<td>Intermediate selection signal (2 or 3-ways selectors)</td>
</tr>
<tr>
<td>$m(t)$</td>
<td>Intermediate memory signal (1-sample delay explicitly initialized)</td>
</tr>
<tr>
<td>$v(t)$</td>
<td>Intermediate table signal (read or read-and-write tables)</td>
</tr>
<tr>
<td>$k(t)$</td>
<td>Constant signal</td>
</tr>
</tbody>
</table>

**Table 4.** Sub-signal formulas naming.

For later reading improvement purposes, the first part intensively uses the natural language to contextualize the documentation as much as possible, giving both contextual information – with the compiler version, the compilation date, a block diagram presentation, the FAUST and SVG URLS, the generated documentation directory tree – and key explanations on the FAUST language itself, its (denotational) mathematical semantics – including the process identifier, signals and signal transformers semantics.

#### 3.2.6 Faust code listings

The fourth and last section provides the complete listings. All FAUST code is inserted into the documentation, the main source code file and all needed libraries, using the pretty-printer system provided by the *listings* LATEX package.

You may wonder why we print FAUST code listings while the FAUST language is also affected by our mathematical abstraction *moto* that maths will last far longer than any computer language... It is mainly to add another help item for contextualization! Indeed, depending on the signal processing algorithms and implementations, some FAUST code can prove extremely helpful to understand the printed formulas, in the view of re-implmenting the same algorithm in decades under other languages.

### 3.3 Manual Documentation

You can specify yourself the documentation instead of using the automatic mode, with five xml-like tags. That permits to modify the presentation and to add your own comments, not only on process, but also about any expression you’d like to. Note that as soon as you declare an `<mdoc>` tag inside your FAUST file, the default structure of the au-
3 Notice

- This document was generated using Faust version 0.9.36 on March 14, 2011.
- The value of a Faust program is the result of applying the signal transformer denoted by the expression to which the process identifier is bound to input signals, running at the $f_s$ sampling frequency.
- Faust (Functional Audio Stream) is a functional programming language designed for synchronous real-time signal processing and synthesis applications. A Faust program is a set of bindings of identifiers to expressions that denote signal transformers. A signal $s$ in $S$ is a function mapping times $t \in \mathbb{Z}$ to values $s(t) \in \mathbb{R}$, while a signal transformer is a function from $S^n$ to $S^m$, where $n, m \in \mathbb{N}$. See the Faust manual for additional information (http://faust.gram.fr/).
- Every mathematical formula derived from a Faust expression is assumed to have been normalized (in an implementation-dependent manner) by the Faust compiler.
- A block diagram is a graphical representation of the Faust binding of an identifier $I$ to an expression $E$; each graph is put in a box labeled by $I$. Subexpressions of $E$ are recursively displayed as long as the whole picture fits in one page.
- The BPF-mdoc/ directory may also include the following subdirectories:
  - \texttt{cpp/} for Faust compiled code;
  - \texttt{pdf/} which contains this document;
  - \texttt{src/} for all Faust sources used (even libraries);
  - \texttt{svg/} for block diagrams, encoded using the Scalable Vector Graphics format (http://www.w3.org/Graphics/SVG/); 
  - \texttt{tex/} for the \LaTeX\ source of this document.

3.4 Practical Aspects

3.4.1 Installation Requirements

Here follows a summary of the installation requirements to generate the mathematical documentation:

- \texttt{faust}, of course!
- \texttt{svg2pdf} (from the Cairo 2D graphics library), to convert block diagrams, as \LaTeX\ doesn’t handle SVG directly yet...
- \texttt{breqn}, a \LaTeX\ package to manage automatic breaking of long equations,
- \texttt{pdflatex}, to compile the \LaTeX\ output file.

3.4.2 Generating the Mathematical Documentation

The easiest way to generate the complete mathematical documentation is to call the \texttt{faust2mathdoc} script on a FAUST file, as the \texttt{--mdoc} option leave the documentation production unfinished. For example:

\texttt{faust2mathdoc myfaustfile.dsp}

The PDF file is then generated in the appropriate directory \texttt{myfaustfile-mdoc/pdf/myfaustfile.pdf}.

3.4.3 Online Examples

To have an idea of the results of this mathematical documentation, which captures the mathematical semantic of FAUST programs, you can look at two pdf files online:

- \texttt{http://faust.gram.fr/pdf/karplus.pdf} (automatic documentation),

3.5 Conclusion

We have presented two extensions to the FAUST compiler: an \textit{architecture} system that provides OSC support to FAUST generated applications, and an automatic documentation generator able to produce a full mathematical description of any FAUST program.

The idea behind the FAUST’s architecture system is separation of concerns between the DSP computation itself and its use. It turns out to be a flexible and powerful idea: any new or improved architecture file, like here OSC support, benefits to all applications without having to modify the FAUST code itself. We have also split some of these architectures into separate Audio and UI modules that are

\begin{dmath*}
 p_{4}(t) = 0.5 \frac{\sin(p_{1}(t))}{\max(0.001, u_{s_{2}}(t))}
\end{dmath*}
• \( x \in \mathbb{R} \).
\[
\text{int}(x) = \begin{cases} 
  x & \text{if } x \geq 0 \\
  x & \text{if } x < 0 \\
  0 & \text{if } x = 0 
\end{cases}
\]

• This document uses the following integer operations:

<table>
<thead>
<tr>
<th>operator</th>
<th>name</th>
<th>semantics</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>integer addition</td>
<td>( \text{normalise}((+), \text{in } \mathbb{Z}) )</td>
</tr>
<tr>
<td>i</td>
<td>integer subtraction</td>
<td>( \text{normalise}(-), \text{in } \mathbb{Z} )</td>
</tr>
<tr>
<td>*</td>
<td>integer multiplication</td>
<td>( \text{normalise}(\cdot), \text{in } \mathbb{Z} )</td>
</tr>
</tbody>
</table>

Integer operations in Faust are inspired by the semantics of operations on the n-bit two’s complement representation of integer numbers: they are internal composition laws on the subset \([-2^{n-1}, 2^{n-1}-1]\) of \(\mathbb{Z}\) with \(n = 32\). For any integer binary operation \(\times\) on \(\mathbb{Z}\), the operation is defined as: \(i = \text{normalise}(i \times)\), with
\[
\text{normalise}(i) = i - \left\lfloor \frac{\text{sign}(i) \times (1 + 2 \times (\text{sign}(i) - 1) \times 2)}{} \right\rfloor,
\]
where \(2^n\) and \(\text{sign}(i) = 0\) if \(i = 0\) and \(\text{sign}(i) = 0\) otherwise. Unary integer operations are defined likewise.

\[
\text{process} = \text{noise} * \text{vslider}(\text{Volume}[\text{style:knob}], 0, 0, 1, 0.1);
\]

\[
\text{process} = \text{random} / 2147483647.0;
\]

Figure 7. Dynamic part of a printed notice.

4 Listing of the input code

The following listing shows the input Faust code, parsed to compile this mathematical documentation.

Listing 1: noisemetadata.dsp

```
// Noise generator and demo file for the Faust math documentation.

// Declare name "Noise";
Declare version "1.0.0";
Declare author "Yghe"
Declare license "BSD";
Declare copyright "(c)GRAME 2009";
random = uniform(0, 1);
noise = random/2147483647.0;
process = noise * vslider("Volume[style:knob]", 0, 0, 1, 0.1);
```

Figure 8. Faust code listing.

easier to maintain or evolve. This provides another layer of flexibility.

The self mathematical documentation system, while not simple to develop, turns out to be feasible because FAUST has a simple and well defined semantic. It is therefore possible to compute a semantic description of what a FAUST program does whatever its complexity. Moreover this semantic description was readily available inside the FAUST compiler because already used to optimize the generated C++ code.

This example shows that semantics is not only of theoretical interest and can have very practical benefits. We would like therefore to encourage developers to consider this aspect, as well as preservation issues, when designing new audio/music tools or languages.

3.6 Acknowledgments

This work has been partially supported by the French National Agency (ANR) in the context of the ASTREE project (ANR-2008-CORD-003-01).

4. REFERENCES

Introducing Kronos
A Novel Approach to Signal Processing Languages

Vesa Norilo
Centre for Music & Technology, Sibelius Academy
Pohjoinen Rautatiekatu 9
00100 Helsinki,
Finland,
vnorilo@siba.fi

Abstract
This paper presents an overview of Kronos, a software package aimed at the development of musical signal processing solutions. The package consists of a programming language specification as well JIT Compiler aimed at generating high performance executable code.

The Kronos programming language aims to be a functional high level language. Combining this with run time performance requires some unusual trade-offs, creating a novel set of language features and capabilities.

Case studies of several typical musical signal processors are presented and the suitability of the language for these applications is evaluated.

Keywords
Music, DSP, Just in Time Compiler, Functional, Programming language

1 Introduction
Kronos aims to be a programming language and a compiler software package ideally suited for building any custom DSP solution that might be required for musical purposes, either in the studio or on the stage. The target audience includes technologically inclined musicians as well as musically competent engineers. This prompts a re-evaluation of design criteria for a programming environment, as many musicians find industrial programming languages very hostile.

On the other hand, the easily approachable applications currently available for building musical DSP algorithms often fail to address the requirements of a programmer, not providing enough abstraction nor language constructs to facilitate painless development of more complicated systems.

Many software packages from Pure Data[Puckette, 1996] to Reaktor[Nicholl, 2008] take the approach of more or less simulating a modular synthesizer. Such packages combine a varying degree of programming language constructs into the model, yet sticking very closely to the metaphor of connecting physical modules via patch cords. This design choice allows for an environment that is readily comprehensible to anyone familiar with its physical counterpart. However, when more complicated programming is required, the apparent simplicity seems to deny the programmer the special advantages provided by digital computers.

Kronos proposes a solution more closely resembling packages like Supercollider[McCarteney, 2002] and Faust[Orlarey et al., 2004], opting to draw inspiration from computer science and programming language theory. The package is fashioned as a just in time compiler[Aycock, 2003], designed to rapidly transform user algorithms into efficient machine code.

This paper presents the actual language that forms the back end on which the comprehensive DSP development environment will be built. In Section 2, Language Design Goals, we lay out the criteria adopted for the language design. In Section 3, Designing the Kronos Language, the resulting design problems are addressed. Section 5, Case Studies, presents several signal processing applications written in the language, presenting comparative observations of the efficacy our proposed solution to each case. Finally, Section 6, Conclusion, summarizes this paper and describes future avenues of research.

2 Language Design Goals
This section presents the motivation and aspirations for Kronos as a programming language. Firstly, the requirements the language should be able to fulfill are enumerated. Secondly, summarized design criteria are derived from the requirements.
2.1 Musical Solutions for Non-engineers

Since the target audience of Kronos includes non-engineers, the software should ideally be easily approached. In this regard, the visually oriented patching environments hold an advantage.

A rigorously designed language offers logical cohesion and structure that is often missing from a software package geared towards rapid visual construction of modular ad-hoc solutions. Consistent logic within the environment should ease learning.

The ideal solution should be that the environment allows the casual user to stick to the metaphor of physical interconnected devices, but also offers an avenue of more abstract programming for advanced and theoretically inclined users.

2.2 DSP Development for Professionals

Kronos also aspires to be an environment for professional DSP developers. This imposes two additional design criteria: the language should offer adequately sophisticated features, so that more powerful programming constructs can be used if desired. The resulting audio processors should also exhibit excellent real time performance.

A particularly challenging feature of a musical DSP programming is the inherent multi-rate processing. Not all signals need equally frequent updates. If leveraged, this fact can bring about dramatic performance benefits. Many systems offer a distinction between control rate and audio rate signals, but preferably this forced distinction should be eliminated and a more general solution be offered, inherent to the language.

2.3 An Environment for Learning

If a programming language can be both beginner friendly and advanced, it should appeal to developers with varying levels of competency. It also results in an ideal pedagogical tool, allowing a student to start with relatively abstraction-free environment, resembling a modular synthesizer, progressing towards higher abstraction and efficient programming practices.

2.4 A Future Proof Platform

Computing is undergoing a fundamental shift in the type of hardware commonly available. It is essential that any programming language designed today must be geared towards parallel computation and execution on a range of differing computational hardware.

2.5 Summary of the Design Criteria

Taking into account all of the above, the language should:

- Be designed for visual syntax and graphical user interfaces
- Provide adequate abstraction and advanced programming constructs
- Generate high performance code
- Offer a continuous learning curve from beginner to professional
- Be designed to be parallelizable and portable

3 Designing the Kronos Language

This section will make a brief case for the design choices adapted in Kronos.

3.1 Functional Programming

The functional programming paradigm [Hudak, 1989] is the founding principle in Kronos. Simultaneously fulfilling a number of our criteria, we believe it to be the ideal choice.

Compared to procedural languages, functional languages place less emphasis on the order of statements in the program source. Functional programs are essentially signal flow graphs, formed of processing nodes connected by data flow.

Graphs are straightforward to present visually. The nodes and data flows in such trees are also something most music technologists tend to understand well. Much of their work is based on making extensive audio flow graphs.

Functional programming also offers extensive abstraction and sophisticated programming constructs. These features should appeal to advanced programmers.

Further, the data flow metaphor of programming is ideally suited for parallel processing, as the language can be formally analyzed and
transformed while retaining algorithmic equivalence. This is much harder to do for a procedural language that may rely on a very particular order of execution and hidden dependencies.

Taken together, these factors make a strong case for functional programming for the purposes of Kronos and recommend its adoption. However, the functional paradigm is quite unlike what most programmers are used to. The following sections present some key differences from typical procedural languages.

3.1.1 No state
Functional programs have no state. The output of a program fragment is uniquely determined by its input, regardless of the context in which the fragment is run. Several further features and constraints emerge from this fundamental property.

3.1.2 Bindings Instead of Variables
Since the language is based on data flow instead of a series of actions, there is no concept of a changeable variable. Functional operators can only provide output from input, not change the state of any external entity.

However, symbols still remain useful. They can be used to bind expressions, making code easier to write and read.

3.1.3 Higher Order Functions Instead of Loops
Since the language has no variables, traditional loops are not possible either, as they rely on a loop iteration variable. To accomplish iterative behavior, functional languages employ recursion and higher order functions[Kemp, 2007]. This approach has the added benefit of being easier to depict visually than traditional loop constructs based on textual languages – notoriously hard to describe in a patching environment.

As an example, two higher order functions along with example replies are presented in Listing 1.

Listing 1: Higher order functions with example replies
/* Apply the mapping function Sqrt to all elements of a list */
Algorithm:Map(Sqrt 1 2 3 4 5) => (1 1.41421 1.73205 2 2.23607)
/* Combine all the elements of a list using a folding function, Add */
Algorithm:Fold(Add 1 2 3 4 5) => 15

3.1.4 Polymorphism Instead of Flow Control
A typical procedural program contains a considerable amount of branches and logic statements. While logic statements are part of functional programming, flow control often happens via polymorphism. Several different forms can be defined for a single function, allowing the compiler to pick an appropriate form based on the argument type.

Polymorphism and form selection is also the mechanism that drives iterative higher order functions. The implementation for one such function, Fold, is presented in Listing 2. Fold takes as an argument a folding function and a list of numbers.

While the list can be split into two parts, x and xs, the second form is utilized. This form recurs with xs as the list argument. This process continues, element by element, until the list only contains a single unsplittable element. In that boundary case the first form of the function is selected and the recursion terminates.

Listing 2: Fold, a higher order function for reducing lists with example replies.
Fold(folding-function x)
{
Fold = x
}
Fold(folding-function x xs)
{
Fold = Eval(folding-function x Fold(folding-function xs))
}

/* Add several numbers */
Fold(Add 1 2 3 4) => 10

/* Multiply several numbers */
Fold(Mul 5 6 10) => 300

3.2 Generic Programming and Specialization
3.2.1 Generics for Flexibility
Let us examine a scenario where a sum of several signals in differing formats is needed. Let us assume that we have defined data types for mono and stereo samples. In Kronos, we could easily define a summation node that provides mono output when all its inputs are mono, and stereo when at least one input is stereo.

An example implementation is provided in Listing 3. The listing relies on the user defining semantic context by providing types, Mono and Stereo, and providing a Coerce method that can upgrade a Mono input to a Stereo output.

Listing 3: User-defined coercion of mono into stereo
Type Mono
Package Mono
| Cons(sample) /* wrap a sample in type context 'Mono' */ |
| {Cons = Make(:Mono sample)} |
Get-Sample(sample) /* retrieve a sample from 'Mono' context */ |
| {Get-Sample = Break(:Mono sample)} |

3.2.2 Specialization
Type Stereo
Package Stereo{
Cons(sample) /* wrap a sample in type context 'Stereo' */
{Cons = Make(:Stereo sample)}
L/R(sample) /* provide accessors to assumed Left and Right channels */
{(L R) = Break(:Stereo sample)}
}

Add(a b) {
/* How to add 'Mono' samples */
Add = Mono:Cons(Mono:Get-Sample(a) + Mono:Get-Sample(b))
/* How to add 'Stereo' samples */
Add = Stereo:Cons(Stereo:L(a) + Stereo:L(b) Stereo:R(a) + Stereo:R(b))
}

Coerce(desired-type smp) {
/* Provide type upgrade from mono to stereo by duplicating channels */
Coerce = When(
Type-Of(desired-type) == Stereo
Coerce = Stereo:Cons(
Mono:Get-Sample(smp) Mono:Get-Sample(smp)))
)
/* Provide a mixing function to sum a number of channels */
Mix-Bus(ch) {
Mix-Bus = ch
}

Mix-Bus(ch chs) {
Mix-Bus = ch + Recur(chs)
}

Note that the function Mix-Bus in Listing 3 needs to know very little about the type of data passed to it. It is prepared to process a list of channels via recursion, but the only other constraint is that a summation operator must exist that accepts the kind of data passed to it.

We define summation for two mono signals and two stereo signals. When no appropriate form of Add can be directly located, as will happen when adding a mono and a stereo signal, the system-provided Add-function attempts to use Coerce to upgrade one of the arguments. Since we have provided a coercion path from mono to stereo, the result is that when adding mono and stereo signals, the mono signal gets upconverted to stereo by Coerce followed by a stereo summation.

The great strength of generics is that functions do not explicitly need to be adapted to a variety of incoming types. If the building blocks or primitives of which the function is constructed can handle a type, so can the function. If the complete set of arithmetic and logical primitives would be implemented for the types Mono and Stereo, then the vast majority of functions, written without any knowledge of these particular types, would be able to transparently handle them.

Generic processing shows great promise once all the possible type permutations present in music DSP are considered. Single or double precision samples? Mono, stereo or multichannel? Real- or complex-valued? With properly designed types, a singular implementation of a signal processor can automatically handle any combination of these.

### 3.2.2 Type Determinism for Performance

Generic programming offers great expressiveness and power to the programmer. However, typeless or dynamically typed languages have a reputation for producing slower code than statically typed languages, mostly due to the extensive amount of run time type information and reflection required to make them work.

To bring the performance on par with a static language, Kronos adopts a rigorous constraint. The output data type of a processing node may only depend on the input data type. This is the principle of type determinism.

As demonstrated in Listing 3, Kronos offers extensive freedom in specifying what is the result type of a function given a certain argument type. However, what is prohibited, based on type determinism, is selecting the result type of a function based on the argument data itself.

Thus it is impossible to define a mixing module that compares two stereo channels, providing a mono output when they are identical and keeping the stereo information when necessary. That is because this decision would be based on data itself, not the type of said data.

While type determinism could be a crippling deficit in a general programming language, it is less so in the context of music DSP. The example above is quite contrived, and regardless, most musical programming environments similarly prevent changes to channel configuration and routing on the fly.

Adopting the type determinism constraint allows the compiler to statically analyze the entire data flow of the program given just the data type of the initial, caller-provided input. The rationale for this is that a signal processing algorithm is typically used to process large streams of statically typed data. The result of a single analysis pass can then be reused thousands or millions of times.

### 3.3 Digital Signal Processing and State

A point must be made about the exclusion of stateful programs, explained in Section 3.1.1. This seems at odds with the established body of DSP algorithms, many of which depend on
state or signal memory. Examples of stateful processes are easy to come by. They include processors that clearly have memory, such as echo and reverberation effects, as well as those with recursions like digital IIR filters.

As a functional language, Kronos doesn’t allow direct state manipulation. However, given the signal processing focus, operations that hide stateful operations are provided to the programmer. Delay lines are provided as operators; they function exactly like the common mathematical operators. A similar approach is taken by Faust, where delay is provided as a built-in operator and recursion is an integrated language construct.

With a native delay operator it is equally simple to delay a signal as it is, for example, to take its square root. Further, the parser and compiler support recursive connections through these operators. The state-hiding operators aim to provide all the necessary stateful operations required to implement the vast majority of known DSP algorithms.

4 Multirate Programming

One of the most critical problems in many signal processing systems is the handling of distinct signal rates. A signal flow in a typical DSP algorithm is conceptually divided into several sections.

One of them might be the set of control signals generated by an user interface or an external control source via a protocol like OSC [Wright et al., 2003]. These signals are mostly stable, changing occasionally when the user adjusts a slider or turns a knob.

Another section could be the internal modulation structure, comprising of low frequency oscillators and envelopes. These signals typically update more frequently than the control signals, but do not need to reach the bandwidth required by audio signals.

Therefore, it is not at all contrived to picture a system containing three different signal families with highly diverging update frequencies.

The naive solution would be to adopt the highest update frequency required for the system and run the entire signal flow graph at that frequency. In practice, this is not acceptable for performance reasons. Control signal optimization is essential for improving the run time performance of audio algorithms.

Another possibility is to leave the signal rate specification to the programmer. This is the case for any programming language not specifically designed for audio. As the programmer has full control and responsibility over the execution path of his program, he must also explicitly state when and how often certain computations need to be performed and where to store those results that may be reused.

Thirdly, the paradigm of functional reactive programming [Nordlander, 1999] can be relied on to automatically determine signal update rates.

4.1 The Functional Reactive Paradigm

The constraints imposed by functional programming also turn out to facilitate automatic signal rate optimization.

Since the output of a functional program fragment depends on nothing but its input, it is obvious that the fragment needs to be executed only when the input changes. Otherwise, the previously computed output can be reused, sparing resources.

This realization leads to the functional reactive paradigm [Nordlander, 1999]. A reactive system is essentially a data flow graph with inputs and outputs. Reactions – responses by outputs to inputs – are inferred, since an output must be recomputed whenever any input changes that is directly reachable by following the data flow upstream.

4.1.1 Reactive Programming in Kronos

Reactive inputs in Kronos are called springs. They represent the start of the data flow and a point at which the Kronos program receives input from the outside world. Reactive outputs are called sinks, representing the terminals of data flow. The system can deduce which sinks receive an update when a particular input is updated.

Springs and Priority

Reactive programming for audio has some special features that need to be considered. Let us examine the delay operators presented in Section 3.3. Since the delays are specified in computational frames, the delay time of a frame becomes the inter-update interval of whatever reactive inputs the delay is connected to. It is therefore necessary to be able to control this update interval precisely.

A digital low pass filter is shown in Listing 4. It is connected to two springs, an audio signal
provided by the argument \(x0\) and an user interface control signal via OSC [Wright et al., 2003]. The basic form of reactive processing laid out above would indicate that the unit delays update whenever either the audio input or the user interface is updated.

However, to maintain a steady sample rate, we do not want the user interface to force updates on the unit delay. The output of the filter, as well as the unit delay node, should only react to the audio rate signal produced by the audio signal input.

**Listing 4:** A Low pass filter controlled by OSC

```c
Lowpass(x0)
{
    cutoff = IO:OSC-Input("cutoff")
    y1 = z-1(0 y0)
    y0 = x0 + cutoff * (y1 - x0)
    Lowpass = y0
}
```

As a solution, springs can be given priorities. Whenever there is a graph junction where a node reacts to two springs, the spring priorities are compared. If they differ, an intermediate variable is placed at the junction and any reaction to the lower priority spring is suppressed for all nodes and sinks downstream of the junction.

When the springs have equal priority, neither is suppressed and both reactions propagate down the data flow. Figure 1 illustrates the reactivity inferral procedure of a graph with several springs of differing priorities.

Typically, priorities are assigned according to the expected update rate so that the highest update rate carries the highest priority.

In the example shown in Listing 5 and Figure 2, an user interface signal adjusts an LFO that in turn controls the corner frequency of a bandpass filter.

There are two junctions in the graph where suppression occurs. Firstly, the user interface signal is terminated before the LFO computation, since the LFO control clock overrides the user interface. Secondly, the audio spring priority again overrides the control rate priority. The LFO updates propagate into the coefficient computations of the bandpass filter, but do not reach the unit delay nodes or the audio output.

**Listing 5:** Mixing user interface, control rate and audio rate signals

```c
Biquad-Filter(x0 a0 a1 a2 b1 b2)
{
    y1 = z-1(0 y0) y2 = z-1(0 y1) x1 = z-1(0 x0) x2 = z-1(0 x1)
    y0 = a0 * x0 + a1 * x1 + a2 * x2 - b1 * y1 - b2 * y2
}
Bandpass-Coefs(freq r amp)
{
    \(a0 \ a1 \ a2\) = (Sqrt(r) 0 Neg(Sqrt(r)))
    \(b1 \ b2\) = (Neg(2 * Crt:cos(freq) * r) r * r)
    Bandpass-Coefs = (a0 a1 a2 b1 b2)
}
Vibrato-Reson(sig)
{
    Use IO
    \(freq = OSC-Input("freq")\)
    \(mod-depth = Crt:pow(OSC-Input("mod-depth") 3)\)
    \(mod-freq = Crt:pow(OSC-Input("mod-freq") 4)\)
    Vibrato-Reson = Biquad-Filter(sig \# Bandpass-Coefs(freq + mod-depth * LFO(mod-freq) 0.95 0.05))
}
```

- **Figure 1:** A reactive graph demonstrating spring priority. Processing nodes are color coded according to which spring triggers their update.
- **Figure 2:** A practical example of a system consisting of user interface signals, coarse control rate processing and audio rate processing.
4.1.2 Explicit Reaction Supression

It is to be expected that the priority system by itself is not sufficient. Suppose we would like to build an envelope follower that converts the envelope of an audio signal into an OSC [Wright et al., 2003] control signal with a lower frequency. Automatic inferral would never allow the lower priority control rate spring to own the OSC output; therefore a manual way to override supression is required.

This introduces a further scheduling complication. In the case of automatic supression, it is guaranteed that nodes reacting to lower priority springs can never depend on the results of a higher priority fragment in the signal flow. This enables the host system to schedule spring updates accordingly so that lower priority springs fire first, followed by higher priority springs.

When a priority inversal occurs, such that a lower priority program fragment is below a higher priority fragment in the signal flow, the dependency rule stated above no longer holds. An undesired unit delay is introduced at the graph junction. To overcome this, the system must split the lower priority spring update into two sections, one of which is evaluated before the suppressed spring, while the latter section is triggered only after the suppressed spring has been updated.

Priority inversal is still a topic of active research, as there are several possible implementations, each with its own problems and benefits.

5 Case Studies

5.1 Reverberation

5.1.1 Multi-tap delay

As a precursor to more sophisticated reverberation algorithms, multi-tap delay offers a good showcase for the generic programming capabilities of Kronos.

Another higher order function, *Curry*, is used to construct a new mapping function. *Curry* attaches an argument to a function. In this context, the single signal *sig* shall be fed to all the delay lines. *Curry* is used to construct a new delay function that is fixed to receive the *curried* signal.

This curried function is then used as a mapping function to the list of delay line lengths, resulting in a bank of delay lines, all of them being fed by the same signal source. The outputs of the delay lines are summed, using *Reduce(Add ...)*. It should be noted that the routine produces an arbitrary number of delay lines, determined by the length of the list passed as the *delays* argument.

5.1.2 Schroeder Reverberator

It is quite easy to expand the multi-tap delay into a proper reverberator. Listing 7 implements the classic Schroeder reverberation [Schroeder, 1969]. Contrasted to the multi-tap delay, a form of the polymorphic *Delay* function that features feedback is utilized.

A third high order function, *Cascade*, is presented, providing means to route a signal through a number of similar stages with differing parameters. Here, the number of allpass comb filters can be controlled by adding or removing entries to the *allpass-params* list.

5.2 Equalization

In this example, a multi-band parametric equalizer is presented. For brevity, the implementation of the function *Biquad-Filter* is not shown. It can be found in Listing 5. The coefficient computation formula is from the widely used Audio EQ Cookbook [Bristow-Johnson, 2011].
This parametric EQ features an arbitrary number of bands, depending only on the size of the lists `freq`, `dBgains` and `qs`. For this example to work, these list lengths must match.

6 Conclusion

This paper presented Kronos, a programming language and a compiler suite designed for musical DSP. Many of the principles discussed could be applied to any signal processing platform.

The language is capable of logically and efficiently representing various signal processing algorithms, as demonstrated in Section 5. As algorithm complexity grows, utilization of advanced language features becomes more advantageous.

While the language specification is practically complete, a lot of implementation work still remains. Previous work by the author on autovectorization and parallelization[Norilo and Laurson, 2009] should be integrated with the new compiler. Emphasis should be placed on parallel processing in the low latency case; a particularly interesting and challenging problem.

In addition to the current JIT Compiler for x86 computers, backends should be added for other compiler targets. Being able to generate C code would greatly facilitate using the system for generating signal processing modules to be integrated into another software package. Targeting stream processors and GPUs is an equally interesting opportunity.

Once sufficiently mature, Kronos will be released as a C-callable library. There is also a command line interface. Various licensing options, including a dual commercial/GPL model are being investigated. A development of PWGLSynth[Laurson et al., 2009] based on Kronos is also planned. Meanwhile, progress and releases can be tracked on the Kronos website[Norilo, 2011].

---

References


Running Csound in Parallel

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

Abstract
One of the largest challenges facing computer scientists is how to harness multi-core processors into coherent and useful tools. We consider one approach to shared-memory parallelism, based on thirty year old ideas from the LISP community, and describe its application to one “legacy” audio programming system, Csound. The paper concludes with an assessment of the current state of implementation.

Parallelism, HiPAC, Csound

In the history of computing we have already seen rather often a mismatch between the available hardware and the state of software development. The current incarnation is currently as bad as it ever has been. While Moore’s Law on the number of transistors on a chip still seems to be accurate, the commonly believed corollary, that processors get faster in the same way, has been shown not to be the case.

Instead we are seeing more processors rather than faster ones. The challenge now is to find ways of using multiple cores effectively to improve the performance of a single program. This paper addresses this issue from a historical perspective and show how 1980s technology can be used, in particular to providing a faster Csound [Boulanger, 2000].

1 The Hardware Imperative
Computing has always had a hardware and a software aspect. It may be usual to view this as a harmonious marriage, but in reality there are a number of tensions in the relationship. Sometimes these conflicts are positive and stimulate innovation, so example the major improvements in compilation following RISC technology.

Usually software follows the hardware, driven by the technological imperative in the words of Robert S. Barton. When I worked with him the late 1970s it was also parallelism that was the cause, as we struggled to provide software to control massively parallel functional computers. I believe that there are many lessons to learn from that attempt to develop parallelism into a usable structure.

2 A Brief History of Parallelism
...and a biased one. Most of my involvement with parallelism has come from a functional or LISP style. For example we proposed a parallel functional machine forty years ago, but this widened following the Barton machine to more LISP-based systems, such as the Bath Concurrent LISP Machine [Fitch and Marti, 1984], and later the developments of simulation to object-based parallelism [Padget et al., 1991]. Much of this work is based on the thesis that users cannot be expected (or trusted) to modify their thinking for parallel execution, and the responsibility needs to be taken by the software translation system that converts the program or specification into an executable form. In particular the compiler analysis can be extended to inform the structure. The particular LISP form of this was described by Marti [1980b; 1980a], and advocated in [Fitch, 1989b] and [Fitch, 1989a]. At the heart of this methodology is determining when different elements of a program (function or object-method) do not interact with each other.

The other aspect of parallelism that needs to be considered is not just if two entities can be run at the same time, but is it worthwhile. All too frequently the overheads of setting up the parallel section is greater that the benefit. The problem is in the general case, to know the cost of a computation is do the computation. This has led a number of compilation systems that perform testing runs of the program in order to estimate the performance. An alternative is to make a compile time estimate [Fitch and Marti, 1989]. Later, in section 5.5, we will make some use of both these techniques.

Parallelism has been an issue in computing for many years, and seems to re-emerge every twenty years as important. It is contended that
we need to be mindful of what worked and what did not (and why) from the past.

3 Ab Initio Parallelism

Considering just the area of audio processing there is a body of existing code, albeit synthesis tools, analysis, mastering etc. The obvious alternative to adapting these to a parallel machine would be to start again, and redesign the whole corpus with an eye on parallelism ab initio. The problem with this approach is the volume of software, and the commitment by users to these programs. The field of computer music has already suffered from software loss without inducing a whole new wave. For this reason the work advocated here has the preservation of the syntax and semantics of existing systems at its heart. This is indeed in line with the longstanding policy of Csound, never to break existing pieces.

Similarly dependence on user annotation is not the way forward. Skilled programmers are not noted for being good at the use of annotations, and we really should not expect our users, most of whom are musicians rather than programmers, to take this responsibility.

It should however be recognised that there have been attempts to recreate audio processing in parallel. Notably there was the 170 Transputer system that extended Csound into real-time [Bailey et al., 1990], which had hardware related problems of heat. A different approach was taken in [Kirk and Hunt, 1996] which streamed data through a distributed network of DSP processing units, to create Midas. Both of these have finer-grained distribution that the system presented here.

4 High Performance Computing

The mainstream projects in parallel processing are currently focused on HPC (High Performance Computing) which has come to mean matrix operations, using systems like MPI [Gropp et al., 1996]. The major interest is in partitioning of the matrix in suitable sizes for cache sizes, distribution between multicore and packet sizes for non-shared memories. Most of this is not directly applicable in audio processing, where low latency is such an important requirement.

This mismatch led to the promotion of High Performance Audio Computing in [Dobson et al., 2008], to draw attention to the differences, and in particular the latency. The other point about which I am concerned is that most of our users have commodity computers, usually with two or more cores, but not a cluster. The parallelism attempt in this paper is for the majority community rather than the privileged HPC users.

5 Towards a Parallel Csound

Csound [Vercoe, 1993] has a long and venerable history. It was written in the 1980s, and despite a significant rewrite ten years ago it remains grounded in the programming style of that period. As a member of the Music V family the system separates the orchestra from the score; that is it distinguishes the description of the sound from the sequence of notes. It also has a control rate, usually slower than the sampling rate, at which new events start, old one finish or control signals are sensed. Instruments are numbered by integers\(^1\), and these labels play an important part in the Csound semantics. During any control cycle the individual instrument instances are calculated in increasing numerical order. Thus if one instrument is controlling another one, it will control the current cycle if it is lower numbered than the target, or the next cycle if higher. The main control loop can be described as

\[
\text{until end of events do}
\]
\[
\text{deal with notes ending}
\]
\[
\text{sort new events onto instance list}
\]
\[
\text{for each instrument in instance list}
\]
\[
\text{calculate instrument}
\]

In order to introduce parallelism into this program the simplest suggestion is to make the “for each” loop run instances in parallel. If the instruments are truly independent then this should work, but if they interact in any way then the results may be wrong.

This is essentially the same problem that Marti tackled in his thesis. We can use code analysis techniques to determine which instruments are independent. Concentrating initially on variables, it is only global variables that are of concern. We can determine for each instrument the sets of global variables that are read, written, or both read and written, the last case corresponding to sole use, while many can read a variable as long as it is not written.

There is a special case which needs to be considered; most instruments add into the output

\(^1\) They can be named, but the names are mapped to integers
bus, but this is not an operation that needs ordering (subject to rounding errors), although it may need a mutex or spin-lock. The language processing can insert any necessary protections in these cases.

This thus gives a global design.

5.1 Design
The components in the design of parallel Csound are first a language analysis phase that can determine the non-local environment of each instrument specification. This is then used to organise the instance list into a DAG, where the arcs represent the need to be evaluated before the descendents. Then the main control operation becomes

until end of events do
    deal with notes ending
    add new events and reconstruct the DAG
until DAG empty
    foreach processor
        evaluate a root from DAG
    wait until all processes finish

We now consider the components of this.

5.2 Compiler
The orchestra language of Csound is basically simple, rather like an assembler with the operations being a range of DSP functions. The language processing in the usual Csound is simple, with a simple ad hoc lexer and hand-written parsing. It was a wish of the Csound5 re-write to produce a new parser, based on flex/bison, so things like functions with more than one argument could be introduced. A draft such parser was in construction while the major changes were made, as described in [ffitch, 2005]. The needs of parallelism added impetus to the new parser project, and it was largely completed by Yi, and is currently being subjected to extreme testing. The new parser was extended to construct the dependency information, and to add necessary locks (see section 5.4).

A simple example of the analysis for a simple orchestra (figure 1) can be seen in figure 2, listing the variables read, written and exclusive. The additional field is to indicate when the analysis has to assume that it might read or write anything.. In our simple example instrument 1 is independent of both instruments 2 and 3 (apart from the out opcode. On the other hand instrument 2 must run to completion before instrument 3, as it gives a value to a global read by instrument 3. Any number of instrument 3 instances can run at the same time but instances of instrument 2 need some care, as we must maintain the same order as a single threaded system.

This dependency analysis is maintained, and used in the DAG.

5.3 DAG
In the main loop to determine the execution of the instrument instances the decisions are determined by maintaining a DAG, the roots of which are the instruments that are available. In the case of our example the raw picture this is shown in figure 3. This DAG is consumed on each control cycle. Naively one must retain the original structure before consumption as it will be needed on the next cycle. This is complicated by the addition and deletion of notes. We investigated DAG updating algorithms but dynamic graphs is a complex area [Holm et al., 2001] and we are led to reject the allure of $O(\log(\log(n))$ algorithms; this complexity led us instead to a recreation of the DAG when there are changes. This is a summary of many experiments, and is one of the major bottlenecks in the system.

The whole dispatcher is very similar to an instruction scheduling algorithm such as [Muchnick and Gibbons, 2004] augmented by some VLIW concepts; it is in effect a bin-packing problem.

5.4 Locking and Barriers
The actually parallel execution is achieved with the POSIX pthreads library. One thread is designated as the main thread, and it is that one that does the analysis and setup. There is a barrier set at the start of each control cycle so after the setup all threads are equal and try to get work from the DAG. This is controlled by a mutex so as not to compromise the structure. When an instrument-cycle finishes

```
instr 1
    a1 oscil p4, p5, 1
    out a1
endin
instr 2
    gk oscil p4, p5, 1
endin
instr 3
    a1 oscil gk, p5, 1
    out a1
endin
```

Figure 1: A simple Orchestra.
there is a further entry to the DAG via a mutex to remove the task and possibly release others. When there is no work the threads proceed to the barrier at the end. The master thread re-asserts itself to prepare the next cycle. The mutex can be either POSIX mutexes or spinlocks, and we have experimented with both.

The other use of mutex/locks is in global variable updating. If a variable is added into, with a statement like

\[ gk1 = gk1 + 1 \]

then there is no need for exclusive use of the variable except during the updating. The compiler creates locks for each such occurrence and introduces calls to private opcodes (not available to users) to take and release the lock. There are other similar types of use that are not yet under the compiler control but could be (see section 5.6).

### 5.5 Load Balancing

A major problem in any non-synchronous parallel execution system is balancing the load between the active processes. Ideally we would like the load to be equal but this is not always possible. Also if the granularity of the tasks is too small then the overhead of starting and stopping a thread dominates the useful work. The initial system assumes that all instruments take about the same time, and that time is much larger than the setup time.

There is code written and tested but not yet deployed to collect instances together to ensure larger granularity. This needs raw data as to the costs of the individual unit generators. This data can come from static analysis (as in [Fitch and Marti, 1989]), or from running the program in a testing mode. In the case of Csound the basic generators are often constant in time, or we may assume some kind of average behaviour. We have been using valgrind on one system (actually Linux i386) to count instructions. With a little care we can separate the three components of cost; initialisation, instructions in each k-cycle and those obeyed on each audio sample. In the case of some of these opcodes the calculation do not take account of the time ranges due to data dependence, but we hope an average time is sufficient. These numbers, a small selection of which are shown in table 1, can be used for load balancing.

### 5.6 Current Status

The implementation of the above design, and many of its refinements are the work of Wilson[2009]. His initial implementation was on OSX and tested with a twin-core processor. The version currently visible on Sourceforge is a cleaned up version, with some of the experimental options removed and a more systematic use of mutexs and barriers.

The parser is enhanced to generate the dependency information and to insert small exclusion zones around global variable updates. The instrument dispatch loop has been rewritten along the lines in section 5, with the necessary DAG manipulations. There is code for load balancing but until the raw data is complete it is not deployed, but it has been tested.

Some opcodes, notably the `out` family have local spin locks, as they are in effect adding into a global variable. There are similar struc-

<table>
<thead>
<tr>
<th>Opcode</th>
<th>init</th>
<th>Audio</th>
<th>Control</th>
</tr>
</thead>
<tbody>
<tr>
<td>table.a</td>
<td>93</td>
<td>23.063</td>
<td>43.998</td>
</tr>
<tr>
<td>table.k</td>
<td>93</td>
<td>0</td>
<td>45</td>
</tr>
<tr>
<td>butterlp</td>
<td>9</td>
<td>29.005</td>
<td>4</td>
</tr>
<tr>
<td>butterhi</td>
<td>19</td>
<td>30.000</td>
<td>35</td>
</tr>
<tr>
<td>butterbp</td>
<td>20</td>
<td>30</td>
<td>71</td>
</tr>
<tr>
<td>bilbar</td>
<td>371.5</td>
<td>1856.028</td>
<td>86</td>
</tr>
<tr>
<td>ags</td>
<td>497</td>
<td>917.921</td>
<td>79475.155</td>
</tr>
<tr>
<td>oscil.kk</td>
<td>69</td>
<td>12</td>
<td>47</td>
</tr>
<tr>
<td>oscili.kk</td>
<td>69</td>
<td>21</td>
<td>49</td>
</tr>
<tr>
<td>reverb</td>
<td>6963.5</td>
<td>77</td>
<td>158</td>
</tr>
</tbody>
</table>

Table 1: Costs of a few opcodes.


<table>
<thead>
<tr>
<th>Sound</th>
<th>ksmps</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Xanadu</td>
<td>1</td>
<td>31.202</td>
<td>39.291</td>
<td>42.318</td>
<td>43.043</td>
<td>48.304</td>
</tr>
<tr>
<td>Xanadu</td>
<td>10</td>
<td><strong>18.836</strong></td>
<td>19.901</td>
<td>20.289</td>
<td>21.386</td>
<td>22.485</td>
</tr>
<tr>
<td>Xanadu</td>
<td>100</td>
<td>16.023</td>
<td>17.413</td>
<td>16.999</td>
<td>16.545</td>
<td><strong>15.884</strong></td>
</tr>
<tr>
<td>Xanadu</td>
<td>300</td>
<td>17.159</td>
<td>16.137</td>
<td>15.141</td>
<td>15.723</td>
<td><strong>14.905</strong></td>
</tr>
<tr>
<td>CloudStrata</td>
<td>1</td>
<td><strong>173.757</strong></td>
<td>191.421</td>
<td>211.295</td>
<td>214.516</td>
<td>261.238</td>
</tr>
<tr>
<td>CloudStrata</td>
<td>10</td>
<td>89.406</td>
<td><strong>80.998</strong></td>
<td>94.023</td>
<td>110.170</td>
<td>98.187</td>
</tr>
<tr>
<td>CloudStrata</td>
<td>100</td>
<td>85.966</td>
<td>86.114</td>
<td><strong>81.909</strong></td>
<td>83.258</td>
<td>85.631</td>
</tr>
<tr>
<td>CloudStrata</td>
<td>300</td>
<td>87.153</td>
<td>76.045</td>
<td>79.353</td>
<td>78.399</td>
<td><strong>74.684</strong></td>
</tr>
<tr>
<td>CloudStrata</td>
<td>900</td>
<td>82.612</td>
<td>76.434</td>
<td><strong>64.368</strong></td>
<td>76.217</td>
<td>74.747</td>
</tr>
<tr>
<td>trapped</td>
<td>1</td>
<td><strong>20.931</strong></td>
<td>63.492</td>
<td>81.654</td>
<td>107.982</td>
<td>139.334</td>
</tr>
<tr>
<td>trapped</td>
<td>100</td>
<td><strong>1.388</strong></td>
<td>1.810</td>
<td>1.928</td>
<td>2.167</td>
<td>2.612</td>
</tr>
<tr>
<td>trapped</td>
<td>300</td>
<td>1.319</td>
<td><strong>1.181</strong></td>
<td>1.205</td>
<td>1.386</td>
<td>1.403</td>
</tr>
<tr>
<td>trapped</td>
<td>900</td>
<td>1.236</td>
<td><strong>1.025</strong></td>
<td>1.085</td>
<td>1.091</td>
<td>1.112</td>
</tr>
</tbody>
</table>

Table 2: Performance figures; time in seconds.

The performance figures are perhaps a little disappointing, but they do show that it is possible to get speed improvements, and more work on the load balance could be useful.

6 Performance

All the above is of little point if there is no performance gain. It should be noted that we are concerned here with time to completion, and not overall efficiency. The need for parallelism here is to provide greater real-time performance and quicker composition.

The initial Wilson system reported modest gains on his dual core machine; 10% to 15% on a few examples with a top gain of 35%. The developed system has not seen such dramatic gains but they are there.

Running a range of tests on a Core-7 quad-core with hyper-threads it was possible to provide a wide range of results, varying the number of threads and the control rate. These are presented in figure 2 with the fastest time being in bold face. As the control rate decreases, corresponding to an increase in ksmps, the potential gain increases. This suggests that the current system is using too small a granularity and the collecting of instruments into larger groups will give a performance gain. It is clearly not always a winning strategy, but with the more complex scores there is a gain when ksmps is 100. Alternatively one might advise large values of ksmps, but that introduces quantisation issues and possibly zipped noise.

The performance figures are perhaps a little disappointing, but they do show that it is possible to get speed improvements, and more work on the load balance could be useful.

7 Conclusions

A system for parallel execution of the “legacy” code in Csound has been presented, that works at the granularity of the instrument. The indications of overheads for this scheme suggest that we need to collect instruments into groups to increase granularity. The overall design, using compiler technology to identify the places where parallelism cannot be deployed. The real cost of the system is in the recreation of the DAG and its consumption, and all too often this overhead swamps the gain from parallelism.

The remaining work that is needed before this can enter the main stream is partially the completion of the new parser, which is nearly done, and dealing with the other places in Csound where data is global. As well as the busses men-
tioned earlier there are global uses of tables. In the earlier versions of Csound tables were immutable, but recent changes has nullified this. The load balancing data needs to be collected. Currently this is a tedious process with much human intervention, and it needs to be scripted, not only to create the initial state but to make adding new opcodes into the parallel version.

Despite the problems identified in this paper parallel Csound is possible via this methodology. I believe that the level of granularity is the correct one, and with more attention to the DAG construction and load balancing it offers real gains for many users. It does not require specialist hardware, and can make use of current and projected commodity systems.

8 Acknowledgements

My thanks go to the many people who have contributed to the work here, In particular Jed Marti for the initial ideas, Arthur Norman for years of discussions, Steven Yi for the new parser and Chris Wilson for bringing it to reality; and the Csound developer community who encouraged me to continue.

References


New Clothes for Pure Data

IOhannes, ZMÖLNIG
Institute of Electronic Music and Acoustics
University of Music and Performing Arts
Graz
Austria
zmoelnig@iem.at

Abstract

After two years, a new version of Pure Data has finally been released. The main new feature is a complete refactoring of the Graphical User Interface. While on the surface things still look like they used to look in the last 10 years, the new implementation gives hope for more radical improvements in the near future.

The refactoring was lead by Hans-Christoph Steiner (of Pd-extended fame), with some help by the author of this paper and a few other contributors.

Keywords

Pure Data, refactoring, user interface

1 Introduction

Among the freely available computer music environments, Pure Data has a long and successful history. One of it’s major strengths - that presumably makes it so appealing to newcomers - is it’s graphical nature, allowing the users to create complex software in a way that feels more like “painting” rather than writing technical instructions.

Interestingly though, the user interface has been among the most neglected parts within the main development line of Pure Data since it’s beginning.

This can be explained by the development model of Pd, which is very centralized and focused on its original author. Unlike other Open Source projects, where a group of volunteers are often expected to contribute as peers (though this expectation if often invalidated, as has been shown e.g. by [1]), the canonical version of Pd (aka Pd-vanilla) is developed and maintained by a single person.

This development model is exemplified by the implicit rule of the Pure Data repository, where only a single person is allowed write access to the directory holding the code for Pd-vanilla.\footnote{This rule is a social agreement, rather than technically enforced. The reason is mainly that sourceforge, where the repository is hosted, does not allow to setup fine grained access controls to SVN repositories.} Con-

1 or rather took, as it’s development has now stalled for several years
Pd-vanilla.

So when Miller Puckette started to publish the first test code for the 0.43 version of Pd in early 2009, Hans-Christoph Steiner saw the opportunity to incorporate a refactored GUI version in this next release.

2 Pd: A Client-Server model?

Pd consists of two parts: the Pd-GUI, handling user-interaction and written in Tcl/Tk, and Pd-CORE, the interpreter and sound-synthesis engine, that is written in C. The two parts communicate via a network socket, theoretically allowing to run Pd-CORE and Pd-GUI on different machines, with the CORE being busy calculating musical structures and feeding the soundcard, and the GUI doing all the user interaction.

In practise, the two are closely coupled together, with most of the GUI interaction handled on the server (Pd-CORE) side. E.g. when the user clicks the mouse button, Pd-GUI will catch this event and send it to Pd-CORE, which calculates, which graphical element is affected by the mouse click, triggers any programmatic consequences (e.g. send a “bang” message, since it turns out that the user clicked on \texttt{[bang()]}, and then tells the GUI, how it should be updated (e.g. it sends a message to make the lines surrounding \texttt{[bang()] become thicker, and after a short amount of time to become thinner again, thus giving a “flashing” visual feedback).

Therefore, Pd adheres more to a ruler/slave model, than to a client/server model.

As the author has argued in the past[4], this has advantages when it comes to collaborative editing, as it is easy to intercept the network stream and distribute it to several GUI-clients, which will all get synchronously updated.

The main drawback of this tight coupling is, that the network socket gets easily congested by the high amount of data that is sent (rather than telling the surrounding lines of the \texttt{[bang()] GUI-element to change their line width two times, it could be sufficient to tell the GUI-element to “flash”), and that the DSP-engine has to deal with non real-time critical data (like finding out which object-box the user wants to interact with) instead of focusing on low latency signal processing.

What makes things worse is, that Pd-CORE will actually send Tcl/Tk code over the network, making it virtually impossible to replace the current implementation by one that is written in another language.

3 Cleaning up the code base

One of the core requirements for the Pd-GUI-rewrite to be included into Pd-vanilla was to not touch the server side of things (the C-implementation of the Pd-CORE). This obviously limited, what could be achieved by any changes made.

It was thus decided, to first start a refactoring of the entire Tcl/Tk on the GUI side. Until now, the GUI was implemented in a single file \texttt{u_main.tk}, where new features were added in a rather unstructured way. In the past years, this single file has grown to several thousand lines of code. What’s more, there was virtually no documentation, making it hard to read and almost impossible to maintain.\footnote{The state of the code was one of the major reasons, why there were no objections to doing a complete rewrite, breaking the paradigm of only accepting small incremental changes to the code base.}

In order to make the code readable, it was thus split into a number of files according to functional units. Additionally, code was structured using namespaces, in order to minimise the potential for name clashes, in case the code should turn obfuscated again.

3.1 Stand alone

Traditionally, Pd-GUI was implemented as a minimal C-program, that would embed a Tcl-interpreter which in turn runs the script implementation of the GUI. This C-program would also supply the network communication, so the GUI can communicate with the Pd-CORE.

This is a somewhat overly complicated approach, given that there are good standalone Tcl/Tk interpreters available (most prominent: \texttt{wish}), and Tcl has built-in capabilities for networking anyhow.

Another drawback with using the wrapper binary approach is that it adds additional dependencies needed to embed Tcl, for no apparent benefit.

3.2 Talking aloud

One thing that users will immediately benefit from, is an improved logging mechanism. Pd prints all warnings, errors and status messages into its status console. While one could normally select the level of verbosity at startup, there used to be no way to change the verbosity once Pd was running. This is particularly annoying if a problem that needs more verbose output only appears after some time. The new
console interface allows, to change the verbosity at runtime, possibly displaying messages that were filtered out before. No messages are lost (unless Pd is told to forget about them explicitly). All messages are (optionally) tagged with an ID belonging to the emitting object, so an error can be tracked to its source by clicking on the error-message, even if other errors happened afterwards.

3.3 Prompt for user-input
An invaluable tool while developing and debugging proved to be an interactive Tcl/Tk prompt, that allows to execute Tcl code within the context of a running Pd session. In order to keep confusion low, in the released version of Pd the Tcl prompt has been hidden and must be activated by the user via the menu.

3.4 Going international...
While Pd as a language contains a number of English based keywords which cannot be appropriately translated to various human languages, it is no longer state of the art to provide localisation support for internationalisation, by the means of the msgcat package, that manages a multi-lingual catalogue of strings, that can be used to provide localised menu-items.

In the course of refactoring the code, adding support for i18n was trivial, so users can now select from menus in Russian or Hindi, if they prefer to.

Tcl/Tk is fully UTF-8 compliant, so it also makes sense to add UTF-8 support to the entirety of Pd. Since most parts of Pd are agnostic to the used character set, only a few routines that deal with characters transmitted from/to the GUI needed to be adapted, enabling users to create comments (and objects, if they feel inclined to) in their native alphabets.

4 Customising your interface
The refactoring of the GUI as described in Section 3 brings some minor improvements of existing features, while making the code more maintainable.

A really new feature has been introduced by the advent of *GUI-plugins*. These plugins can be seen as what externals are to the Pd-CORE, as they allow new features to be added without having to incorporate them into the upstream distribution.

On startup, Pd's search paths are scanned for directories with a suffix `-plugin`. Any files therein which have a suffix `-plugin.tcl` are loaded in the interpreter, automatically injecting the code.

Due to the nature of the Tcl language, virtually anything can be done this way, including monkey patching\(^4\) the original sources, thus completely redefining the current functionality of the GUI.

More often however, GUI-plugins are expected to simply add hooks to otherwise unused events. For instance, one could define shortcuts for often used actions. Listing 1 implements a simple plugin that binds the key-combination `Control-i` to create an `[inlet]` object, and `Control-o` to create an `[outlet]` object.

Other plugins could change the colour-scheme\(^5\) of Pd, or implement a KIOSK-mode to prevent people from quitting Pd in public installations.

A still small but growing list of publicly available GUI-plugins can be found at [http://puredata.info/downloads](http://puredata.info/downloads)

Of course there is a catch to all this: there is almost no (stable) API defined to be used by plugin authors (unlike with Pd-externals).

5 Towards a more complete separation of CORE and GUI
Apart from the new GUI-plugin feature, the GUI-rewrite should, in our opinion, mainly be seen as a paving of the way to a more complete separation between the DSP-part of Pd and it's visual representation.

While Pd has some hacks to utilise multiple cores of modern CPUs, it is still essentially a single-threaded application. This makes it even less tolerable, that the real time critical DSP-process should bother with details that are only relevant to the GUI side, while the GUI process only has to process pre-cut and pre-chewed data and is thus mostly idling.

Currently, the CORE still sends Tcl/Tk code to the GUI, for example telling it to draw a white rectangle, three small black rectangles and the letter 'f' (Listing 2), rather than telling it to display an object `[f]` with no arguments and with 2 inlets and 1 outlet (Listing 3), ideally in a format that is native to Pd rather than to Tcl.

A proposal for a FUDI inspired syntax for the communication between Pd-CORE and Pd-GUI

\(^4\) replacing existing functions by user-defined ones
\(^5\) the white background is often disturbing in otherwise dark concert halls during traditional computer music performances.
could look like:

- creating objects:
  `<windowid> obj <objid> <x> <y>
   {<objname>} {<obj args...>}
   {<#inlets> <#outlets>}

- creating message boxes:
  `<windowid> msg <msgid> <x> <y>
   {<selector>} {<msg args...>}

- connecting:
  `connect <obj0id> <outletid> <obj1id>
   <inletid> <type>

- moving:
  `<objid> move <x> <y>

- deleting:
  `<objid> delete

The entire code for drawing the object/message/connection resides on the GUI side. Since no (Tcl) language specific code is sent over the socket, the GUI could be implemented in any language.

6 Conclusions

The user interface part of Pure Data has been completely refactored for the current release of Pd (0.43). Due to the nature of refactoring this does not mean, that things have dramatically changed from a user's point of view. However, the new code should make maintenance of the GUI much easier in the future, allowing to go one step further towards a better separation between the number crunching DSP process and it's visual representation within the next release(s) of Pd.

In the meantime, developers are given an easy possibility to add new interface features by the means of GUI plugins.

7 Acknowledgements

All thanks go to Hans-Christoph Steiner, who initiated the refactoring of the user interface, and convinced Miller Puckette to include it in the new release of Pure Data.

References


Listing 1: simple GUI-plugin

```tcl
proc create_object {window x y args} {
    set mytoplevel [winfo toplevel $window]
    if {([winfo class $mytoplevel] == "PatchWindow")} {
        ::pd_connect::pdsend "$mytoplevel obj $x $y $args"
    }
}
bind all <Control-i> {create_object %W %x %y inlet}
bind all <Control-o> {create_object %W %x %y outlet}
```

Listing 2: Tcl code sent to Pd-gui to display [f]

```tcl
CNVID create line 207 117 229 117 229 134 207 134 207 117
    -dash "" -tags [list RCTID obj]
CNVID create rectangle 207 133 214 134
    -tags [list O0ID outlet]
CNVID create rectangle 207 117 214 118
    -tags [list I0ID inlet]
CNVID create rectangle 222 117 229 118
    -tags [list I1ID inlet]
pdtk_text_new CNVID {TXTID obj text}
    209.000000 119.000000 {f} 10 black
```

Listing 3: (potential) Pd-code sent to Pd-gui to display [f]

```tcl
CNVID obj OBJID 207 117 {f} {} {2 1};
```
Abstract
Supernova is a new implementation of the SuperCollider server scsynth, with a multi-threaded audio synthesis engine. To make use of this thread-level parallelism, two extensions have been introduced to the concept of the SuperCollider node graph, exposing parallelism explicitly to the user. This paper discusses the semantic implications of these extensions.

Keywords
SuperCollider, Supernova, parallelism, multi-core

1 Introduction
These days, the development of audio synthesis applications is mainly focussed on off-the-shelf hardware and software. While some embedded, low-power or mobile systems use single-core CPUs, most computer systems which are actually used in musical production use multi-core hardware. Except for some netbooks, most laptops use dual-core CPUs, single-core workstations are getting rare.

Traditionally, audio synthesis engines are designed to use a single thread for audio computation. In order to use multiple CPU cores for audio computation, this design has to be adapted by parallelizing the signal processing work.

This paper is divided into the following sections: Section 2 describes the SuperCollider node graph which is the base for the parallelization of Supernova. Sections 3 introduces the Supernova extensions to SuperCollider with a focus on their semantic aspects. Section 4 discusses different approaches of other parallel audio synthesis systems.

2 SuperCollider Node Graph
SuperCollider has a distinction between instrument definitions, called SynthDefs, and their instantiations, called Synths. Synths are organized in groups, which are linked lists of nodes (synths or nested groups). The groups therefore form a hierarchical tree data structure, the node graph with a group as root of the tree.

Groups are used for two purposes. First, they define the order of execution of their child nodes, which are evaluatated sequentially from head to tail using a depth-first traversal algorithm. The node graph therefore defines a total order, in which synths are evaluated. The second use case for groups is to structure the audio synthesis and to be able to address multiple synths as one entity. When sending a node command to a group it is applied to all its child nodes. Groups can be moved inside the node graph like a single node.

2.1 Semantic Constraints for Parallelization
The node graph is designed as data structure for structuring synths in a hierarchical manner. Traversing the tree structure is used to determine the order of execution, but it does not contain any notion of parallelism. While synths may be able to run in parallel, it is impossible for the synthesis engine to know this in advance. Synths do not communicate with each other directly, but instead they use global busses to exchange audio data. So any automatic parallelization would have to create a dependency graph depending on the access pattern of synths to global resources. The current implementation lacks a possibility to determine, which global resources are accessed by a synth. But even if it would be possible, the resources which are accessed by a synth are not constant, but can change at control rate or even at audio rate. Introducing automatic parallelization would therefore introduce a constant overhead and the parallelism would be limited by the granularity in which resource access could be predicted by the runtime system.

Using pipelining techniques to increase the throughput would only be of limited use, ei-
The synthesis engine dispatches commands at control rate and during the execution of each command, it needs to have a synchronized view of the node graph. In order to implement pipelining across the boundaries of control rate blocks, a speculative pipelining with a rollback mechanism would have to be used. This approach would only be interesting for non-realtime synthesis. Introducing pipelining inside control-rate blocks would only be of limited use, since control rate blocks are typically small (usually 64 samples). Also the whole unit generator API would need to be restructured, imposing considerable rewrite effort.

Since neither automatic graph parallelization nor pipelining a feasible, we introduced new concepts to the node graph in order to expose parallelism explicitly to the user.

### 3 Extending the SuperCollider Node Graph

To make use of thread-level parallelism, SuperNova introduces two extensions to the SuperCollider node graph. This enables the user to formulate parallelism explicitly when defining the synthesis graph.

#### 3.1 Parallel Groups

The first extension to the node graph are **parallel groups**. As described in Section 2, groups are linked lists of nodes which are evaluated in sequential order. Parallel groups have the same semantics as groups, but with the exception, that their child nodes are not ordered. This implies that they can be executed in in separate threads. This concept is similar to the `SEQ` and `PAR` statements, which specify blocks of sequential and parallel statements in the concurrent programming language [Hyde, 1995].

Parallel groups are very easy to use in existing code. Especially for additive synthesis or granular synthesis with many voices, it is quite convenient to instantiate synths inside a parallel group, especially since many users already use groups for these use cases in order to structure the synthesis graph. For other use cases like polyphonic phrases, all independent phrases could be computed inside groups, which are themselves part of a parallel group.

Listing 1 shows a simple example, how parallel groups can be used to write a simple polyphonic synthesizer of 4 synths, which are evaluated before a effect synth.

```plaintext
var generator_group, fx;
generator_group = ParGroup.new;
4.do {
    Synth.head(generator_group, \myGenerator)
};
fx = Synth.after(generator_group, \myFx);
```

#### 3.2 Satellite Nodes

Parallel groups have one disadvantage. Each member of a parallel group is still synchronized with two other nodes, it is executed after the parallel group’s predecessor and before its successor. For many use cases, only one relation is actually required. Many generating synths can be started without waiting for any predecessor, while synths for disk recording or peak followers for GUI applications can start running after their predecessor has been executed, but no successor has to wait for its result.

These use cases can be formulated using **satellite nodes**. These satellite nodes, are nodes which are in dependency relation with only one reference node. The resulting dependency graph has a more fine-grained structure, compared to a dependency graph, which is only using parallel groups.

Listing 2 shows, how the example of Listing 1 can be formulated with satellite nodes under the assumption, that none of the generator synths depends on the result of any earlier synth. Instead of packing the generators into a parallel group, they are simply defined as satellite predecessors of the effect synth.

It is even possible to prioritize dependency graph nodes to optimize graph progress. In order to achieve the best throughput, we need to ensure, that there are always at least as many parallel jobs available as audio threads. To ensure this, a simple heuristic can be used, which always tries to increase the number of jobs, that are actually runnable.

- Nodes with successors have a higher priority than nodes without.
- Nodes with successors early in the dependency graph have a high priority.

These rules can be realized with a heuristic that splits the nodes into three categories.
with different priorities: ‘regular’ nodes having the highest priority, satellite predecessors with medium priority and satellite successors with low priority. While it is far from optimal, this heuristic can easily be implemented with three lock-free queues, so it is easy to use it in a real-time context.

3.3 Common Use Cases & Library Integration

The SuperCollider language contains a huge class library. Some parts of the library are designed to help with the organization of the audio synthesis like the pattern sequencer library or the Just-In-Time programming environment JITLIB.

The pattern sequencer library is a powerful library, that can be used to create sequences of Events. Events are dictionaries, which can be interpreted as musical events, with specific keys having predefined semantics as musical parameters [McCartney, ]. Events may contain the keys \texttt{group} and \texttt{addAction}, if present are used to specify the position of a node on the server. With these keys, both parallel groups and satellite nodes can be used from a pattern environment. In many cases, the pattern sequencer library is used in a way that the created synths are mutually independent and do not require data from other synths. In these cases both parallel groups and satellite predecessors can safely be used.

The situation is a bit different with JITLIB. When using JITLIB, the handling of the synthesis graph is completely hidden from the user, since the library wraps every synthesis node inside a proxy object. JITLIB nodes communicate with each other using global busses. This approach makes it easy to take the output of one node as input of another and to quickly reconfigure the synthesis graph. JITLIB therefore requires a deterministic order for the read/write access to busses, which cannot be guaranteed when instantiating nodes in parallel groups, unless additional functionality is implemented to read always those data, which are written during the previous cycle. Satellite nodes cannot be used to model the data flow between JITLIB nodes, since they cannot be used to formulate cycles.

4 Related Work

During the last few years, support for multicore audio synthesis has been introduced into different systems, that impose different semantic constraints.

4.1 Max/FTS, Pure Data & Max/MSP

One of the earliest computer-music systems, the Ircam Signal Processing Workstation (ISPW) [Puckette, 1991], used the Max dataflow language to control the signal processing engine, running on a multi-processor extension board of a NeXT computer. FTS was implementing explicit pipelining, so patches could be defined to run on a specific CPU. When audio data was sent from one CPU to another, it was delayed by one audio block size.

Recently a similar approach has been implemented for Pure Data [Puckette, 2008]. The \texttt{pd~} object can be used to load a subpatch as a separate process. Moving audio data between parent and child process adds one block of latency, similar to FTS. Therefore it is not easily possible to modify existing patches without changing the semantics, unless a latency compensation is taken into account.

The latest versions of Max/MSP contains a \texttt{poly~} object, which can run several instances of the same abstraction in multiple threads. However, it is not documented, if the signal is delayed by a certain amount of samples or not. And since only the same abstraction can be distributed to multiple processors, it is not a general purpose solution.

An automatic parallelization of max-like systems is rather difficult to achieve, because max- graphs have both explicit dependencies (the signal flow) and implicit ones (resource access). In order to keep the semantics of the sequential program, one would have to introduce implicit dependencies between all objects, which access a specific resource.

4.2 CSound

Recent versions of CSound implement automatic parallelization in order to make use of multicore hardware [ffitch, 2009]. This is feasible, because the CSound parser has a lot of
knowledge about resource access patterns and the instrument graph is more constrained compared to SuperCollider. Therefore the CSound compiler can infer many dependencies automatically, but if this is not the case, the sequential implementation needs to be emulated.

The automatic parallelization has the advantage, that existing code can make use of multicore hardware without any modifications.

4.3 Faust
Faust supports backends for parallelization, an open-mp based code generator and a custom work-stealing scheduler [Letz et al., 2010]. Since Faust is only a signal processing language, with little notion of control structures. Since Faust is a compiled language, it cannot be used to dynamically modify the signal graph.

5 Conclusions
The proposed extensions to the SuperCollider node graph enable the user to formulate signal graph parallelism explicitly. They integrate well into the concepts of SuperCollider and can be used to parallelize many use cases, which regularly appear in computer music applications.

References


Abstract
This paper is to introduce a realisation of Imitative Additive Synthesis in Csound, which can be employed for the realtime analysis of the spectrum of a sound suitable for additive synthesis. The implementation described here can be used for analysing, re-playing and modifying sounds in a live situation, as well as saving the analysis results for further use.

Keywords
Spectral analysis, sound synthesis, signal processing, realtime resynthesis.

1 What is Imitative Additive Synthesis?
Additive Synthesis is known as the method of synthesizing sound by single sinusoids. Based on a fundamental frequency and a number of sinusoids, the main parameters are
1. a frequency multiplier, and
2. a relative amplitude
for each partial. For instance, the well-known additive synthesized bell by Jean-Claude Risset has the values listed in table 1.¹

In this case, the frequency/amplitude values were derived from the analysis of a natural sound. This is an example of what I call "Imitative Additive Synthesis", as opposed to creating spectra which do not exist anywhere in the non-electronic world of sound.

In real-world sounds, a table like the one given above is just a snapshot. The actual amplitudes vary all the time. Or, in other words, each partial has its own envelope. It is again Jean-Claude Risset who has described this early, when he analyzed a trumpet tone. The visualization can be done in a three-dimensional way like this:

<table>
<thead>
<tr>
<th>Partial Number</th>
<th>Frequency multiplier</th>
<th>Amplitude multiplier</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.56</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0.56 + 1 Hz</td>
<td>0.67</td>
</tr>
<tr>
<td>3</td>
<td>0.92</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>0.92 + 1 Hz</td>
<td>1.8</td>
</tr>
<tr>
<td>5</td>
<td>1.19</td>
<td>2.67</td>
</tr>
<tr>
<td>6</td>
<td>1.7</td>
<td>1.67</td>
</tr>
<tr>
<td>7</td>
<td>2</td>
<td>1.46</td>
</tr>
<tr>
<td>8</td>
<td>2.74</td>
<td>1.33</td>
</tr>
<tr>
<td>9</td>
<td>3</td>
<td>1.33</td>
</tr>
<tr>
<td>10</td>
<td>3.74</td>
<td>1</td>
</tr>
<tr>
<td>11</td>
<td>4.07</td>
<td>1.33</td>
</tr>
</tbody>
</table>

Table 1: Bell spectrum based on Risset 1969

The method described here will start with the imitation of a spectrum. For this reason it is called imitative additive synthesis. But "imitative" does not mean that the spectral envelopes of the sound are to be imitated in their progression. Rather, the partials can be modified in different ways which is to be described in more detail.

2 Tasks for Performing Imitative Additive Synthesis in Realtime

In order to use Imitative Additive Synthesis in realtime performance, the Csound application should do these jobs:

- Analyze any sound input - sampled or live - and retrieve the N strongest partials.
- Let the user switch between live input or sampled sound. For the latter, let him choose one sample from a bank of sounds.
- If the input is a sound sample, set the position of the analysis by means of a pointer. Provide several options for moving the pointer:
  - User-controlled by a slider.
  - Continuously moving in defined steps forwards or backwards.
  - Jumping randomly in a given range.
- If the input is a live audio stream, analyze it whenever a new trigger signal (a midi button or a mouse click) has been received.
- Resynthesize the sound with additive synthesis with any number of partials up to N, and with possible offset (not starting at the strongest one).
- Allow for variation between resynthesized notes of the same snapshot by way of random frequency deviations, amplitude changes and by modifying the duration of the partials.
- Facilitate playing the synthesis on a usual midi-keyboard:
  - Define one key which plays the sound at the same pitch it has been recorded.
  - Define a transposition range for the other keys.
  - Let the user control a usual ADSR envelope.
- Allow to print out the analysis results, and to export them in different ways for further use.

The following description shows how these tasks can be realized in Csound. Andrés Cabrera's QuteCsound frontend will be used. It provides an easy-to-program graphical interface which gives the user a visual feedback and lets him control parameters either by midi or by widgets.

3 Retrieving the N strongest partials of a sound and triggering the resynthesis with M partials

The usual way of analyzing the frequency components of a sound is the Fast Fourier Transform (FFT). Thanks to the "phase vocoder streaming" opcodes, FFT in Csound is both simple and fast. There are several opcodes which transform an audio input (realtime, buffer or soundfile) into an "f-signal" which contains the spectral components.

The problem is how to extract the N strongest frequency components total number of bins. This is done by the following operation:

- After performing the FFT, all the amplitude and frequency values of the analyzed sound snapshot are written in the tables giamps and gifreqs. This can be done in Csound by the opcode pvsftw.
- Then, the amplitude table is examined to return the positions of the N strongest values. These positions are written again in a table

![Figure 1: Partial progression of a guitar tone courtesy of Wolfgang Fohl, Hamburg](image)

The number of bins or analysis channels of the FFT depends on the size of the analysis window. If the window size is 1024 (which is a common size), you will get 512 values with pairs of frequency and amplitude (bin 0 is omitted).
This is done by a function which was programmed for this task.\(^3\)

Figure 2: Analysing a sound snapshot

- Whenever a note is to be played, the total sum of the amplitudes required for the resynthesis of Mpartials\(^4\) is calculated. This is necessary for two reasons:
  - The sound input may have very different levels, but the user will want the output volume to depend only on the velocity of the midi-keyboard.
  - If you decided during playing to reduce the number of sinoids M from say 20 to 2, you may nevertheless want to keep the same output level.
- For each note then, the \textit{gimaxindc} table is read for the first M positions - eventually shifted by an offset -, and for each position one instance of an instrument is called. This instrument plays one partial, and it is fed with the relevant input values: the amplitude and frequency of this bin, the summed amplitude, the midi velocity, the midi pitch.

\(^3\) In Csound, defined functions are called User Defined Opcodes. After defining in the orchestra header or an \#include file, they can be used like usual opcodes. For more information, see the page OrchUDO.html in the Canonical Csound Manual (available online at www.csounds.com/manual/).

\(^4\) In the range 1 to N

4 Input and Time Pointer Options

Input can be selected from either a bank of soundfiles, or live input. There is a switch to determine which is to be used.

If soundfiles are used, the most important decision is in what way time pointer should be applied. One option is to manually adjust the position, either by mouse or by a midi-controller. But it can also be nice to "hop" through a sample, starting at a certain position, in a variable hop size. Each time a note is played, the pointer moves a bit
forward or backward. Fast repetitions can cross the whole file, like a flip-book. The third option implemented is a random jumping of the pointer, getting a new position after each note.

The user can decide how many partials he wants to use for the additive resynthesis. This part of the instrument is separated from the analysis, so the snapshot can be probed for the strongest 32 partials, and then all 32, or 10, 5, or just one can be used. An offset parameter allows to start not always at the strongest partial, but at a weaker one. So you can synthesize a rich spectrum or a more sine-like one, and choose whether you want to prefer the most significant or the lesser significant partials. But the partials will always remain ordered by their respective amplitudes.

To avoid always producing the same sound, the user can add random deviations, both to the frequency and to the amplitude multipliers. The maximum frequency deviation is given in cent values, so 100 means that each partial can reach a frequency deviation of up to one semitone. The maximum amplitude deviation is given in deciBel, so 6 means that an amplitude multiplier of 0.5 can vary in the range between 0.25 and 1. With these values, you will get a set of sounds which are different each time, but nevertheless recognizable as one sound.

This random deviation within a certain range is also applied to the individual durations of the partials. Like in natural sounds, each partial has its own "life span". The simplest way of doing this is to assign random deviations to each partial. This is technically possible, because synthesis is carried out by one instance of a sub-instrument for each partial. So it is no problem to give each partial its own duration. The input is given in percent values, 100 meaning that the main duration of 2 seconds can vary between 0.5 and 4 seconds.

A common user-definable adsr envelope is applied, defining the attack time, the decay time, the sustain level, and the release time. It is the same shape for all the partials, but because of the duration differences, the overall shape will differ between the partials.

For playing the sounds via a usual midi keyboard, a key must be defined which plays the sound snapshot at the same pitch as it has been recorded. This reference key can be set by the user arbitrarily. Every other key will transpose the sound. The degree of transposition can also be

---

**5 Playback Options**

There are many options for playing back the sound snapshot. Some basic features have been implemented; some more ideas are discussed at the end of this section.
adjusted by the user, to a semitone, or to any other value. If you set this parameter to 50, the next key on the midi keyboard will shift the sound by 50 cent (a quartertone). If you set it to 200, the next key shifts by a whole tone. If you type 0, the sound will not be transposed at all, so it will be at the same pitch on all keys.

This is the user interface for all these options:

![Playback Parameters](image)

Figure 6: Playback parameters

The playback options very much depend on which kind of music the user wants to play, and how they want to use the instrument. These are some ideas for other possibilities:

- Different tuning systems instead of equal steps from one key to the next.
- Manipulations of the partial relationship to become more harmonic or more inharmonic.
- Make partial durations depend on the pitch so that higher partials decay faster.

6 Export Options

The instrument described here can also be used to perform an FFT analysis and to query for the N strongest bins in this situation. For later use of the analysis results, some export options are implemented.

First, the user can choose to see the results in the gui. This is just a list of values for frequencies and amplitudes, like this:

![Analysis Values](image)

Figure 7: Analysis printout

This list can also be exported to a text file. Either this file contains the same information as the gui printout, or the plain frequency and amplitude data.

If the user wants to use the data in any Csound context, it can be useful to have them transformed in two generalized function tables: one containing the data for the frequency multipliers, one for the amplitude values, like this:
7 Conclusion

This paper is to show how Imitative Additive Synthesis in realtime can be implemented in QuteCsound. The different options presented here likely strain the limits of accessibility; wanting to show what is possible. For really playing it as a live instrument, each user will adapt the code and the gui to their needs, omitting some features and concentrating on others.

8 Acknowledgements

Thanks to Anna for reading the manuscript.

References


The QuteCsound file described here is part of the official QuteCsound distribution (Examples > Synths > Imitative_Additive) since March 2011 in the repositories, or in future releases (after 0.6.1): http://sourceforge.net/projects/qutecsound

It can also be found for download at http://joachimheintz.de/soft/qcsfiles/Imitative_Additive.csd.
Abstract

The article describes an implementation of a synthesis module capable of performing all known types of time based granular synthesis. The term particle synthesis is used to cover granular synthesis and all its variations. An important motivation for this all-inclusive implementation is to facilitate interpolation between the known varieties of particle synthesis. The requirements, design and implementation of the synthesis generator is presented and discussed. Examples of individual varieties are implemented along with a longer interpolated sequence morphing between them. Finally an application, the Hadron Particle Synthesizer, is briefly presented.

Keywords

Granular synthesis, Particle synthesis, CSound

1 Introduction

Granular synthesis is a well established technique for synthesizing sounds based on the additive combination of thousands of very short sonic grains into larger acoustic events [1]. Its potential for musical and sonic expression is abundantly rich through fine-grained (!) control of properties in both the time- and frequency-domain.

The foundation for granular synthesis was laid by the British physicist Dennis Gabor in his studies of acoustical quanta as a means of representation in the theory of hearing [2]. The idea of using grains of sound in music was later expanded into a compositional theory by Iannis Xenakis in his book *Formalized Music* [3].

In its basic form granular synthesis offers low-level control of single grains through parameters such as waveform, frequency, duration and envelope shape, and it typically provides global organization of grains through another set of parameters such as density, frequency band and grain cloud envelope.

There are several variations of the basic scheme. A comprehensive survey of different granular techniques can be found in Curtis Roads' excellent book “Microsound” [4]. We will present a brief summary in the next section. The book suggests the term particle synthesis as a general term covering granular synthesis and all its variations. Although not a formal definition, we will adopt that usage in this paper. Hence our all-in-one implementation of all these techniques is aptly named partikkel, the Norwegian word for 'particle'.

Due to its popularity numerous implementations of granular synthesis have been made available through the years, starting with the pioneering works of Roads (see [4]) and later Truax [5]. Today we find real-time granular synthesis modules included in commercial software such as Absynth and Reaktor from Native Instruments [6] or Max/MSP from Cycling '74 [7]. Granular synthesis is also a household component of open-source, platform-independent audio programming languages such as CSound [8], PureData [9] and SuperCollider [10].

Common to most of these implementations is that they focus on a particular variety of granular synthesis, for instance sound file granulation or asynchronous granular synthesis. The opcode partikkel [11] that we have implemented in the audio processing language CSound, is an attempt to support all known types of time based granular synthesis. To our knowledge it is the only open-source, platform-independent all-in-one solution for granular synthesis.

This paper will motivate the design of our particle generator by extracting requirements from known particle synthesis varieties. After some additional considerations we present the partikkel implementation. Finally we briefly introduce the Hadron Particle Synthesizer that provides a powerful and compact user interface to the particle generator.

---

1An opcode is a basic CSound module that either generates or modifies signals.
2 Particle synthesis

The term particle synthesis covers all the varieties of granular synthesis as described by Roads [4]. In this section we will take a closer look at each variety, starting with basic granular synthesis. We will focus on specific design requirements posed by the variety as input to an all-including implementation.

2.1 Basic granular synthesis

The building block of granular synthesis is the grain, a brief microacoustic event with duration near the threshold of human hearing, typically in the range 1 to 100 milliseconds [4]. Figure 1 shows a typical grain: a sinusoidal waveform shaped by a Gaussian envelope. The parameters necessary to control the grain are:

- Source audio: arbitrary waveform (sampled or periodic)
- Grain shape: envelope function for each grain
- Grain duration
- Grain pitch: playback rate of source audio inside the grain
- Phase (or time pointer): start position for reading the waveform inside each grain

The grain shape does not have to be symmetric. Figure 2 shows a grain envelope with independently specified attack and decay, and a sustain portion in the middle. A flexible implementation should permit updates of both grain envelope and waveform during playback.

The global organization of grains introduces one more parameter:

- Grain rate: the number of grains per second

In synchronous granular synthesis, the grains are distributed at regular intervals as shown in Figure 3. For asynchronous granular synthesis the grain intervals are irregularly distributed, and in this case it might be more correct to use the term grain density than grain rate. An all-including implementation should permit various degrees of soft or hard synchronization.

The concept of a granular cloud is typically associated with asynchronous grain generation within specified frequency limits. The latter can easily be controlled from outside the grain generator by providing a randomly varied, band-limited grain pitch variable. Similarly the amplitude envelope of a cloud of grains may be implemented as external global control of the individual grain amplitudes.

2.2 Glisson synthesis

Glisson synthesis is a straightforward extension of basic granular synthesis in which the grain has an independent frequency trajectory [4]. The grain or glisson creates a short glissando (see Figure 4 above). In order to meet this requirement the granular generator must allow specification of both start and end frequency for each individual particle and also allow control over the pitch sweep curve.
2.3 Grainlet synthesis

Grainlet synthesis is inspired by ideas from wavelet synthesis. We understand a wavelet to be a short segment of a signal, always encapsulating a constant number of cycles. Hence the duration of a wavelet is always inversely proportional to the frequency of the waveform inside it. Duration and frequency are linked (through an inverse relationship). Grainlet synthesis is based on a generalization of the linkage between different synthesis parameters.

Obviously, the greater the number of parameters available for continuous control, the greater the number of possible combinations for parameter linkage. The most common linkage of grainlets is the frequency/duration linkage found in wavelets. More exotic combinations mentioned by Roads [4] are duration/space, frequency/space and amplitude/space. The space parameter refers to the placement of a grain in the stereo field or the spatial position in a 3D multichannel setup.

Grainlet synthesis does not impose additional requirements on the design of the granular generator itself, but suggests the possibility of linking parameters, which can conveniently be accomplished in a control structure external to the actual granular audio generator unit.

2.4 Trainlet synthesis

The specific property that characterizes a trainlet (and also gives rise to its name) is the audio waveform inside each grain. The waveform consists of a band-limited impulse train as shown in Figure 5. The trainlet is specified by:

- Pulse period (or its counterpart, the base frequency)
- Number of harmonics
- Harmonic balance (chroma): The energy distribution between high and low frequency harmonics

In terms of designing a general purpose granular synthesis generator, as we set out to do in this paper, it should be noted that the trainlet waveform has to be synthesized in real time to allow for parametric control over the impulse train. This dictates that the trainlet must be considered a special case when compared to single cycle or sampled waveforms used in the other varieties of particle synthesis.

2.5 Pulsar synthesis

Pulsar synthesis introduces two new concepts to our universal particle synthesis engine: duty cycle and masking. Here the term pulsar is used to describe a sound particle consisting of an arbitrary waveform (the pulsaret) followed by a silent interval. The total duration of the pulsar is labeled the pulsar period, while the duration of the pulsaret is labeled the duty cycle. The pulsaret itself can be seen as a special kind of grainlet, where pitch and duration is linked. A pulsaret can be contained by an arbitrary envelope, and the envelope shape obviously affects the spectrum of the pulsaret due to the amplitude modulation effects inherent in applying the envelope to the signal. Repetitions of the pulsar signal form a pulsar train.

A feature associated with pulsar synthesis is the phenomenon of masking. This refers to the separate processing of individual pulsars, most commonly by applying different amplitude gains to each pulsaret (see Figure 6 for an example). Masking may be done on a periodic or stochastic basis. If the masking pattern is periodic, subharmonics of the pulsar frequency will be generated. To be able to synthesize pulsars in a flexible manner, we should enable grain masking in our general granular synthesizer.

2.6 Formant synthesis

Granular techniques are commonly used to create a spectrum with controllable formants, for example to simulate vocals or speech. Several variants of particle-based formant synthesis (FOF, Vosim, Window Function Synthesis) have been
proposed [4]. As a gross simplification of these techniques one could state that the base pitch is constituted by the grain rate (which is normally periodic), the formant position is determined by the pitch of the source audio inside each grain (commonly a sine wave), and the grain envelope has a significant effect on the formant’s spectral shape. Formant wave-function (FOF) synthesis requires separate control of grain attack and decay durations, and commonly uses an exponential decay shape (see Figure 7). These requirements must be met by the design of our all-including granular generator.

**Figure 7**: Grain shape with complex envelope. The envelope is made up of an overall exponential decay combined with sinusoidal attack and decay segments.

3 Design considerations

3.1 Grain clock

Different varieties of particle synthesis use different methods for organizing the distribution of grains over time, from periodic grain dispersion to asynchronous scattering of grains. A general purpose granular generator must be able to dynamically change the rate and the periodicity of the internal clock used for grain generation. Generation of truly asynchronous grain clouds may require that an external clock source is allowed to trigger grain generation (possibly by disabling the internal clock). In any case, enabling an optional external clock source to control grain dispersion ensures maximum flexibility of grain scheduling. In order to support exotic and yet unknown synchronous granular synthesis varieties it would be useful to add the possibility to gradually synchronize internal and external clocks.

When deliberating the question of the most flexible clock source for our granular generator, we should also consider making the clock adjustable at audio rate\(^2\), so as to enable frequency modulation effects on the clock rate. Obviously, the effect of continuously modulating a clock rate is only manifested at the actual tick output of the clock. Hence the clock rate could be considered as some kind of “clock modulation sampling rate”. Frequency modulation of the grain rate will be the source of further investigation in later research projects.

3.2 Grain masking

The masking concept introduced in relation to pulsar synthesis could be extended to other parameters than amplitude. We could for instance dynamically distribute individual grains to different locations in space. Thus our particle generator could provide a channel mask option and thereby allow individual grains to be routed to specific audio outputs. This feature would also enable the possibility to apply different signal processing effects (for instance different filtering) on individual grains by post-processing the output channels of the generator.

3.3 Waveform

One important reason for designing an all-including particle generator is to enable dynamic interpolation between the different varieties. As we have already pointed out, the generator should support arbitrary waveforms within the grains. As a matter of fact the grain waveform is a distinguishing characteristic of several varieties. In order to morph between them the particle generator must support gradual transitions from one waveform to another.

The most obvious approach to waveform transitions is crossfading. Crossfading between two different waveforms would be sufficient, but it might be interesting to investigate the effects of simultaneously crossmixing even more waveforms into each grain. We also need a crossfading option for trainlet sources, since trainlet synthesis must be treated as a special case. The masking technique discussed in the previous section can easily be extended to include source waveform mixing: a wave-mix mask for truly exotic pulsars.

Providing several simultaneous source waveforms for each grain would naturally also require independent transposition and phase (time pointer) control for each source wave to enable flexible mixing and matching of source waves. As a simple extension to the already flexible playback and mixing of source audio material within each grain, the generator could add support for frequency modulation of the source waveforms.

\(^2\)Audio rate corresponds to the sample rate (as opposed to control rate which normally is orders of magnitude slower)
It is computationally cheap, but its effects in granular synthesis have been sparsely explored. Frequency modulation of source waveform playback pitch could be implemented as phase modulation, using an external audio input to modulate the reading position of the source audio waveform(s).

4 The partikkel Csound opcode

A generalized implementation enabling all known varieties of particle synthesis in one single generator will facilitate new forms of the synthesis technique. To enable the use of such a generalized granular generator in a broader context, it seems apt to implement it in an already existent audio processing language. To broaden the context as much as possible it would be preferable to use an open source language with a large library of signal processing routines already in place. To comply with these requirements, the authors chose to implement the new generator as an opcode for the audio programming language Csound. The opcode was given the name partikkel.

We will now try to sum up the features of partikkel. Where appropriate, we will refer to the specific type of particle synthesis that each feature originates from.

The basic parameters of granular synthesis are grain rate, grain pitch and grain shape/duration, as well as the audio waveform inside each grain. We decided to enable grain rate modifications at audio rate since this might open up new possibilities for frequency modulating the grain rate. The internal grain clock may also be disabled completely for truly asynchronous grain clouds, or it may be run as an internal clock with soft synchronization to an external clock source. For simpler grain displacements (semi-synchronous), a separate grain distribution parameter has been implemented, moving single grains within a time slot of 1/grainrate seconds.

Grain pitch should be relatively straightforward, defined as the playback speed of the audio waveform inside each grain. However, since we use four separate source waveforms we need four separate pitch controls, in addition to one master pitch control. Grain pitch can also be modified at audio rate via a separate frequency modulation audio input parameter to partikkel. Trainlets (or pulse trains) can be used as a fifth source, and we actually need a separate pitch control for them too.

As glission synthesis requires pitch glissandi within each grain, an additional layer of pitch control with start and end pitch for each grain has been added. This type of control over individual grains is implemented in a general manner via a grain masking method. We will return to that topic later.

Different varieties of particle synthesis require different source audio waveforms, and to enable the interpolation between different synthesis types partikkel has the ability to crossfade between different waveforms. Separate phase control over the four source waveforms completes this design requirement. Trainlet synthesis requires a special source waveform of band limited pulse trains. This waveform is synthesized in real time to allow for parametric control over harmonics and chroma.

Both pulsars and formant synthesis require flexible grain envelopes with separate control over shape and time for both the attack and decay portion of the envelope. As a further adjustment to the envelope shape, a sustain time parameter has been added, where the grain amplitude is at maximum for the duration of the sustain segment. To enable even more flexibility, a second enveloping window (full grain length) might be used on top of the primary attack, sustain and decay shape.

Pulsar synthesis introduces a grain masking feature. Normally, this masking would be confined to amplitude and output channel modifications. In partikkel, the masking methods have also been extended to include source waveform mix, pitch glissandi (with separate start and end pitch values), and frequency modulation index masks. The masking feature is implemented by using tables of successive values, partikkel reading one value for each grain before progressing onto the next table index. Start and end/loop indices are also part of this data set, so the mask length and content can be continuously modified while the generator is running. For simplified random particle bursts, a separate parameter (random mask) can be used to randomly mute separate grains.

Trainlet synthesis has not been explicitly accounted for so far. This is because we chose to design the core granular generator to be as generic as possible, and as part of that design decision we determined that any parameter linkage should be left to external implementation. Still, the parameter set and the supported rate of change for each parameter have been designed with parameter linkage in mind.

---

3The choice of four source waveforms is a more or less arbitrary trade-off between complexity and expressivity.
4.1 Implementation notes

The processing in the partikkel opcode consists of two primary phases: grain scheduling and grain rendering. The grain scheduler will place grains according to the time parameters, with each grain being given attributes according to the parameters describing pitch, amplitude, etc.

Grain rendering consists of synthesizing the actual grain waveforms. Despite the large number of parameters utilized by partikkel, the core grain rendering itself is quite simple, and consists of the following stages:

1. interpolated sample reading or DSF\(^4\) synthesis for trainlets
2. frequency modulation
3. frequency sweep (glisson) curve synthesis
4. applying envelope
5. mixing to output buffer(s)

Most of the internal parameters these stages depend upon are calculated on creation of the grain and stored away in a linked list containing one entry per grain, and will not be modified until the end of the grain's life cycle. This is a tradeoff, meaning that partikkel cannot (with the exception of waveform FM) alter any properties influencing a grain during its lifetime, but also means that all the most demanding calculations are performed one time per grain, leaving most processing power to render as many grains as possible at the same time. This might at first seem a limitation, but it can be argued that granular synthesis is at its most promising exactly when grains are allowed to be different, and evolve in succession rather than simultaneously.

5 Examples

A number of implementation examples [12] accompany this paper. The examples are intended to show how different varieties of particle synthesis can be implemented using the generalized technique as described in the paper. First we present a number of individual examples, followed by a long morphing sound, gluing together all the individual examples into a long, continuous transformation.

5.1 Example 1: Basic granular synthesis, sample player with time stretch

In this example, a sound file is used as the source waveform for grains and we use a flexible time pointer (moving phase value) to set the starting point for waveform reading within each grain. This technique is commonly used for time stretching and other time manipulations.

5.2 Example 2: Single cycle source waveform

A sine wave is used as source waveform for each grain. In itself this is a trivial example, but is included to show the transition (in example 10) from reading sampled waveforms to single cycle waveforms. The transition can be considered nontrivial for most oscillators. Not only must the oscillator waveform change on the fly, but the pitch ratio for sampled sounds and single cycle waveforms are usually very different.

5.3 Example 3: Glistons

Glistons in this example have a converging pitch sweep profile. Each single glisson may start on a pitch above or below, gliding quickly and stabilizing on a central pitch.

5.4 Example 4: Trainlets

Trainlets with 20 partials and chroma varying from 1 to 1.5.

5.5 Example 5: Simple pulsars/grainlets

This example shows simple pulsar synthesis. A pulsaret is generated at periodic intervals, followed by a silent interval. The duration of the pulsaret is inversely proportional to the pulsaret pitch, and the pitch gradually changes over the duration of the example. The waveform of the pulsaret is a trainlet.

5.6 Example 6: Pulsar masking

The example starts with a pulsar train similar to the one in example 5. By grain masking we gradually reduce the amplitude of every second grain, then gradually creating a stereo pattern of grains (left-center-right-center). Towards the end of the example, stochastic masking is added.

5.7 Example 7: Formant synthesis

Using granular techniques similar to the classic FOF (Fonction d'onde formantique) generators, where the grain rate constitutes the perceived pitch. The grain pitch (transposition of the waveform inside each grain) controls the placement of a formant region, and the grain shape controls the spectral contour of the formant region. Since our

---

\(^4\) Discrete Summation Formulae (DSF) (see for instance [17])
granular generator allows 4 separate source waveforms with independent pitch, we can create 4 formant regions with one granular generator. The example generates formants as found in the vowel “a” in a male basso voice.

5.8 Example 8: Asynchronous GS

A gradual transformation from synchronous to asynchronous granular synthesis. An asynchronous clock pulse is generated by using a probability function, this clock pulse is used to trigger individual grains.

5.9 Example 9: Waveform mixing

Crossfading between 4 sampled waveforms. From a vocal sample to distorted vibraphone, to cello and finally to a synth pad.

5.10 Example 10: Morphing between all previous examples

One continuous long transformation moving through the granular techniques explored in each previous example.

6 Performing with the partikkel opcode

The all-including implementation of particle synthesis in a single generator encourages further experimentation with granular synthesis techniques. Interpolation between the different granular varieties may reveal new and interesting sonic textures, as would experimentation with some of the more exotic design considerations suggested in section 3.

The flexibility of the partikkel opcode comes with a price. The parameter set is large and unwieldy, particularly in a live performance setting. There seems to be an unavoidable trade-off between high-dimensional control and playability. We have therefore investigated various strategies for meta-parametric control to reduce parameter dimensionality and with that performance complexity, greatly inspired by research on mapping in digital musical instruments [13-15]. The partikkel opcode takes on the role as the fundamental building block in a particle-based digital instrument where the mapping between performance parameters and opcode variables plays a significant part. Not only to increase playability, but also to provide particle synthesis features external to the core generator, such as the parameter linkage of grainlet synthesis. The most recent "front-end" for partikkel is the Hadron Particle Synthesizer. It adds several new features including a number of modulators such as low-frequency oscillators, envelopes and random generators, all interconnected by a dynamic modulation matrix [16]. A simplified control structure was developed to allow real-time performance with precise control over the large parameter set using just a few user interface controls (see Figure 8).

Hadron is freely available, open-source software [12] and will be publicly released in 2011. The screen shot in Figure 8 is taken from the Max For Live version. Other plug-in formats such as VST, RTAS and AU will be supported. Hadron can also be run as a standalone instrument under CSound. Expansion packs with additional parameter states will be made available for purchase at www.partikkelaudio.com.

Acknowledgements

The partikkel opcode was designed by first author Øyvind Brandtsegg. It was implemented as a diploma project by students Thom Johansen and Torgeir Strand Henriksen under the supervision of Sigurd Saue.

References


Audio Plugin development with Cabbage

Rory Walsh
Dundalk Institute of Technology,
Co. Louth,
Ireland
rory.walsh@dkit.ie

Abstract
This paper describes a novel new approach to developing cross-platform audio plugins with Csound. It begins with a short historical overview of projects that led to the development of Cabbage as it exists today, and continues with a more detailed description of Cabbage and its use within digital audio workstations. The paper concludes with an example of an audio effect plugins and a simple MIDI based plugin instrument.

Keywords
Cabbage, WinXound, Csound, Audio Plugin, Audio programming languages.

1 Introduction
In an industry dominated by commercial and closed-source software, audio plugins represent a rare opportunity for developers to extend the functionality of their favourite digital audio workstations, regardless of licensing restrictions. Developers of plugins can concentrate solely on signal processing tasks rather than low-level audio and MIDI communication.

The latest version of Cabbage seeks to provide for the first time a truly cross-platform, multi-format Csound-based plugin solution. Cabbage allows users to generate plugins under three major frameworks: the Linux Native VST[1], Virtual Studio Technology (VST) [2], and Apple's Audio Units [3]. Plugins for the three systems can be created using the same code, interchangeably. Cabbage also provides a useful array of GUI widgets so that developers can create their own unique plugin interfaces.

When combined with the latest version of WinXound[4] computer musicians have a powerful, fully integrated IDE for audio software development using the Csound programming language.

1.1 The Csound host API
The main component of the framework presented here is the Csound 5 library[5], accessed through its API. This is used to start any number of Csound instances through a series of different calling functions. The API provides several mechanisms for two-way communication with an instance of Csound through the use of 'named software' buses.

Cabbage accesses the named software bus on the host side through a set of channel functions, notably Csound::setChannel() and Csound::getChannel(). Csound instruments can read and write data on a named bus using the chnget/chnset opcodes.

In general, the host API allows software to control Csound in a very flexible way, without it the system described in this paper would not have been possible.

2 Background
The ability to run open source audio software in tandem with commercial DAWs is not something new to computer musicians. Systems such as Pluggo[6], PdVST[7] and CsoundVST[8] all provide users with a way to develop audio plugins using open source audio languages. CsoundVST is still available for download but it's anything but a lightweight plugin system. Pluggo and PdVst have been discontinued or are no longer under development.

The software presented in this paper may well have been inspired by the systems mentioned above but is in fact an amalgamation of 3 projects that have been rewritten and redesigned in order to take full advantage of today's emerging plugin frameworks. Before looking at Cabbage in its present state it is worth taking a look at the two main projects it is derived from.

2.1 csLADSPA/csVST
csLADSPA[9] and csVST[10] are two lightweight audio plugin systems that make use of the Csound API. Both toolkits were developed so
that musicians and composers could harness the power of Csound within a host of different DAWs. The concept behind these toolkits is very simple and although each makes use of a different SDK, they were both implemented in the very same way. A basic model of how the plugins work is shown below in fig.1.

![Figure 1. Architecture of a Csound plugin](image)

The host application loads the csLADSPA or csVST plugin. When the user processes audio the plugin routes the selected audio to an instance of Csound. Csound will then process this audio and return it to the plugin which will then route that audio to the host application. The main drawback to these systems is that they do not provide any tools for developing user interfaces. Both csLADSPA and csVST use whatever native interface is provided by the host to display plugin parameters.

### 2.2 Cabbage 2008

Cabbage was first presented to the audio community at the Linux Audio Conference in 2008[11]. The framework provided Csound programmers with no low-level programming experience with a simple, albeit powerful toolkit for the development of standalone cross-platform audio software. The main goal of Cabbage at that time was to provide composers and musicians with a means of easily building and distributing high-end audio applications. Users could design their own graphical interfaces using an easy to read syntax that slots into a unified Csound text file(.csd). This version of Cabbage had no support for plugin development.

### 3 Cabbage 2011

The latest version of Cabbage consolidates the aforementioned projects into one user-friendly cross-platform interface for developing audio plugins. Combining the GUI capabilities of earlier versions of Cabbage with the lightweight csLADSPA and csVST systems, means users can now develop customised high-end audio plugins armed with nothing more than a rudimentary knowledge of Csound and basic programming.

Early versions of Cabbage were written using the wxWidgets C++ GUI library.[12] Whilst wxWidgets provides a more than adequate array of control and other useful classes it quickly became clear that creating plugins with wxWidgets was going to be more trouble than it was worth due to a series of threading issues.

After looking at several other well documented GUI toolkits a decision was made to use the JUCE Class library[13]. Not only does JUCE provide an extensive set of classes for developing GUIs, it also provides a relatively foolproof framework for developing audio plugins for a host of plugin formats. On top of that it provides a robust set of audio and MIDI input/output classes. By using these audio and MIDI IO classes Cabbage bypasses Csound's native IO devices completely. Therefore users no longer need to hack Csound command line flags each time they want to change audio or MIDI devices.

The architecture of Cabbage has also undergone some dramatic changes since 2008. Originally Cabbage produced standalone applications which embedded the instrument's .csd into a binary executable that could then be distributed as a single application. Today Cabbage is structured differently. Instead of creating a new standalone application for each instrument Cabbage is now a dedicated plugin system in itself.

#### 3.1 The Cabbage native host

The Cabbage native host loads and performs Cabbage plugins from disk. The only difference between the Cabbage host and a regular host is that Cabbage can load .csd files directly as plugins. To load Cabbage plugins in other hosts users must first export the Cabbage patch as some form of shared library, dependant on the OS. The Cabbage host provides access to all the audio/MIDI devices available to the user and also allows changes to be made to the sampling rate and buffer sizes. The function of the Cabbage host is twofold. First it provides a standalone player for running GUI based Csound instruments. In this context it functions similarly to the Max/MSP runtime player[6]. Secondly it provides a platform for developing and testing audio plugins. Any instrument that runs in the Cabbage native host can be exported as a plugin.
3.1.1 Cabbage Syntax

The syntax used to create GUI controls is quite straightforward and should be provided within special xml-style tags `<Cabbage>` and `</Cabbage>` which can appear either above or below Csound's own `<CsoundSynthesizer>` tags. Each line of Cabbage specific code relates to one GUI control only. The attributes of each control is set using different identifiers such as `colour()`, `channel()`, `size()` etc. Cabbage code is case sensitive.

3.1 Cabbage widgets

Each and every Cabbage widget has 4 common parameters: position on screen(x, y) and size(width, height). Apart from position and size all other parameters are optional and if left out default values will be assigned. As x/y, width and height are so common there is a special identifier named `bounds(x, y, width, height)` which lets you pass the four values in one go. Below is a list of the different GUI widgets currently available in Cabbage. A quick reference table is available with the Cabbage documentation which illustrates which identifiers are supported by which controls.

```
form caption("title"), pos(x,y), size(width, height), colour("colour")

Form creates the main plugin window. X, Y, Width and Height are all integer values. The default values for size are 400x600. Forms do not communicate with an instance of Csound. Only interactive widgets can communicate with an instance of Csound, therefore no channel identifier is needed. The colour identifier will set the background colour. Any HTML and CSS supported colour can be used.

slider chan("chanName"), pos(x,y), size(width, height), min(float), max(float), value(float), caption("caption"), colour("colour")

There are three types of slider available in Cabbage. A horizontal slider(hslider), a vertical slider(vslider) and a rotary slider(rslider). Sliders can be used to send data to Csound on the channel specified through the "chanName" string. The "chanName" string doubles up as the parameter name when running a Cabbage plugin. For example, if you choose "Frequency" as the channel name it will also appear as the identifier given to the parameter in a plugin host. Each slider that is added to a Cabbage patch corresponds with a plugin parameter on the host side. Min and Max determine the slider range while value initialises the slider to a particular value. If you wish to set Min, Max and Value in one go you can use the `range(min, max, value)` identifier instead. All sliders come with a number box which displays the current value of the slider. By default there is no caption but if users add one Cabbage will automatically place the slider within a captioned groupbox. This is useful for giving labels to sliders.

```
button chan("chanName") pos(x,y), size(width, height), items("OnCaption","OffCaption")

Button creates a on-screen button that sends an alternating value of 0 or 1 when pressed. The “channel” string identifies the channel on which the host will communicate with Csound. “OnCaption” and “OffCaption” determine the strings that will appear on the button as users toggle between two states, i.e., 0 and 1. By default these captions are set to “On” and “Off” but users can specify any strings they wish. If users wish they can provide the same string to both the 'on' and 'off' caption. A trigger button for example won't need to have its captions changed when pressed.

```
checkbox chan("chanName"), pos(x,y), size(width, height), value(val), caption("Caption"), colour("Colour")

Checkboxes function like buttons. The main difference being that the associated caption will not change when the user checks it. As with all controls capable of sending data to an instance of Csound the “chanName” string is the channel on which the control will communicate with Csound. The value attribute defaults to 0.

```
combobox chan("chanName"), caption("caption"), pos(x,y), size(width, height), value(val), items("item1","item2",...)

Combobox creates a drop-down list of items which users can choose from. Once the user selects an item, the index of their selection will be sent to Csound on a channel named by the string “chanName”. The default value is 1 and three items named “item1”, “item2” and “item3” fill the list by default.
Groupbox creates a container for other GUI controls. It does not communicate with Csound but is useful for organising the layout of widgets.

Image draws a shape or picture. The file name passed to file() should be a valid pixmap. If you don't use the file() identifier image will draw a shape. Three type of shapes are supported:

- rounded: a rectangle rounded corners (default)
- sharp: a rectangle with sharp corners
- ellipse: an elliptical shape.

Keyboard creates a virtual MIDI keyboard widget that can be used to test MIDI driven instruments. This is useful for quickly developing and prototyping MIDI based instruments. In order to use the keyboard component to drive Csound instruments you must use the MIDI interop command line flags to pipe the MIDI data to Csound.

3.1.2 MIDI control

In order to control your Cabbage instruments with MIDI CC messages you can use the midictrl(chan, ctrl) identifier. midictrl() accepts two integer values, a controller channel and a controller number. As is the case with the MIDI keyboard widget mentioned above Cabbage handles all it's own MIDI IO. The following code will attach a MIDI hardware slider to a Cabbage slider widget:

slider chan("oscFreq"), bounds(10, 10, 100, 50), range(0, 1000, 0), midictrl(1, 1)

By turning on MIDI debugging in the Cabbage host users can see the channel and controller numbers for the corresponding MIDI hardware sliders. Using midictrl() means that you can have full MIDI control over your Cabbage instruments while running in the standalone host. This feature is not included with Cabbage plugins as the host is expected to take control over the plugin parameters itself.

3.1.3 Native Plugin Parameters

Most plugin hosts implement a native interface for displaying plugin parameters. This usually consists of a number of native sliders that corresponds to the number of plugin parameters as can been seen in the following screen-shot.

![Screen-shot of a Cabbage plugin loaded with Renoise](image)

While slider widgets can be mapped directly to the plugin host GUI, other widgets must be mapped differently. Toggling buttons for example will cause a native slider to jump between maximum and minimum position. In the case of widgets such as comboboses native slider ranges will be split into several segments to reflect the number of choices available to users. If for example a user creates a combobox with 5 elements, the corresponding native slider will jump a fifth each time the user increments the current selection.

![Host automation in Renoise](image)
The upshot of this is that each native slider can be quickly and easily linked with MIDI hardware using the now ubiquitous 'MIDI-learn' function that ships with almost all of today's top DAWs. Because care has being taken to map each Cabbage control with the corresponding native slider, users can quickly set up Cabbage plugins to be controlled with MIDI hardware or through host automation as in fig.4.

4 Cabbage plants

Cabbage plants are GUI abstractions that contain one or more widgets. A simple plant might look like this:

![Image of a basic ADSR abstraction.](image)

An ADSR is a component that you may want to use over and over again. If so you can group all the child components together to form an abstraction. These abstractions, or plants, are used as anchors to the child widgets contained within. All widgets contained within a plant have top and left positions which are relative the the top left position of the parent.

While all widgets can be children of an abstraction, only groupboxes and images can be used as plants. Adding the identifier `plant("plantName")` to an image or groupbox widget definition will cause them to act as plants. Here is the code for a simple LFO example:

```plaintext
image plant("OSC1"), bounds(10, 10, 100, 120), colour("black"), outline("orange"), line(4) {
  rslider channel("Sigfreq1"), bounds(10, 5, 80, 80), caption("OSC 1") colour("white")
  combobox channel("Sigwave1"), bounds(10, 90, 80, 20), items("Sin", "Tri", "Sqr Bi"), colour("black"), textcolour("white")
}
```

![Image of multiple plants together.](image)

The `plant()` identifier takes a string that denotes the name of the plant. This is important because all the widgets that are contained between the pair of curly brackets are now bound to the plant in terms of their position. The big advantage to building abstractions is that you can easily move them around without needing to move all the child components too. Once a plant has been created any widget can link to it by overloading the `pos()` identifier so that it takes a third parameter, the name of the plant as in `pos(0, 0, "LFO")`.

Apart from moving plants around you can also resize them, which in turn automatically resizes its children. To resize a plant we use the `scale(newWidth, newHeight)` identifier. It takes new width and height values that overwrite the previous ones causing the plant and all its children to resize. Plants are designed to be reused across instruments so you don't have to keep rebuilding them from scratch. They can also be used to give your applications a unique look and feel. As they can so easily be moved and resized they can be placed into almost any instrument.

5 Examples

The easiest way to start developing Cabbage instruments and plugins is with WinXound.
WinXound is an open-source editor for Csound and is available on all major platforms. Communication between Cabbage and WinXound is made possible through interprocess communication. Once a named pipe has been established users can use WinXound to take complete control of the Cabbage host meaning they can update and export plugins from the Cabbage host without having to leave the WinXound editor.

When writing Cabbage plugin users need to add -n and -d to the CsOptions section of their .csd file. -n causes Csound to bypass writing of sound to disk. Writing to disk is solely the responsibility of the host application (including the Cabbage native host). If the user wishes to create an instrument plugin in the form of a MIDI synthesiser they should use the MIDI-interop command line flags to pipe MIDI data from the host to the Csound instrument. Note that all Cabbage plugins are stereo. Therefore one must ensure to set nchnls to 2 in the header section of the csd file. Failure to do so will results in extraneous noise being added to the output signal.

The first plugin presented below is a simple effect plugin. It makes use of the PVS family of opcodes. These opcodes provide users with a means of manipulating spectral components of a signal in realtime. In the following example the opcodes pvsanal, pvsblur and pvsynth are used to manipulate the spectrum of an incoming audio stream. The plugin averages the amp/freq time functions of each analysis channel for a specified time. The output is then spatialised using a jitter-spline generator.

```csound
form caption("PVS Blur") size(450, 80)
hslider pos(1, 1), size(430, 50) \ channel("blur"), min(0), max(1), \ caption("Blur time")
</Cabbage>
<CsoundSynthesizer>
<CsOptions>
-d -n -+rtmidi=null -M0 -b1024
</CsOptions>
<CsInstruments>
sr = 44100
ksmps = 32
nchnls = 2
<br>
instr 1
kblurtime chnget "blur"
asig inch 1
fsgl pvsanal asig, 1024, 256, 1024, 1
fts pvsblur fsgl, kblurtime, 2
atps pvsynth ftps
apan jspline 1, 1, 3
outs atps*apan, atps*(1-apan)
endin
</CsInstruments>
</CsoundSynthesizer>
</CsScore>

Figure 6. A simple spectral blurring audio effect

The second plugin is a MIDI-driven plugin instrument. You will see how this instrument uses the MIDI-interop command line parameters in CsOptions to pipe MIDI data from the host into Csound. This plugin also makes use of the virtual MIDI keyboard. The virtual MIDI keyboard is an invaluable tool when it comes to prototyping instruments as it sends MIDI data to the plugin just as a regular host would.

```csound
form caption("Subtractive Synth") size(474, \ 270), colour("black")
groupbox caption("") pos(10, 1), size(430, \ 130)
rslider pos(30, 20), size(90, 90) \ channel("cf"), min(0), max(20000), \ caption("Centre Frequency"), \ colour("white")
rslider pos(130, 20), size(90, 90) \ channel("res"), size(350, 50), min(0), max(1), \ caption("Resonance"), \ colour("white")
rslider pos(230, 20), size(90, 90) \ channel("lfo_rate"), size(350, 50), min(0), max(10), \ caption("LFO Rate"), \ colour("white")
rslider pos(330, 20), size(90, 90) \ channel("lfo_depth"), size(350, 50), min(0), max(10000), \ caption("LFO Depth"), \ colour("white")
keyboard pos(1, 140), size(450, 100)
</Cabbage>
<CsoundSynthesizer>
<CsOptions>
-d -n -+rtmidi=alsa -M0 -b1024 
</CsOptions>
<CsInstruments>
;mInitialize the global variables.
sr = 44100
ksmps = 32
nchnls = 2
masign 0, 1
<br>
instr 1
kcf chnget "cf"
kres chnget "res"
kforate chnget "lfo_rate" 
kfodepth chnget "lfo_depth"
aenv liner 1, 0.1, 1, 0.01
asig vco p5, p4, 1
kfo lfo kfodepth, kforate, 5
aflit moogladder asig, kcf+kfo, kres
outs aflit*aenv, aflit*aenv
endin
</CsInstruments>
</CsoundSynthesizer>
</CsScore>
```

```csound
f1 0 1024 10 1
l1 0 3600
</CsScore>
```
```csound
</CsoundSynthesizer>
```
6 Conclusion

The system has been shown to work quite well in a vast number of hosts across all platforms. It is currently being tested on undergraduate and postgraduate music technology modules in the Dundalk Institute of Technology and the feedback among users has been very positive. The latest alpha version of Cabbage, including a version of WinXound with support for Cabbage can be found at http://code.google.com/p/cabbage/. A full beta version is expected to be released very soon.

7 Acknowledgements

I'd like to express my sincere thanks to everyone on the Csound, Renoise and Juce mailing lists. Without their help and assistance this project would not have been possible. I'd also like to thank the author of WinXound, Stefano Bonetti for his kind help and assistance over the past few months.

References

[1] Linux VST Homepage
   http://www.linux-vst.com/
[3] Apple Audio units Developer Homepage
[4] WinXound Homepage
   http://winxound.codeplex.com/
[6] Cycling 74 Homepage
   http://cycling74.com/products/maxmspjit
[7] PdVst Homepage
   http://crca.ucsd.edu/~jsarlo/pdvst/
[8] CsoundVST
   http://michael-gogins.com
[12] WxWidgets Homepage
   www.wxwidgets.org
[13] Juce website
   http://www.rawmaterialsoftware.com/juce.php
Abstract
This paper presents a new compiler called Poing Impératif. Poing Impératif extends Faust with common features from imperative and object oriented languages.
Imperative and object oriented features make it easier to start using Faust without having to immediately start thinking in fully functional terms. Furthermore, imperative and object oriented features may enable semi-automatic translation of imperative and object oriented code to Faust.
Performance seems equal to pure Faust code if using one of Faust’s own delay operators instead of the array functionality provided by Poing Impératif.

Keywords
Faust, functional programming, imperative programming, object oriented programming, compilation techniques.

1 Introduction
Poing Impératif is a new compiler that extends Faust with imperative and object oriented features.¹
The input code is either Poing Impératif code (described in this paper), or pure Faust code. Pure Faust code and Poing Impératif code can be freely mixed. Pure Faust code goes through Poing Impératif unchanged, while Poing Impératif code is translated to pure Faust code.

1.1 About Faust
Faust [Orlarey et al., 2004] is a programming language developed at the Grame Institute in Lyon. Faust is a fully functional language especially made for audio signal processing. Faust code is compact and elegant, and the compiler produces impressively efficient code. It is simple to compile Faust code into many types of formats such as LADSPA plugins, VST plugins, Q, SuperCollider, CSound, PD, Java, Flash, LLVM, etc. Faust also offer options to automatically take advantage of multiple processors [Orlarey et al., 2009; Letz et al., 2010] and generate code which a C++ compiler is able to vectorize (i.e. generating SIMD assembler instructions) [Scaringella et al., 2003].

1.2 Contributions of Poing Impératif
Purely functional programming is unfamiliar for many programmers, and translating existing DSP code written in object oriented or imperative style into Faust is not straightforward because of different programming paradigms. Poing Impératif can:
1. Make it easier to start using Faust without having to immediately start thinking in fully functional terms.
2. Make it easier to translate imperative and object oriented code to Faust. Porting programs to Faust makes them:
   (a) Easily available on many different platforms and systems.
   (b) Automatically take advantage of multiple processors.
   (c) Possibly run faster. Faust automatically optimizes code in ways which (i) are much hassle to do manually, (ii) are hard to think of, or (iii) may have been overlooked.

1.3 Usage
By default, Poing Impératif starts the Faust compiler automatically on the produced code. Any command line option which is unknown to Poing Impératif is sent further to the Faust compiler. Example:

```
$poing-imperatif -a jack-gtk.cpp -vec freeverb_oo.dsp >freeverb.cpp
$g++ freeverb.cpp -O2 'pkg-config --libs --cflags gtk+-2.0' -ljack -o freeverb
$./freeverb
```

¹Note that since inheritance (subclasses) and polymorphism (method overloading) are not supported, Poing Impératif should probably not be categorized as an OO language.
2 Features

- Setting new values to variables. (i.e. providing imperative operators such as =, ++, +=, etc.)
- Conditionals. (if/else)
- Arrays of floats or ints.
- return operator.
- Classes, objects, methods and constructors.
- Optional explicit typing for numeric variables. Function types are implicitly typed, while object types are explicitly typed. The void type is untyped.
- All features of Faust are supported. Faust code and Poing Impératif code can be mixed.

3 Syntax (EBNF)

```plaintext
class = "class" classname "\{" [var_list] "\}"
  "\{" [class_elem] "\}" .
var_list = var ("," var) .
var = [number_type] varname .
number_type = "int" | "float" .
class_elem = array_decl | object_decl | method | statement .
array_decl = number_type arrayname "\[" expr "\]" ["=" expr] ";" .
object_decl = classname objectname "\{" [expr] "\}" ;.
method = [number_type] methodname "\("" [var_list] "\)" {" (statement) " } .
expr = faust_expression | inc_assign | dec_assign | class | method_call | object_var | array_ref .
  (* Inside classes, faust expressions are extended to expr! *)
object_var = objectname "." varname .
array_ref = arrayname "\[" expr "\]" .
statement = method_call ; | block | single_decl | if | return | assignment .
method_call = objectname "." methodname "\("" [expr] "\)" .
block = "\{" (statement) "\}" .
single_decl = number_type name_list ["=" expr] ";" .
if = "if" "(" expr ")" statement ["else" statement] .
return = "return" expr ";" .
assignment = set_assign | int_assign | dec_assign | cni_assign | obvar_set | array_set .
set_assign = name_list "=" expr ";" .
int_assign = name "=" "+" ; | "-" ; | "*" ; | "/" ; | "%" ; | "+=" | "-=" | "*=" | "/=" | "%=" name ;
cni_assign = name assign_op "=" expr ";" .
assign_op = "+" | "-" | "*" | "/" | "%" |
obvar_set = objectname "." varname "=" expr ";" .
array_set = arrayname "\[" expr "\]" "=" expr ";" .
classname = name .
varname = name .
arrayname = name .
objectname = name .
methname = name .
name_list = name ("," name) .
name = alpha, [alpha | digit | "."] .
```

4 Example of C++ code translated to Poing Impératif

The C++ implementation of the Freeverb\(^2\) all-pass filter looks like this:

```plaintext
class Allpass(
  float feedback;
  int bufsize;
  int bufidx;
  float *buffer;
)
  Allpass(float bufsize, float feedback)(
    this.bufsize = bufsize;
    this.feedback = feedback;
    buffer = calloc(sizeof(float), bufsize);
  }
  float Allpass::process(float input){
    float bufout = buffer[bufidx];
    float output = -input + bufout;
    buffer[bufidx] = input + (bufout * feedback);
    if(bufidx == bufsize)
      bufidx = 0;
    return output;
  }
};
```

A semi-automatic translation to Poing Impératif yields:

```plaintext
class Allpass(int bufsize, float feedback){
  float buffer[bufsize];
  int bufidx;
  process(float input){
    float bufout = buffer[bufidx];
    float output = -input + bufout;
    buffer[bufidx] = input + (bufout * feedback);
    if(bufidx == bufsize)
      bufidx = 0;
    return output;
  }
};
```

5 Constructor

In the Allpass example above, the Poing Impératif class had a slightly different form than the C++ version since a constructor was not needed.

For classes requiring a constructor, imperative code can be placed directly in the class block. A class describing a bank account giving 50 extra euros to all to new accounts, can be written like this:

```plaintext
class Account(int euros){
  euros += 50; // Constructor!
  debit(int amount){
    euros -= amount;
  }
  deposit(int amount){
    euros += amount;
  }
};
```

\(^2\)Freeverb is a popular reverb algorithm made by “Jezar at Dreampoint”. See Julius O. Smith’s Freeverb page for more information about it: https://ccrma.stanford.edu/~jos/pasp/Freeverb.html (The web page is from his book “Spectral Audio Signal Processing.”)
6 Accessing a Poing Impératif class from Faust

The process method is used to bridge Poing Impératif and Faust. If a class has a method called process, and that process method contains at least one return statement, Poing Impératif creates a Faust function with the same name as the class. We call this function the class entry function.

The arguments for the class entry function is created from the class arguments and the process method arguments.

⇒ For instance, the entry function for this class:

```csharp
class Vol(float volume){
    process(float input){
        return input*volume;
    }
}
```

...looks like this:

```csharp
Vol(volume,input) = input*volume;
```

...and it can be used like this:

```csharp
half_volume = Vol(0.5);
process(input) = half_volume(input);
```

6.1 Recursive variables

In case the class has variables which could change state between calls to the class entry function, we use the recursive operator (\(\_\)) to store the state of those variables.

⇒ For instance, this class:

```csharp
class Accumulator{
    int sum;
    process(int inc){
        sum += inc;
        return sum;
    }
}
```

...is transformed into the following Faust code:

```csharp
Accumulator(inc) = (func0 ~ (,_,_)) : retfunc0 with{
    func0(sum,not_used) = (sum+inc, inc); // sum += inc;
    retfunc0(sum,inc) = sum; // return sum;
};
```

6.2 Constructors in the class entry function

In case a class contains constructor code or numeric values initialized to a different value than 0 or 0.0, an additional state variable is used to keep track of whether the function is called for the first time. In case this additional state variable has the value 0 (i.e. it is the first time the class entry function is called), a special constructor function is called first to initialize those state variables.

7 Conversion from Poing Impératif to Faust

7.1 Setting values of variables

Faust is a purely functional languages. It is not possible to give a variable a new value after the initial assignment, as illustrated by the following pseudocode:

Possible:

```csharp
{ int a = 5;
  return a
}
```

Impossible:

```csharp
{ int a = 5;
  a = 6;
  return a;
}
```

One way to circumvent this is to use a new variable each time we set new values. For instance, adding 1 to \(a\) would look like this:

\[ a_2 = a_1 + 1. \]

However, Poing Impératif uses a different approach, which is to make all operations, including assignments, into function calls.

⇒ For example, the following code:

```csharp
float a = 1.0, b=0.0;
a = a + 1.0;
b = a + 2.3;
```

...is transformed into:

```csharp
func0(a,b) = func1(1.0 , 0.0); // a = 1.0, b=0.0
func2(a,b) = func3(a+1.0, b); // a = a+1.0
func4(a,b) = func5(a , a+2.3); // b = a+2.3
```

7.2 Conditionals

When every operation is a function call, branching is simple to implement.

⇒ For instance, the following code:

```csharp
if(a==0)
a=1;
else
a=2;
```

...is transformed into:

```csharp
func1(a) = if(a==0,func2(a),func3(a)); // if(a==0)
func2(a) = func4(1); // a=1
func3(a) = func4(1); // a=2
```

if is here a Faust macro [Gräf, 2010], and it is made to supports multiple signals. The if macro looks like this:

```csharp
if(a,(k1,k2),(k3,k4)) = if(a,k1,k3),if(a,k2,k4);
if(a,k1,k2) = select2(a,k2,k1);
```
7.3 Methods
In Poing Impératif, an object is a list of all the variables used in a class (including method arguments). Behind the scene, every method receives the “this” object (and nothing more). Every method also returns the “this” object (and nothing more). Naturally, the “this” object may be modified during the execution of a method.

⇒ For instance, the method add in:

```c
class Bank{
    int a;
    add(int how_much){
        a += how_much;
    }
}
```

...is transformed into:

```c
Bank__add(a, how_much) = func0(a, how_much) with{
    func0(a, how_much) = (a + how_much, how_much); // a += how_much
}
```

If a method takes arguments, the corresponding variable in the “this” object is set automatically by the caller before the method function is called.

7.4 Return
A special return function is created for each method which calls return. The reason for using the return function to return values, instead of for instance using a special variable to hold a return value, is because it is possible to return more than one value (i.e. to return parallel signals). Furthermore, it is probably cleaner to use a special return function than to figure out how many signals the various methods might return and make corresponding logic to handle situations that might show up because of this.

The return function uses an ‘n’ argument (holding an integer) to denote which of the return expressions to return.

⇒ For instance, the body of process in the following class:

```c
class A{
    process(int selector){
        if(selector)
            return 2;
        else
            return 3;
    }
}
```

...look like this:

```c
A__process(selector, n) = func0(selector, n) with{
    func0(selector, n) = if(selector, func1(selector, n), func2(selector, n));
    func1(selector, n) = (selector, 0); // First return
    func2(selector, n) = (selector, 1); // Second return
};
A__process__return(selector, n) =
    if(n==0,
        2, // Return 2 from the first return
    3); // Return 3 from the second return
```

7.5 Arrays
Faust has a primitive called rwtable which reads from and writes to an array. The syntax for rwtable looks like this:

```c
rwtable(size, init, write_index, write_value, read_index);
```

Using rwtable to implement imperative arrays is not straightforward. The problem is that rwtable does not return a special array object. Instead, it returns the numeric value stored in the cell pointed to by ‘read_index’. This means that there is no array object we can send around.

Our solution is to use rwtable only when reading from an array. When we write to an array, we store the new value and array position in two new variables.

⇒ For instance, the body of process in the following class:

```c
class Array{
    float buf[1000]=1.0;
    process(int i){
        float a = buf[i];
        buf[i] = a + 1.0;
    }
}
```

...is transformed into:

```c
/* float a = buf[i] */
func0(a, i, buf_pos, buf_val) =
    func1(rwtable(1000, 1.0, buf_pos, buf_val, i), i, buf_pos, buf_val);
/* buf[i] = a + 1.0 */
func1(a, i, buf_pos, buf_val) =
    (a, i, buf_pos, buf_val);
```

However, this solution has a limitation: If a buffer is written two times in a row, only the second writing will have effect.

It might be possible to use Faust’s foreign function mechanism to achieve complete array functionality, by implementing arrays directly in C. However, this could limit Faust’s and the C compilers ability to optimize. It would also complicate the compilation process, and limit Poing Impératif to only work with C and C++ (i.e. it would not work with Java, LLVM or other languages (or other bitcode/binary formats) Faust supports unless we implement array interfaces to Faust for those as well.)

It is also quite complicated to figure out how many output signals an expression has. See [Orlarey et al., 2004].
A fairly relevant question is how important full array functionality is? Since full array functionality is not needed for any programs written for pure Faust, it’s tempting to believe this functionality can be skipped.\footnote{One situation where it quite undoubtedly would be useful to write or read more than once per sample iteration, is for doing resampling. But for resampling, the Faust developers are currently working on a implementing a native solution. [Jouvelot and Orlarey, 2009]}

8 Performance compared to Faust

In Poing Impératif, Freeverb can be implemented like this:\footnote{The values for the constants `combtuningL1`, `combtuningL2`, `allpasstuningL1`, etc. are defined in the file “examples/freeverb.dsp” in the Faust distribution.}

```plaintext
class Allpass(int bufsize, float feedback){
    int bufidx;
    float buffer[bufsize];
    process(input){
        float bufout = buffer[bufidx];
        float output = -input + bufout;
        buffer[bufidx] = input + (bufout*feedback);
        if(++bufidx>=bufsize)
            bufidx = 0;
        return output;
    }
}

class Comb(int bufsize, float feedback, float damp){
    float filterstore;
    int bufidx;
    float buffer[bufsize];
    process(input){
        filterstore = (buffer[bufidx]*(1.0-damp)) + (filterstore*damp);
        float output = input + (filterstore*feedback);
        buffer[bufidx] = output;
        if(++bufidx>=bufsize)
            bufidx = 0;
        return output;
    }
}

class MonoReverb(float fb1, float fb2, float damp, float spread){
    Allpass allpass1(allpasstuningL1+spread, fb2);
    Allpass allpass2(allpasstuningL2+spread, fb2);
    Allpass allpass3(allpasstuningL3+spread, fb2);
    Allpass allpass4(allpasstuningL4+spread, fb2);
    Comb comb1(combtuningL1+spread, fb1, damp);
    Comb comb2(combtuningL2+spread, fb1, damp);
    Comb comb3(combtuningL3+spread, fb1, damp);
    Comb comb4(combtuningL4+spread, fb1, damp);
    Comb comb5(combtuningL5+spread, fb1, damp);
    Comb comb6(combtuningL6+spread, fb1, damp);
    Comb comb7(combtuningL7+spread, fb1, damp);
    Comb comb8(combtuningL8+spread, fb1, damp);
    process(input){
        return allpass1.process(
            allpass2.process( 
                allpass3.process( 
                    allpass4.process( 
                        comb1.process(input) +
                        comb2.process(input) +
                        comb3.process(input) +
                        comb4.process(input) +
                        comb5.process(input) +
                        comb6.process(input) +
                        comb7.process(input) +
                        comb8.process(input) ) ) 
            ) 
        )
    }
}

class StereoReverb(float fb1, float fb2, float damp, int spread){
    MonoReverb rev0(fb1,fb2,damp,0);
    MonoReverb rev1(fb1,fb2,damp,spread);
    process(float left, float right){
        return rev0.process(left,right),
               rev1.process(left,right);
    }
}

class FxCtrl(float gain, float wet, Fx){
    process(float left, float right){
        return left *(1-wet) + x.left *wet,
               right*(1-wet) + x.right*wet;
    }
}

process = FxCtrl(fixedgain,
                 wetSlider,
                 StereoReverb(combfeed,
                               allpassfeed,
                               dampSlider,
                               stereospread
                 )
                )
```

The version of freeverb included with the Faust distribution (performing the exact same computations) looks like this:\footnote{Slightly modified for clarity.}

```plaintext
allpass(bufsize, feedback) = 
      (_, <: (*(feedback),_:+:@(bufsize)), -) ~ _ : (!,_);
comb(bufsize, feedback, damp) = 
      (+:@(bufsize)) ~ (*(1-damp) : (+ ~ *(damp)) : *(feedback));
monoReverb(fb1, fb2, damp, spread) = _ <: comb(combtuningL1+spread, fb1, damp),
                                         comb(combtuningL2+spread, fb1, damp),
                                         comb(combtuningL3+spread, fb1, damp),
                                         comb(combtuningL4+spread, fb1, damp),
                                         comb(combtuningL5+spread, fb1, damp),
                                         comb(combtuningL6+spread, fb1, damp),
                                         comb(combtuningL7+spread, fb1, damp),
                                         comb(combtuningL8+spread, fb1, damp)
                                         >>
                                         allpass (allpasstuningL1+spread, fb2)
                         : allpass (allpasstuningL2+spread, fb2)
                         : allpass (allpasstuningL3+spread, fb2)
                         : allpass (allpasstuningL4+spread, fb2)
                         :
stereoReverb(fb1, fb2, damp, spread) = _ <: monoReverb(fb1, fb2, damp, 0),
                                         monoReverb(fb1, fb2, damp, spread);
fxctrl(gain,wet,Fx) = _ <: (*(gain),*(gain) : Fx : *(wet),*(wet)),
                       *(1-wet),*(1-wet)
                       +> _,_;
process = fxctrl(fixedgain,
                 wetSlider,
                 stereoReverb(combfeed,
                               allpassfeed,
                               dampSlider,
                               stereospread
                 )
                )
```

Benchmarking these two versions against each other showed that the version written for pure Faust was approximately 30% faster than the version written for Poing Impératif.
After inspecting the generated C++ source for the Allpass class and the Comb class, it seemed like the only reason for the difference had to be the use of `rwtable` to access arrays.

By changing the Poing Impératif versions of Comb and Allpass to use Faust’s delay operator `@` instead of `rwtable`, we get this code:

```cpp
class Allpass(int bufsize, float feedback){
    float bufout;
    process(float input){
        float output = -input + bufout;
        bufout = input + (bufout*feedback) @ bufsize;
        return output;
    }
}

class Comb(int bufsize, float feedback, float damp){
    float filterstore;
    float bufout;
    process(float input){
        filterstore = (output*(1.0-damp)) + (filterstore*damp);
        bufout = input + (filterstore*feedback) @ bufsize;
        return bufout;
    }
}
```

Now the pure Faust version was only 7.5% faster than the Poing Impératif version. This result is quite good, but considering that semantically equivalent C++ code were generated both for the Comb class and the Allpass class (the Allpass class was even syntactically equivalent), plus that optimal Faust code were generated for the three remaining classes (MonoReverb, StereoReverb, and FxCtrl), both versions should in theory be equally efficient. However, after further inspection of the generated C++ code, a bug in the optimization part of the Faust compiler was revealed. After manually fixing the two non-optimal lines of C++ code caused by this bug in the Faust compiler, both versions of Freeverb produce similarly efficient code. The final two C++ sources also look semantically equivalent.

### 9 Implementation

The main part of Poing Impératif is written in the Qi language [Tarver, 2008]. Minor parts of the source are written in C++ and Common Lisp. Poing Impératif uses Faust’s own lexer.

The source is released under GPL and can be downloaded from: [http://www.notam02.no/arkiv/src/](http://www.notam02.no/arkiv/src/)

### 10 Acknowledgments

Funded by Integra, a 3-year project part-funded by the Culture programme of the European Commission.

### References


---

8Semantically equivalent means here that the code is equal, except that variable names might differ, independent statements could be placed in a different order, or that the number of unnecessary temporary variables differ.

9The decreased performance was caused by two different summing orders of the same group of signals (which is a bug, order is supposed to be equal). This again caused sub-summations not to be shared, probably because equal order is needed to identify common subexpressions. The bug only causes a slight decreased performance in certain situations, it does not change the result of the computations. The bug can also be provoked by recoding the definition of allpass in the pure Faust version of Freeverb to:

```cpp
allpass(bufsize, feedback, input) = (process '(_,!)) : (!,_) with{
    process(bufout) = {
        (input + (bufout*feedback) @ bufsize ),
        (-input + bufout )
    };
};
```
...which is just another way to write the same function.

The bug was reported right before this paper was submitted, it has been acknowledged, and the problem is being looked into. Thanks to Yann Orlarey for a fast response and for confirming what could be the problem.
An LLVM-based Signal-Processing-Compiler embedded in Haskell

Henning Thielemann
Institut für Informatik, Martin-Luther-Universität Halle-Wittenberg,
Von-Seckendorff-Platz 1,
06110 Halle,
Germany,
henning.thielemann@informatik.uni-halle.de

Abstract
We discuss a programming language for real-time audio signal processing that is embedded in the functional language Haskell and uses the Low-Level Virtual Machine as back-end. With that framework we can code with the comfort and type safety of Haskell while achieving maximum efficiency of fast inner loops and full vectorisation. This way Haskell becomes a valuable alternative to special purpose signal processing languages.

Keywords
Functional programming, Haskell, Low-Level Virtual machine, Embedded Domain Specific Language

1 Introduction
Given a data flow diagram as in Figure 1 we want to generate an executable machine program. First we must (manually) translate the diagram to something that is more accessible by a machine. Since we can translate data flows almost literally to function expressions, we choose a functional programming language as the target language, here Haskell [Peyton Jones, 1998]. The result can be seen in Figure 2. The second step is to translate the function expression to a machine oriented presentation. This is the main concern of our paper.

Since we represent signals as sequences of numbers, signal processing algorithms are usually loops that process these numbers one after another. Thus our goal is to generate efficient loop bodies from a functional signal processing representation. We have chosen the Low-Level Virtual-Machine (LLVM) [Lattner and Adve, 2004] for the loop description, because LLVM provides a universal representation for machine languages of a wide range of processors. The LLVM library is responsible for the third step, namely the translation of portable virtual machine code to actual machine code of the host processor.

Our contributions are

- a representation of an LLVM loop body that can be treated like a signal, described in Section 3.1,
- a way to describe causal signal processes, which is the dominant kind of signal transformations in real-time audio processing and which allows us to cope efficiently with multiple uses of outputs and with feedback of even small delays, guaranteed deadlock-free, developed in Section 3.2,
- and the handling of internal filter parameters in a way that is much more flexible than traditional control rate/sample rate schemes, presented in Section 3.3.

Due to space constraints we omitted some parts, like the use of vector arithmetic and according benchmarks, that you can find in [Thielemann, 2010b].

2 Background
We want to generate LLVM code from a signal processing algorithm written in a declarative way. We like to write code close to a data flow diagram and the functional paradigm seems to be appropriate.
We could design a new language specifically for this purpose, but we risk the introduction of design flaws. We could use an existing signal processing language, but usually they do not scale well to applications other than signal processing. Alternatively we can resort to an existing general purpose functional programming language or a subset of it, and write a compiler with optimisations adapted to signal processing needs. But writing a compiler for any modern “real-world” programming language is a task of several years, if not decades. A compiler for a subset of an existing language however would make it hard to interact with existing libraries. So we can still tune an existing compiler for an existing language, but given the complexity of modern languages and their respective compilers this is still a big effort. It might turn out that a change that is useful for signal processing kills performance for another application.

A much quicker way to adapt a language to a special purpose is the Embedded Domain Specific Language (EDSL) approach [Landin, 1966]. In this terminology “embedded” means, that the domain specific (or “special purpose”) language is actually not an entirely new language, but a way to express domain specific issues using corresponding constructs and checks of the host language. For example, writing an SQL command as string literal in Java and sending it to a database, is not an EDSL. In contrast to that, Hibernate [Elliott, 2004] is an EDSL, because it makes database table rows look like ordinary Java objects and it makes the use of foreign keys safe and comfortable by making foreign references look like Java references.

In the same way we want to cope with signal processing in Haskell. In the expression

\[
\text{amplify} \quad \text{(exponential halfLife amp)} \quad \text{(osci Wave.saw phase freq)}
\]

the call to osci shall not produce a signal, but instead it shall generate LLVM code that becomes part of a signal generation loop later. In the same way amplify assembles the code parts produced by exponential and osci and defines the product of their results as its own result. In the end every such signal expression is actually a high-level LLVM macro and finally, we pass it to a driver function that compiles and runs the code. Where Hibernate converts Java expressions to SQL queries, sends them to a database and then converts the database answers back to Java objects, we convert Haskell expressions to LLVM bitcode, send it to the LLVM Just-In-Time (JIT) compiler and then execute the resulting code. We can freely exchange signal data between pure Haskell code and LLVM generated code.

The EDSL approach is very popular among Haskell programmers. For instance interfaces to the Csound signal processing language [Hu-dak et al., 1996] and the real-time software synthesiser SuperCollider [Drape, 2009] are written this way. This popularity can certainly be attributed to the concise style of writing Haskell expressions and to the ease of overloading number literals and arithmetic operators. We shall note that the EDSL method has its own shortcomings, most notably the sharing problem that we tackle in Section 3.2.

In [Thielemann, 2004] we have argued extensively, why we think that Haskell is a good choice for signal processing. Summarised, the key features for us are polymorphic but strong static typing and lazy evaluation. Strong typing means that we have a wide range of types that the compiler can distinguish between. This way we can represent a trigger or gate signal by a sequence of boolean values (type Bool) and this cannot be accidentally mixed up with a PCM signal (sample type Int8), although both types may be represented by bytes internally. We can also represent internal parameters of signal processes by opaque types that can be stored by the user but cannot be manipulated (cf. Section 3.3). Polymorphic typing means that we can write a generic algorithm that can be applied to single precision or double precision floating point numbers, to fixed point numbers or complex numbers, to serial or vectorised signals. Static typing means that the Haskell compiler can check that everything fits together when compiling a program or parts of it. Lazy evaluation means, that we can transform audio data, as it becomes available, while programming in a style, that treats those streams, as if they would be available at once.

The target of our embedded compiler is LLVM. It differs from Csound and SuperCollider in that LLVM is not a signal processing system. It is a high-level assembler and we have to write the core signal processing building blocks ourselves. However, once this is done, assembling those blocks is as simple as writing Csound orchestra files or SuperCollider SCLang programs. We could have chosen a concrete
machine language as target, but LLVM does a much better job for us: It generates machine code for many different processors, thus it can be considered a portable assembler. It also supports the vector units of modern processors and target dependent instructions (intrinsics) and provides us with a large set of low-level to high-level optimisations, that we can even select and arrange individually. We can run LLVM code immediately from our Haskell programs (JIT), but we can also write LLVM bytecode files for debugging or external usage.

3 Implementation
We are now going to discuss the design of our implementation [Thielemann, 2010a].

3.1 Signal generator
In our design a signal is a sequence of sample values and a signal generator is a state transition system, that ships a single sample per request while updating the state. E.g. the state of an exponential curve is the current amplitude and on demand it returns the current amplitude as result while decreasing the amplitude state by a constant factor. In the same way an oscillator uses the phase as internal state. Per request it applies a wave function on the phase and delivers the resulting value as current sample. Additionally it increases the phase by the oscillator frequency and wraps around the result to the interval [0, 1). This design is much inspired by [Coutts et al., 2007].

According to this model we define an LLVM signal generator in Haskell essentially as a pair of an initial state and a function, that returns a tuple containing a flag showing whether there are more samples to come, the generated sample and the updated state.

\[
\text{type Generator a = forall state.} \\
\text{\hspace{2em} (state,} \\
\text{\hspace{4em} state \rightarrow \text{Code (V Bool, (a, state)))}
\]

Please note, that the actual type definition in the library is a bit different and much larger for technical reasons.

The lower-case identifiers are type variables that can be instantiated with actual types. The variable a is for the sample type and state for the internal state of the signal generator. Since Generator is not really a signal but a description for LLVM code, the sample type cannot be just a Haskell number type like Float or Double. Instead it must be the type for one of LLVM’s virtual registers, namely V Float or V Double, respectively. The types V and Code are imported from a Haskell interface to LLVM [O’Sullivan and Augustsson, 2010]. Their real names are Value and CodeGenFunction, respectively.

The type parameter is not restricted in any way, thus we can implement a generator of type Generator (V Float, V Float) for a stereo signal generator or Generator (V Bool, V Float) for a gate signal and a continuous signal that are generated synchronously. We do not worry about a layout in memory of an according signal at this point, since it may be just an interim signal that is never written to memory. E.g. the latter of the two types just says, that the generated samples for every call to the generator can be found in two virtual registers, where one register holds a boolean and the other one a floating point number.

We like to complement this general description with the simple example of an exponential curve generator.

\[
\text{exponential ::} \\
\text{Float \rightarrow \text{Float \rightarrow Generator (V Float)}} \\
\text{exponential halfLife amp \text{=} \text{(valueOf amp,}} \\
\text{\hspace{2em} \text{\textbackslash y0 \rightarrow do}} \\
\text{\hspace{4em} y1 \leftarrow \text{mul y0 (valueOf}} \\
\text{\hspace{6em} (2**(-1/halfLife)))}} \\
\text{\hspace{4em} \text{return (valueOf True, (y0, y1)))}
\]

For simplification we use the fixed type Float but in the real implementation the type is flexible. The implementation is the same, only the real type of exponential is considerably more complicated because of many constraints to the type parameters.

The function valueOf makes a Haskell value available as constant in LLVM code. Thus the power computation with ** in the mul instruction is done by Haskell and then implanted into the LLVM code. This also implies that the power is computed only once. The whole transition function, that is the second element of the pair, is a lambda expression, also known as anonymous function. It starts with a back-slash and its argument y0, which identifies the virtual register, that holds the current internal state. It returns always True because the curve never terminates and it returns the current amplitude y0 as current sample and the updated amplitude computed by a multiplication to be found in the register identified by y1.
We have seen, how basic signal generators work, however, signal processing consists largely of transforming signals. In our framework a signal transformation is actually a generator transformation. That is, we take apart given generators and build something new from them. For example the controlled amplifier dissects the envelope generator and the input generator and assembles a generator for the amplified signal.

```haskell
amplify ::
  Generator (V Float) ->
  Generator (V Float) ->
  Generator (V Float)
amplify (envInit, envTrans)
  (inInit, inTrans) =
  ((envInit, inInit),
   (\(e0,i0) -> do
    (eCont,(ev,e1)) <- envTrans e0
    (iCont,(iv,i1)) <- inTrans i0
    y <- mul ev iv
    cont <- and eCont iCont
    return (cont, (y, (e1,i1)))))
```

So far our signals only exist as LLVM code, but computing actual data is straightforward:

```haskell
render ::
  Generator (V Float) ->
  V Word32 -> V (Ptr Float) ->
  Code (V Word32)
render (start, next) size ptr = do
  (pos,_) <- arrayLoop size ptr start $
  \ ptri s0 -> do
    (cont,(y,s1)) <- next s0
    ifThen cont () (store y ptri)
    return (cont, s1)
  ret pos
```

The ugly branching that is typical for assembly languages including that of LLVM is hidden in our custom functions `arrayLoop` and `ifThen`. **Haskell** makes a nice job as macro assembler. Again, we only present the most simple case here. The alternative to filling a single buffer with signal data is to fill a sequence of chunks, that are created on demand. This is called **lazy evaluation** and one of the key features of **Haskell**.

At this point, we might wonder, whether the presented model of signal generators is general enough to match all kinds of signals, that can appear in real applications. The answer is “yes”, since given a signal there is a generator that emits that signal. We simply write the signal to a buffer and then use a signal generator, that manages a pointer into this buffer as internal state. This generator has a real-world use when reading a signal from a file. We see that our model of signal generators does not impose a restriction on the kind of signals, but it well restricts the access to the generated data: We can only traverse from the beginning to the end of the signal without skipping any value. This is however intended, since we want to play the signals in real-time.

### 3.2 Causal Processes

While the above approach of treating signal transformations as signal generator transformations is very general, it can be inefficient. For example, for a signal generator \( x \) the expression \( \text{mix } x \times x \) does not mean that the signal represented by \( x \) is computed once and then mixed with itself. Instead, the mixer runs the signal generator \( x \) twice and adds the results of both instances. I like to call that the **sharing problem**. It is inherent to all DSLs that are embedded into a purely functional language, since in those languages objects have no identity, i.e. you cannot obtain an object’s address in memory. The sharing problem also occurs, if we process the components of a multi-output signal process individually, for instance the channels of a stereo signal or the lowpass, bandpass, highpass components of a state variable filter. E.g. for delaying the right channel of a stereo signal we have to write `stereo (left x) (delay (right x))` and we run into the sharing problem, again.

We see two ways out: The first one is relying on LLVM’s optimiser to remove the duplicate code. However this may fail since LLVM cannot remove duplicate code if it relies on seemingly independent states, on interaction with memory or even on interaction with the outside world. Another drawback is that the temporarily generated code may grow exponentially compared to the code written by the user. E.g. in

```haskell
let y = mix x x
  z = mix y y
in mix z z
```

the generator \( x \) is run eight times.

The second way out is to store the results of a generator and share the storage amongst all users of the generator. We can do this by rendering the signal to a lazy list, or preferably to a lazily generated list of chunks for higher performance. This approach is a solution to the
general case and it would also work if there are signal processes involved that shrink the time line, like in `mix x (timeShrink x)

While this works in the general case, there are many cases where it is not satisfying. Especially in the example `mix x x` we do not really need to store the result of `x` anywhere, since it is consumed immediately by the mixer. Storing the result is at least inefficient in case of a plain Haskell singly linked list and even introduces higher latency in case of a chunk list.

So what is the key difference between `mix x x` and `mix x (timeShrink x)`? It is certainly, that in the first case data is processed in a synchronous way. Thus it can be consumed (mixed) as it is produced (generated by `x`). However, the approach of signal transformation by signal generator transformation cannot model this behaviour. When considering the expression `mix x (f x)` we have no idea whether `f` maintains the “speed” of its argument generator. That is, we need a way to express that `f` emits data synchronously to its input. For instance we could define

```haskell
type Map a b = a -> Code b
```

that represents a signal transformation of type `Generator a -> Generator b`. It could be applied to a signal generator by a function `apply` with type

```haskell
Map a b -> Generator a -> Generator b
```

and where we would have written `f x` before, we would write `apply f x` instead.

It turns out that `Map` is too restrictive. Our signal process would stay synchronous if we allow a running state as in a recursive filter and if we allow termination of the signal process before the end of the input signal as in the Haskell list function `take`. Thus, what we actually use, is a definition that boils down to

```haskell
type Causal a b = forall state.
  (state, (a, state)) ->
  Code (V Bool, (b, state))
```

With this type we can model all kinds of causal processes, that is, processes where every output sample depends exclusively on the current and past input samples. The `take` function may serve as an example for a causal process with termination.

```haskell
take :: Int -> Causal a a
take n = (valueOf n, \(a,\_\_\) -> do
  cont <- icmp IntULT (valueOf 0) toDo
  stillToDo <- sub toDo (valueOf 1)
  return (cont, (a, stillToDo)))
```

The function `apply` for applying a causal process to a signal generator has the signature

```haskell
apply :: Causal a b ->
  Generator a -> Generator b
```

and its implementation is straightforward. The function is necessary to do something useful with causal processes, but it loses the causality property. For sharing we want to make use of facts like that the serial composition of causal processes is causal, too, but if we have to express the serial composition of processes `f` and `g` by

```haskell
apply f (apply g x),
```

then we cannot make use of such laws. The solution is to combine processes with processes rather than transformations with signals. E.g. with `>>>` denoting the serial composition we can state that `g >>> f` is a causal process.

In the base Haskell libraries there is already the `Arrow` abstraction, that was developed for the design of integrated circuits in the Lava project, but it proved to be useful for many other applications. The `Arrow` type class provides a generalisation of plain Haskell functions. For making `Causal` an instance of `Arrow` we must provide the following minimal set of methods and warrant the validity of the arrow laws [Hughes, 2000].

```haskell
arr :: (a -> b) -> Causal a b
(<<<) :: Causal a b -> Causal b c
first :: Causal a b -> Causal(a,c)(b,c)
```

The infix operator `<<<` implements (serial) function composition, the function `first` allows for parallel composition, and the function `arr` generates stateless transformations including re-arrangement of tuples as needed by `first`. It turns out, that all of these `combinators` maintain causality. They allow us to express all kinds of causal processes without feedback. If `f` and `mix` are causal processes, then we can translate the former `mix x (f x)` to

```haskell
arr (\x -> (x,x)) >>> second f >>> mix
```

where `second p = swap >>> p >>> swap

```haskell
  swap = arr (\(a,b) -> (b,a))
```

For implementation of feedback we need only one other combinator, namely `loop`.

```haskell
loop :: Generator a -> Generator a
```
The function \texttt{loop} feeds the output of type \texttt{c} of a process back to its input channel of the same type. In contrast to the \texttt{loop} method of the standard \texttt{ArrowLoop} class we must delay the value by one sample and thus need an initial value of type \texttt{c} for the feedback signal. Because of the way, \texttt{loop} is designed, it cannot run into deadlocks. In general deadlocks can occur whenever a signal processor runs ahead of time, that is, it requires future input data in order to compute current output data. Our notion of a causal process excludes this danger.

In fact, feedback can be considered another instance of the sharing problem and \texttt{loop} is its solution. For instance, if we want to compute a comb filter for input signal \texttt{x} and output signal \texttt{y}, then the most elegant solution in \texttt{Haskell} is to represent \texttt{x} and \texttt{y} by lists and write the equation \texttt{let y = x + delay y in y} which can be solved lazily by the \texttt{Haskell} runtime system. In contrast to that if \texttt{x} and \texttt{y} are signal generators, this would mean to produce infinitely large code since it holds

\[
y = x + \text{delay } y \\
= x + \text{delay } (x + \text{delay } y) \\
= x + \text{delay } (x + \text{delay } (x + \text{delay } y)) \\
\ldots
\]

With \texttt{loop} however we can share the output signal \texttt{y} with its occurrences on the right hand side. Therefore, the code would be

\[
y = \text{apply } (\text{mixFanout } >>> \text{second delay}) \ x \\
\text{where mixFanout} = \\
\text{arr } \ (\text{\(\lambda\)(a,b) \to (a+b,a+b)}) \ .
\]

Since the use of arrow combinators is somehow less intuitive than regular function application and \texttt{Haskell}'s recursive \texttt{let} syntax, there is a preprocessor that translates a special arrow syntax into the above combinators. Further on there is a nice abstraction of causal processes, namely commutative causal arrows [Liu et al., 2009].

We like to note that we can even express signal processes that are causal with respect to one input and non-causal with respect to another one. E.g. frequency modulation is causal with respect to the frequency control but non-causal with respect to the input signal. This can be expressed by the type

\[
\text{freqMod} :: \text{Generator } (\text{V } a) \to \\
\text{Causal } (\text{V } a) (\text{V } a) \ .
\]

In retrospect, our causal process data type looks very much like the signal generator type. It just adds a parameter to the transition function. Vice versa the signal generator data type could be replaced by a causal process with no input channel. We could express this by

\[
\text{type Generator } a = \text{Causal } () a
\]

where () is a nullary tuple. However for clarity reasons we keep \texttt{Generator} and \texttt{Causal} apart.

### 3.3 Internal parameters

It is a common problem in signal processing that recursive filters [Hamming, 1989] are cheap in execution, but computation of their internal parameters (mainly feedback coefficients) is expensive. A popular solution to this problem is to compute the filter parameters at a lower sampling rate [Vercoe, 2009; McCartney, 1996]. Usually, the filter implementations hide the existence of internal parameters and thus they have to cope with the different sampling rates themselves.

In this project we choose a more modular way. We make the filter parameters explicit but opaque and split the filtering process into generation of filter parameters, filter parameter resampling and actual filtering. Static typing asserts that filter parameters can only be used with the respective filters.

This approach has several advantages:

- A filter only has to treat inputs of the same sampling rate. We do not have to duplicate the code for coping with input at rates different from the sample rate.
- We can provide different ways of specifying filter parameters, e.g. the resonance of a lowpass filter can be controlled either by the slope or by the amplification of the resonant frequency.
- We can use different control rates in the same program.
- We can even adapt the speed of filter parameter generation to the speed of changes in the control signal.
- For a sinusoidal controlled filter sweep we can setup a table of filter parameters for logarithmically equally spaced cutoff frequencies and traverse this table at varying rates according to arcus sine.
- Classical handling of control rate filter parameter computation can be considered as
resampling of filter parameters with constant interpolation. If there is only a small number of internal filter parameters, then we can resample with linear interpolation of the filter parameters.

The disadvantage of our approach is that we cannot write something simple like lowpass \((\text{sine controlRate}) \cdot \text{(input sampleRate)}\) anymore, but with Haskell’s type class mechanism we let the Haskell compiler choose the right filter for a filter parameter type and thus come close to the above concise expression.

4 Related Work

Our goal is to make use of the elegance of Haskell programming for signal processing. Our work is driven by the experience, that today compiled Haskell code cannot compete with traditional signal processing packages written in C. There has been a lot of progress in recent years, most notably the improved support for arrays without overhead, the elimination of temporary arrays (fusion) and the Data-Parallel Haskell project that aims at utilising multiple cores of modern processors for array oriented data processing. However there is still a considerable gap in performance between idiomatic Haskell code and idiomatic C code. A recent development is an LLVM-backend for the Glasgow Haskell Compiler (GHC), that adds all of the low-level optimisations of LLVM to GHC. However we still need some tuning of the high-level optimisation and a support for processor vector types in order to catch up with our EDSL method.

In Section 2 we gave some general thoughts about possible designs of signal processing languages. Actually for many combinations of features we find instances: The two well-established packages Csound [Vercoe, 2009] and SuperCollider [McCartney, 1996] are domain specific untyped languages that process data in a chunky manner. This implies that they have no problem with sharing signals between signal processors, but they support feedback with short delay only by small buffers (slow) or by custom plugins (more development effort). Both packages support three rates: note rate, control rate and sample rate in order to reduce expensive computations of internal (filter) parameters. With the Haskell wrappers [Hudak et al., 1996; Drape, 2009] it is already possible to control these programs as if they were part of Haskell, but it is not possible to exchange audio streams with them in real-time. This shortcoming is resolved with our approach.

Another special purpose language is ChucK [Wang and Cook, 2004]. Distinguishing features of ChucK are the generalisation to many different rates and the possibility of programming while the program is running, that is while the sound is playing. As explained in Section 3.3 we can already cope with control signals at different rates, however the management of sample rates at all could be better if it was integrated in our framework for physical dimensions. Since the Haskell systems Hugs and GHC both have a fine interactive mode, Haskell can in principle also be used for live coding. However it still requires better support by LLVM (shared libraries) and by our implementation.

Efficient short-delay feedback written in a declarative manner can probably only be achieved by compiling signal processes to a machine loop. This is the approach implemented by the Structured Audio Orchestra Language of MPEG-4 [Scheirer, 1999] and Faust [Orlarey et al., 2004]. Faust started as compiler to the C++ programming language, but it does now also support LLVM. Its block diagram model very much resembles Haskell’s arrows (Section 3.2). A difference is, that Faust’s combinators contain more automatisms, which on the one hand simplifies binding of signal processors and on the other hand means, that errors in connections cannot be spotted locally.

Before our project the compiling approach embedded in a general purpose language was chosen by Common Lisp Music [Schottstaedt, 2009], Lua-AV [Smith and Wakefield, 2007], and Feldspar (Haskell) [programming group at Chalmers University of Technology, 2009].

Of all listed languages only ChucK and Haskell are strongly and statically typed, and thus provide an extra layer of safety. We like to count Faust as being weakly typed, since it provides only one integer and one floating point type.

5 Conclusions and further work

The speed of our generated code is excellent, yet the generating Haskell code looks idiomatic. The next step is the integration of the current low-level implementation into our existing framework for signal processing, that works with real physical quantities and statically checked physical dimensions. There is also a lot of room for automated optimisations by
 GHC rules, be it for vectorisation or for reduction of redundant computations of frac.

6 Acknowledgments
I like to thank the careful proofreaders of the draft for their valuable suggestions.

References


OpenDAW - a reference platform for GNU/Linux Audio

Daniel James
64 Studio Ltd.
P.O. Box 37, Freshwater.
Isle of Wight, Great Britain, PO40 9ZR
daniel@64studio.com

Abstract

Not being able to give a definitive answer to the question “Will my application or hardware work reliably with GNU/Linux?” presents a barrier to adoption by pro audio users. The OpenDAW platform offers a potential solution.

Keywords

GNU/Linux distributions, reference platform, hardware support, business models

1 Introduction

For servers, network devices or high performance computing clusters, it’s a redundant question to ask if a piece of hardware or a particular software component works with GNU/Linux. It’s no exaggeration to say that GNU/Linux is a standard operating system in these fields, so a lack of support for the Free Software platform usually indicates a business model based on vendor lock-in. In other fields, such as mobile, GNU/Linux may not be installed on the majority of devices yet, but it has become too significant to be ignored. In particular, the standardization of the Android platform, and the associated marketing push given to GNU/Linux by Google and its hardware partners, have perhaps done more to put Free Software into the hands of end users than the many GNU/Linux distributions have achieved in the last twenty years. Web browser surveys for January 2011 indicate that Android phones already account for one third of all GNU/Linux based Internet client devices, despite the fact that the Android platform has only been available to the public on a certified phone handset for just over two years.

The audio software world, in general, is different. Proprietary operating systems are deployed by the vast majority of users, with an unusually large number of Mac users compared to the general population. To give one example, the latest Sound on Sound magazine reader survey found that 58.4% of readers reported using a Mac, 54.7% reported using a PC, and only 1.6% reported not using a computer at all. This compares to web browser statistics for January 2011 suggesting that all Macs combined account for less than 7% of client devices.

What could be the reasons for such a high level of proprietary Mac adoption among audio users? It certainly isn’t technical superiority, despite the smug attitude among members of the Cupertino cult. Macs didn’t even have preemptive multitasking as standard until the launch of OS X in 2001. Before then, printing a large document on OS 9 often meant taking an enforced lunch break.

I would argue that perceived continuity and standardisation are more important to audio users than choice, or price/performance ratio. Apple has typically presented a very limited range of hardware choices, and yet this has somehow been presented as an advantage. Apple has not allowed its users to have a choice of hardware suppliers either, the notable exception being a brief period during the lifetime of System 7.

Apple hardware has often lagged behind PC hardware in terms of raw performance, for example towards the end of the PowerPC era, when the company was advertising the G5 as the ‘world’s fastest computer’ right until they dropped it, in favour of x86. (In the UK, Apple was forced to withdraw this bold claim by both TV and print advertising regulators in 2003/2004).

Although Apple successfully presents the image of continuity through marketing - using the name Mac for more than 27 years - in fact, the company has disrupted its own development community and user base several times as it jumped ship from one
hardware option to another, or when it abandoned its own operating system for a proprietary UNIX flavour. The switch from OS 9 to OS X was marketed as a continuity from 'nine' to 'ten', even though it was a major disruptive change for both audio developers supporting the Mac platform, and the audio users who were compelled to scrap their PowerPC machines. Forced obsolescence is not only expensive and inconvenient for the pro audio community; it is also a significant contributor to the global problem of e-waste.

| Change |
|-----------------|-----------------|
| Dropping the 68K CPU, introducing PowerPC |
| Dropping Nubus, introducing PCI | |
| Suppression of third-party Mac 'clones' | |
| Endorsement of Mac clones for System 7 | |
| Suppression of Mac clones from OS 8 onwards | |
| Dropping Old World ROM, introducing New | |
| Dropping Mac OS, introducing OS X | |
| Dropping the 'classic' GUI, introducing Aqua | |
| Dropping New World ROM, introducing EFI | |
| Dropping PowerPC, introducing x86 | |

**Figure 1: Some disruptive changes in Mac history**

Neither do I buy the idea that Apple is particularly sensitive to the needs of pro audio users. For all the support of Apple by pro audio customers, those users remain a small niche market of the somewhat larger niche of creative professionals, almost insignificant in corporate profit terms when compared to the revenue from disposable consumer products like the iPod, iPhone and iPad.

I would argue that it is the third-party audio software and hardware developer support of a particular platform that have made it popular with audio users, rather than anything that the proprietary operating system vendors have done. This phenomenon is not exclusive to the Mac. If it were not for Steinberg creating ASIO, there might not be any pro audio users running Windows at all.

Perhaps this is because in audio, users are not fault tolerant. We deal with once in a lifetime or never to be repeated events on a daily basis, and they happen in real time. Waiting a few seconds for a task to complete is not acceptable. This might be what makes audio users relatively conservative in their platform choice, sticking to Macs despite their limitations.

So we need to keep drawing the wider pro audio development community towards the Free Software platform. Unfortunately, the major commercial GNU/Linux distributions are about as interested in pro audio users as Apple or Microsoft are. The GNU/Linux server market may be worth billions of euros annually, but the special requirements of pro audio don't really figure in that market.

By learning from the lessons of continued Mac adoption among audio users, and the more recent upsurge of Android adoption among phone buyers, we can create a hardware, operating system and application ecosystem designed specifically by and for pro audio users.

### 2 The OpenDAW design

OpenDAW is a reference GNU/Linux distribution designed to create a minimal, stable and high performance platform for hardware manufacturers, system integrators and the application development community. It is also suitable for end users with some GNU/Linux experience. The emphasis is on providing a selection of known reliable packages with as little duplication of functionality as possible, in a standardized platform with continuity and long-term support. Hardware and software certification services are available from 64 Studio Ltd.

The base distribution is essentially a subset of Debian Squeeze amd64 with a real-time patched Linux kernel version 2.6.33 or later, using the proven design of 64 Studio distribution releases from 1.0 through to 2.1. The default desktop is GNOME, for continuity with these earlier 64 Studio releases.

Debian provides a very wide selection of packages, but a more important reason for selecting it as the basis of OpenDAW is its quality threshold rather than date-based release model. While Debian may be perceived as having a long release cycle, it was in fact only two years between the 5.0 'Lenny' and 6.0 'Squeeze' stable releases. This cycle length compares well with Windows and Mac minor releases. Windows XP and Mac OS X are both almost ten years old, typically having had a minor update or 'service pack' released every two years or so. (Windows XP users may be forced to upgrade to Windows 7
when they buy new hardware, because of forced obsolescence, but Debian offers continuity and the ease of performing full system upgrades on a running machine with apt).

The Linux kernel supports many more hardware architectures than either Windows or Mac OS X, and does not force users to change architecture. For example, Apple dropped the 68K processor with the introduction of the Power Mac in 1994, but this CPU is still supported in the Linux 2.2, 2.4 and 2.6 kernels.[4]

A timeline of Linux releases shows that not only does the kernel enjoy long periods of stability between major releases, but that the long overlap between major releases means that forced upgrades of production systems are unlikely.

![Figure 2: Timeline of Linux kernel releases. Source: Wikipedia (Creative Commons Attribution-ShareAlike License)](image)

3 Distributions and upstream developers

In the early years of the GNU/Linux distributions, between 1992 and 1998, the target audience was almost entirely made of developers. The principle of free-as-in beer code reuse was equitable because a user was likely to contribute additional code, creating a virtuous circle. The initial public releases of the KDE and GNOME desktop projects, in 1998 and 1999 respectively, enabled GNU/Linux for a non-developer audience. Some of these non-developers contributed to the community by offering informal help on mailing lists, writing documentation, or producing artwork. However, as installation methods became simpler, it became possible to be a GNU/Linux user without being an active member of the Free Software community. It was no longer necessary to join a user group to puzzle out technical problems, and some users brought their consumerist expectations from proprietary platforms.

As the proportion of non-contributing end users increased through the 2000's, it could be argued that the relationship between developers and end users has become less equitable. Financial contributions are passively solicited by some development projects, but anecdotal evidence suggests that these contributions rarely add up to much. The LinuxSampler annual report for 2009 lists one donation, of two euros.[5]

If only a tiny minority of end users donate voluntarily for Free Software, they disproportionately contribute, which is not equitable either. The alternative of developers providing direct support services is not always practical or desirable. Ironically, the better the software is, the less support that end users will need or pay for.

Distributions created by for-profit companies might actually make it harder for independent Free Software authors to redress the imbalance. Much of the value in these distributions is created by upstream developers who are not explicitly credited, let alone compensated.

Red Hat charges a compulsory subscription for its Enterprise distribution, but does not distribute this revenue to upstream authors, unless you count authors who are direct employees of Red Hat. At least Red Hat does employ a significant number of key developers, including real-time kernel contributors.
Figure 3: Screenshot showing a general lack of upstream developer credit in the Ubuntu Software Centre application. At least Apple gets credit in the caption of a package.

Figure 4: The 'Install – Free' button in distribution tools like the Ubuntu Software Centre might undermine efforts by upstream authors to raise revenue. Note the ambiguity about updates and licensing. Again, there’s no mention of the upstream developer, and no link to the upstream website.

The Android Market\[6\] offers a potential model for funding Free Software audio development. An application store model built into OpenDAW would enable end users to download certified applications for the platform, with full credit, a link to the upstream homepage, and optionally, payment. Developers who wished to release their apps free-as-in-beer on the store could still do so.

The GNU GPL and other Free Software licences do not prevent charging end users for software, as long as source code is available to those users. The problem of distributions which are non-crediting and non-revenue-contributing remains, without the use of GPL exceptions, which are themselves problematic. An application store offering GPL software would have to compete on some other level with free-as-in-beer distributions, perhaps on certification or support.

Another problem with an application store model is that end users do not typically pay for libraries, or infrastructure such as drivers. This puts developers of libraries or drivers who do not also code end user applications at a disadvantage.

An alternative example of upstream development funding is provided by the independently produced online game, Minecraft.\[7\] The developer of Minecraft directly asks users for a one-off payment, rising from 10 euros to 20 euros as the game is finished, providing an incentive to users to fund development early on. 10 euros isn’t much for a user to contribute, but it adds up when you have almost four and a half million users, around 30% of whom have paid for the game. Minecraft uses some open source components, and the developer has suggested that he will release the source code to the game at some unspecified date in the future. This delayed source release model has prevented GNU/Linux distributions from shipping the game, for the time being, but the revenue has enabled the developer to set up a company to secure the future of the software.

Pricing is difficult - how do we value the priceless gift of software freedom? Does it cheapen the gift to ask users for a small amount of money? I would like to hear the views of upstream authors on these issues.

4 Conclusion

GNU/Linux provides the greatest continuity of any generally available operating system, on the widest possible range of hardware. It therefore provides an excellent platform for long-lived audio deployments and products.

The OpenDAW platform provides a reference distribution of GNU/Linux specifically designed
for pro audio users, with a two year release cycle and five years of deployment support as standard.

Because full source code is available, commercial interests cannot force 'end of life' obsolescence on the platform. This makes long-term deployment more cost-effective, enables hardware re-use, and reduces the generation of e-waste.

OpenDAW is not a semi-closed type of open platform, like Android. Our aim at 64 Studio is for all packages in the reference distribution to be Free Software. We may still have to include non-free firmware if pro audio cards require it, since there are no known 'Free Hardware' pro audio cards (yet).

This initiative is not meant to colonise or eliminate other audio distribution projects; diversity leads to innovation. Rather, it is meant to provide a standard which can drive GNU/Linux adoption forward in the wider pro audio community.

References

[1] Wikipedia calculates mean averages of web client market share surveys:

[2] Operating system usage of Sound on Sound readers reported in media pack:

[3] Apple censured by UK advertising regulators for misleading claims:
http://www.macobserver.com/tmo/article/UK_Advertising_Standards_Authority_Upholds_Complaint_Against_Apples_G5_Adve/

[4] The Linux/Mac68k project:
http://www.mac.linux-m68k.org/

[5] LinuxSampler annual report:

[6] Android Market:
https://market.android.com/

[7] Minecraft:
http://www.minecraft.net/
Abstract

PulseAudio is becoming the standard audio environment on many Linux desktops nowadays. As it offers network transparency as well as other interesting features OS X won’t offer its users natively, it’s time to have a closer look on possibilities on how to port this piece of software over to yet another platform.

In the recent months, I put some effort into attempts to port PulseAudio to Mac OS X, aiming for cross-platform interoperability between hetero- 
gen audio host networks and enabling other features PulseAudio offers such as uPnP/AV streaming.

This paper will first give a short overview about how CoreAudio, the native audio system on Mac OS X, is structured, the various ways that can be used to plug into this system, and then focus on the steps it takes to port PulseAudio to Mac OS X.

Keywords

PulseAudio, Mac OS X, network audio, virtual audio driver, portability

1 CoreAudio essentials

1.1 IOAudio Kernel extensions

• The Darwin kernel is in charge of handling hardware drivers, abstracted via the IOKit API framework.

• The kernel’s representation of an audio device is an object derived from the IOAudioDevice base class, which holds a reference of an IOAudioEngine (or a derived type thereof).

• The kernel’s way of adding audio streams to a device is attaching objects of type IOAudioStream to an IOAudioEngine.

• The kernel’s API is only one way to provide an audio device to the system; the other is a plugin for the HAL (see above).

• Sample material is organized in ring buffers which are usually shared with the hardware.

• IOAudioEngines are required to report their sample rate by delivering exact timestamps whenever their internal ring buffer rolls over. The more precise, the better, as its userspace counterpart (the HAL, see below) can do better estimation of the device’s speed and approximate closer to the actual hardware sample pointer positions, resulting in smaller latencies.

1.2 HAL

The HAL is part of the CoreAudio framework and is automatically instantiated within the process image of each CoreAudio client application. During its startup, it scans for plugins in /Library/Audio/Plugins/HAL and this way offers the possibility of loading userspace implementations of audio drivers. The HAL is also in charge of interfacing to the IOAudio based kernel drivers and hence acts as their bridge to userspace clients.

1.3 AudioHardwarePlugins for HAL

Automatically loaded by the HAL code upon creation of an audio client, AudioHardwarePlugins are instantiated via the standard CFBundle load mechanisms. An interface must be implemented to provide the hooks needed by the HAL, and a full-fledged infrastructure of APIs for adding audio devices, streams and controls are available. Unlike kernel drivers, virtual drivers implemented as HAL plugin are working on a per-client base, so their implementations must care for mixing and inter-client operability themselves.

1.4 System Sound Server

This daemon is in charge for handling system- 

internal sound requests such as interface and alert sounds.

1.5 coreaudiod

coreaudiod is a system-wide daemon that gives home to the System Sound Server and
provides the AudioHardwareServices API for querying parameters of available audio drivers. The daemon also handles the default sound interface configuration on a per-user level\(^1\).

### 1.6 AudioUnits

AudioUnits are Mac OS X typical CFBundle s which can be installed user-wide or system-wide to fixed locations in the file system and which can be accessed by arbitrary applications with an standarized API for audio processing. They can also offer a graphical representation for parameter control and visualization. The two supported types of AudioUnit plugins are effect processors and virtual instruments.

### 2 Possible audio hooks

The purpose of this project is to be able to hook into the transport channels of all audio applications - including system sounds, if desired - and re-route audio through an either local or remote PulseAudio server connection.

Mac OS X officially offers a number of ways to access the audio material:

- A virtual sound card interface implemented as kernel driver which can either be configured as standard sound interface for all applications and/or system sounds. Applications may let the user decide which sound card to use for input and output sound rendering, but for those which don’t (like iTunes, QuicktimePlayer, iChat, ...), setting the system-wide default is the only option.

- A virtual sound card interface implemented as AudioHardwarePlugin for the HAL. The same rules as for the kernel versions apply: if an application doesn’t allow its user to choose the device for audio output, the system falls back to the configured default.

- An AudioUnit which is loaded by more advanced applications such as Logic. For application which don’t use this plugin interface, this is no option.

Another possible way of interaction is unofficial, somewhat hackish and based on the idea of library pre-loading for selected applications. Binaries are relaunched with their CoreAudio libraries temporarily replaced by versions which re-route audio differently. An example of this approach is the closed-source shareware utility AudioHijack\(^2\). More research is needed in order to find out whether this approach is also feasible for PulseAudio sound re-routing. At the time of writing, this option is not being investigated on.

### 3 PulseAudio on OS X

In order to bring PulseAudio to Mac OS X, some tweaks are needed to the core system, and some parts have to be re-developed from scratch.

#### 3.1 pulseaudiod

Porting the daemon is of course the main part of the work as it is the heart of the whole system other pieces connect to. Since a couple of versions, pulseaudiod, along with a selection of its essential modules, builds fine on OS X. Some adoptions were necessary to make this happen.

- `poll()` is broken since Mac OS X 10.3, dis-respecting the timeout argument and returning immediately if no file descriptor has any pending event. This was circumvented by using the `select()` syscall, just like PulseAudio does for Windows.

- `recv()` with `MSG_PEEK` does in fact eat up data from the given file descriptor. The workaround was to use a different `ioctl()` for this purpose.

- OS X lacks a proper implementation of POSIX locks but implements its own thing as defined in `Multiprocessing.h`. A version which uses them internally for the PulseAudio daemon was needed.

- clock functions work differently than on Linux, so a specialized version for the clock wrapper functions in PulseAudio was also necessary.

- Mac OS X offers a powerful API to give userland tasks high priority. This is essential for real-time applications just like PulseAudio, so an implementation using this API was added to the daemon.

- Some library PulseAudio uses are not suitable for OS X. Work on the build system was done to build some parts of the suite conditionally.


\(^2\)http://rogueamoeba.com/audiohijackpro/
3.2 CoreAudio device detection module
In order to make use of audio input and output devices CoreAudio knows about, a new pulseaudiod module was written which uses the CoreAudio specific callback mechanisms to detect hotplugged devices. For each detected device, a new module instance of module-coreaudio-device is loaded, and unloaded on device removal, accordingly.

This module is part of the official PulseAudio sources since some months and is called module-coreaudio-detect.

3.3 CoreAudio source/sink module
Loaded and unloaded by the house-keeping module module-coreaudio-detect, this module accesses the actual CoreAudio device, queries its properties and acts as translation layer between CoreAudio and PulseAudio. An important implementation detail is that code in this module has to cope with the fact that audio is exchanged between different threads.

This module is part of the official PulseAudio sources since many months and is called module-coreaudio-device.

3.4 Bonjour/ZeroConf service discovery module
Porting the dependency chain for Avahi (dbus, ...) wasn’t an easy and straight-forward task to do, and given the fact that Mac OS X features a convenient API for the same task, a new module for mDNS service notification was written. The code for this module purely uses Apple’s own API for announcing services to members of a local network.

This module is also part of the official PulseAudio source tree since a while and is called module-bonjour-publish.

3.5 Framework
On Mac OS X, libraries, headers and associated resources are bundled in framework bundles. As PulseAudio libraries and the libraries they are linked against are shared amongst several components for this project, they are all put in one single location (/Library/Frameworks/pulse.framework). This path was passed to the configure script as --prefix= directive when PulseAudio was built. A script (fixupFramework.sh) is in charge to resolve libraries dependencies which are not part of a standard Mac OS X installation. All libraries that are found to be dependencies for others are copied to the framework bundle and the tool install_name_tool which ships with Xcode is called to remap the path locations recursively.

3.6 PulseConsole
PulseConsole is a Cocoa based GUI application written in Objective-C that aims to be a comfortable configuration tool for PulseAudio servers, both local and remote instances. It offers a way to inspect and possibly modify details and parameters and a nice GUI for per-stream mixer controls and routing settings.

The plan is to make this tool as convenient as possible, also with GUIs for mixer controls, detailed server inspection and all the like. This will need some time to finish, but is actively developed already.

3.7 AudioHardwarePlugin for HAL
CoreAudio allows to add software plugins to register virtual sound interfaces. Such a plugin was developed for PulseAudio, with the following key features.

- Allows audio routing to both the local and any remote server instances.
- Multiple plugin instances communicate with each other over a distributed notification center. This is essential for sharing stream volume information.
- Each plugin instance announces itself to a system-wide message bus and can receive setup controls. This way, an existing connection to a sound server can be changed to some other server instance.
- The plugin is capable of creating multiple virtual sound interfaces. This can be helpful to cope with more than the standard stereo channel mapping. The configuration of which interfaces are created is controlled by the Preference Pane implementation (see below).

3.8 PulseAudio AudioUnits
For a more fine-grained way of routing specific audio paths through the PulseAudio daemon, AudioUnit plugins were developed. They connect to the local audio daemon and act as sound source and sound sink, respectively. All audio hosts that are capable of dealing with this type of plugin interface (ie, Apple Logic) can use this way of connecting specific sound paths to PulseAudio.
3.9 Virtual audio driver (kext)

Another way of adding an audio device driver to a system is hooking up a kernel driver for a virtual device and communicating with this driver from user space to access the audio material. This is what the virtual audio driver does.

This part of the project mostly exists for historical reasons, before the AudioHardwarePlugin approach was followed, which turned out to be much more interesting and feasible for the purpose. The code is still left in the source tree for reference and as proof-of-concept which might act as reference in the future.

Some of its key features include:

- support for any number of interfaces, featuring a configurable number of input and output channels each.
- userspace interface to control creation and deletion of interfaces.
- usage of shared memory between userspace and kernel space, organized as ring buffer.
- infrastructure to register a stream to userspace for each client that is connected to the interface. The framework for this code exists, but all attempts to actually make it work failed so far.

The concept of the driver model is to have one abstract IOService object (instance of PADriver) which is the root node for all other objects. Upon creation (at load time of the driver), the PADriver will be announced to the userspace.

A IOUserClient class named PADriverUserClient can be instanciated by user space, and commands can be issued to create new and delete instances of PAVirtualDevices. A PAVDevice is derived from IOAudioDevice and acts as a virtual audio device. To export audio functions, it has to have an PAEngine (derived from IOAudioEngine).

Depending on the type of audio engine (one for the mixed audio stream or one for each individual user client), the PAEngine can have one or many references to PAVirtualDevice, respectively.

Once a PAVirtualDevice is created, it is announced to the userspace, just like a PADriver. A userclient will create an object of type PAVirtualDeviceUserClient which can be used to issue commands specific to a PAVirtualDevice.

More information can be found in the repository at github.com.

3.10 virtual audio driver adapter module

Acting as counterpart of the virtual audio driver kernel module, a special purpose module for pulseaudiod takes notice of added and removed virtual sound card instances, maps the shared memory offered by the kernel and creates stream instances inside the PulseAudio daemon. The name for these streams are taken from the kernel space interface. As the kernel extension is not currently used anymore, this part of the source tree is also considered legacy.

3.11 Preference pane

The PulseAudio preference pane hooks itself into the standard Mac OS X system preferences and offers the following features:

- control the startup behaviour of the PulseAudio daemon
- configure authentication settings for network connections
- GUI for adding and deleting virtual sound interfaces

3.12 Component locations

Mac OS X organizes its file system contents in a quite different way than Linux installations. As described above, a framework is built in order to share the PulseAudio libraries amongst the various components. Components linking to the PulseAudio libraries have their linker settings configured to this path. Hence, the daemon and command line utilitily binaries as well as the loadable modules are found at the framework location as well, and if you want to access the PulseAudio command line tools (pacmd, paplay, ...) in the shell, the $PATH environment variable needs tweaking.

Apart from that, the other components are expected to be installed into specific locations so they can be found by the system. There will be documentation in the source tree to describe the exacte pathes.

3.13 Installer and packaging

A PackageMaker receipt has been created to generate installer packages that can be processed by the standard Mac OS X package installer, giving the user the general look and feel.
and procedure as most OS X add-ons. Depending on Apple’s policy for such tool suites, attempts might be made to publish the package via Apple’s application store.

3.14 License and source code
All parts of this suite are licensed under the GNU General Public License in version 2 (GPLv2).

The source code is accessible in the public git repository found at https://github.com/zoneque/PulseAudioOSX

4 Possible scenarios
Once the whole suite is developed as described and stable to a acceptable level, interesting audio routing scenarios are imaginable.

- Sound played back by iTunes can be routed through the virtual PulseAudio sound interface and from there be sent to an uPnP/AV audio sink.
- Sound played back by iDVD can be routed through the virtual PulseAudio sound interface and then be sent to an Airport Express using PulseAudio’s ROAP module. Mac OS X can not natively do that.
- A LADSPA proxy plugin could be developed to communicate with PulseAudio directly on Linux hosts. The stream for this plugin could be re-routed to a network host running PulseAudio on Mac OS X, and there be used as virtual input stream in Logic, hence allowing virtual instruments and effect plugins on Mac OS X to be used in LADSPA environments.
- Without any network interaction, simply routing all audio through the virtual PulseAudio sound interface allows users to control volumes of all connected audio clients individually (eg, silence annoying flash player in your browser, leveling audio applications that don’t offer a way to do this natively, etc).
- Soundcards that are not supported by ALSA driver can be accessed from Linux over the network, using a Mac OS X audio host.

5 Challenges and TODOs
This project is considered work in progress and is not yet finished. There are many details that need to be refined in order to make this toolchain fully usable. In particular, the following topics need to be addressed.

- Get the latency down. There are currently problems with untight scheduling in the PulseAudio client implementation, and too big buffer sizes.
- Considerations for multi-architecture libraries and binaries. XCode is not the problem in this regard, but the autoconf/automake build system is.
- The clocking model is subject to reconsideration. While things are comparatively easy in scenarios dealing with real hardware soundcards, it becomes more obfuscated in this virtual case as the PulseAudio daemon is the driving part for all clocks. That means that if audio is actually routed into a null-sink on the PulseAudio side, the virtual sound card will play at high speed, which might cause problems with audio applications that assume real-time playing.
- Cosmetic work on the GUI tools to give them the look of a nice tool users want to accept as part of their system. Currently, they look like debug tools for developers.
- Testing. Of course. The whole project is rather fresh, so it hasn’t seen a lot of testers yet.

6 Trademarks
Mac, and Mac OS, Mac OS X, iTunes, iDVD, Logic, Airport and Cocoa are trademarks of Apple Inc., registered in the U.S. and other countries.

7 Acknowledgements
My thanks go to the world-wide Linux audio community for providing ALSA and PulseAudio as sophisticated audio layers on Linux, making this project possible at all.
Abstract

We present Airtime[1], a new open source web application targeted at broadcast radio for automated audio playout. Airtime's workflow is adapted to a multi-user environment found at radio stations with program managers and DJs. Airtime is written in PHP and Python and uses Liquidsoap to drive the audio playout.

Keywords

Radio, broadcast, scheduled playout, web apps.

1 Introduction

Airtime is an open source web application targeted at radio stations who need to automate part or all of their audio playout. Its first release happened on Valentine's Day February 2011.

Airtime is funded by Sourcefabric, a non-profit organization dedicated to making the best open source tools for independent journalism around the world. One of it’s primary missions is to support independent media in developing democracies. Sourcefabric is currently funded by grants and has guaranteed funding for at least two more years. Within that time we expect to become self-sustaining.

In this paper we present a common workflow found at radio stations, then present how Airtime’s workflow matches that model. We then cover a number of non-workflow based features as well as the technology used to build both the web interface and backend player. We finish up with a preview of future development.

2 Radio Station Workflow

We have designed the interface workflow in a way that many multi-person radio stations work. The two roles present in radio stations related to Airtime are program managers and DJs. Program managers are responsible for organizing the schedule for the DJs and making sure that the schedule is fully booked. They usually plan out the schedule weeks or months in advance. DJs are responsible for preparing and presenting the audio during their assigned time slots (“time slots” are also known as “shows”). If the show is live, quite often DJs will bring their own equipment for playout such as turn tables, CDs, or iPods. If the show is automated, the DJ has the responsibility to fill in their show with audio.

3 Airtime Overview

Before we present the Airtime workflow, we present a few of the key concepts in the application: shows, playlists, and roles.

3.1 Shows

A “show” in Airtime corresponds to a block of time allocated to a DJ. It is also a container for audio clips. Shows can be assigned to one or more users, in which case only those users are able to modify the audio within that show. It is possible to create repeating shows on a daily, weekly, bi-weekly, or monthly basis.
3.2 Playlists

Airtime also has playlists, which can be inserted into a show. Playlists can be created before the shows have been scheduled and can be reused. Playlists and shows are completely separated – if a user schedules a playlist inside a show and then deletes the playlist, the schedule still has its own copy of the song list and playout will not be affected.

3.3 Roles

Airtime has three roles: admin, host, and guest. The “admin” role corresponds to the program manager job; this role has the ability to add, change, or delete shows. They also have the rights of a DJ.

The “host” role is equivalent to a DJ. They have the ability to create playlists and schedule them within the shows they have been assigned.

The “guest” role is a read-only role that allows someone to log in and see what is going on without being able to change anything.

4 Airtimes Workflow

The expected workflow for Airtime works as follows: the program manager logs in under the admin role and creates the shows in the calendar for all the DJs. Repeating shows can be scheduled on a daily, weekly, or bi-weekly, or monthly basis. The interface in the calendar is very similar to Google Calendar, where the user has the ability to move shows around by drag and drop as well as resize shows with the mouse to change their length.

The DJs log in at their leisure, upload their audio, use the audio library to create playlists, and add their playlists to a show. Any uploaded audio files are automatically scanned for metadata and additional metadata is retrieved from online databases. Replay gain is calculated on the audio files to normalize the output volume.

A status area at the top of the screen displays what song and show is currently playing along with timing and progress information. A more detailed list of the upcoming audio tracks can be viewed on the “Now Playing” screen, which also allows you to see the full list of planned audio for any given day. Any breaks of silence are displayed in red.

Shows that have already played cannot be removed, as this information is typically needed for various regulation purposes.

The backend audio player looks to see what show is scheduled for a specified time and starts playing it. It is completely disconnected from the web interface in that it fetches all the information it needs via HTTP requests and downloads a copy of the music it needs to play.

5 Non-workflow Features

The non-workflow features available in Airtime are internationalization and live show recording.

5.1 Internationalization

The Airtime interface can be internationalized into any language.

5.2 Show Recording and Archiving

Airtime ships with a separate application that hooks into Airtime's schedule which will record the audio during a live show if the user requests it. The audio is saved to a file, and inserted back into the audio database with metadata attached. These audio files can then be replayed again in future shows.

6 Technology

Airtime is written in PHP using the Zend Framework and Propel as the ORM layer. The web interface makes heavy use of jQuery and various jQuery plugins. The playout engine is Liquidsoap controlled by Python scripts. By default we output to both the sound card via
We currently only support the Linux operating system at the moment, which is mainly due to the fact that Liquidsoap is primarily supported on *UNIX platforms.

6.1 Design of the Playout System

The scripts used to drive Liquidsoap are collectively called “pypo” for Python PlayOut. These scripts were developed in conjunction with Open Broadcast in Switzerland. There are three separate processes which drive the playout:

1. Liquidsoap
2. Pypo-fetch
3. Pypo-push

Liquidsoap is an open source programmable audio stream engine for radio stations. It expects to always be playing something. We have written custom Liquidsoap scripts to drive the playout based on what Airtime users have scheduled. The Liquidsoap developers have been kind enough to add functionality for our playout model.

Pypo-fetch is responsible for fetching the playout schedule and downloading the music tracks before playout starts. There are configuration values for how far in advance to start downloading the audio as well as how long to keep the audio after the playout has occurred.

Pypo-push is responsible for controlling Liquidsoap and switching the playlist at the right time. It connects to Liquidsoap via a local telnet connection and switches between playlists using the queuing technology found in Liquidsoap.

Each of these programs is installed as a daemon via daemontools under a separate Linux user named “pypo”.

7 Future Development

The first release of Airtime has been made for one narrowly defined use case. In the coming year we are planning to develop the additional functionality shown below.

7.1 Very Near Term (3 months)

7.1.1 Scheduling Webstreams

The ability to automatically connect to webstream at a certain time and rebroadcast it.

7.1.2 Jingle Support

Users have requested a quick and easy way to add jingles to a playlist.

7.1.3 AutoDJ (Smart/Random Playlists)

Automatically generate playlists based on certain criteria.

7.1.4 RDS Support

RDS is the technology that displays the name of the song on your radio.

7.2 Mid-term (3-6 months)

7.2.1 Advertising Support

We plan to make Airtime understand the difference between ads and songs. The advertising manager will be able to put ads in the schedule with time boundaries within which those ads must be played. Ads will have different rights than audio and cannot be removed by someone without “advertising manager” rights.

7.2.2 RESTful API

Allow 3rd party applications to get the data out of the database via a REST interface. This would allow others to create other views of the data, such as a Web widget which would display the currently playing audio and display the upcoming schedule.

7.2.3 Playlist Import/Export

This is the ability to export a playlist to a file and import it back in.
7.2.4 **Airtime/Newscoop Integration**

Newscoop is Sourcefabric’s enterprise newsroom software. Integrating with this would allow a station to run its web site and control its playout with an integrated set of tools.

7.2.5 **SaaS Hosting**

We plan on offering a hosted version of Airtime.

7.3 **Longer Term (6 months – 1 year)**

7.3.1 **Live Shows**

We are planning to support live shows by allowing 3rd party playout software to access the audio files through a FUSE filesystem. We are also planning on implementing a “live mode” in the browser to allow a DJ to play songs on-demand.

7.3.2 **Graphical Crossfading Interface**

Display the waveform for an audio file in the browser and allow the user to drag and drop the crossfade points with their mouse and preview it.

7.3.3 **Smartphone/Tablet Interface**

Allow users to create playlists and schedule them on their favorite smartphone or tablet.

7.3.4 **Networked Stations**

Allow stations to share content with each other.

8 **Conclusion**

Airtime is under active development by three developers, a graphic designer, a QA engineer, and a manager. We are engaged with radio stations around the world to listen to feedback and make the most useful project possible. Since it is open source, outside developer participation is welcome in the project. You can try out Airtime right now by going to the demo site[2].

References


[2] Airtime demo site: [airtime-demo.sourcefabric.org](airtime-demo.sourcefabric.org)
A position sensing midi drum interface

Andrew Baxter
64 Blades Street
Lancaster, LA1 1SQ
andy@ganglion.me

Abstract
This paper describes my attempts at making a cheap, playable, midi drum interface which is capable of detecting not only the time and velocity of a strike, but also its location on the playing surface, so that the sound can be modulated accordingly. The design I settled on uses an aluminium sheet as the playing surface, with piezo sensors in each corner to detect the position and velocity of each strike. I also discuss the electronics (arduino based), the driver software, and the synths I have written which use this interface.

Keywords
Drum interface, MIDI, Position sensitive, Supercollider, DIY

1 Introduction
Midi drum interfaces are now a widely available piece of consumer electronics. However, in most cases they are only capable of reproducing a small range of the sounds you can make using a real drum. A significant weakness of these interfaces is that they do not give any indication of where on the playing surface you have struck them, so they are limited to playing a single sample or synthesised sound across the whole surface.

In this paper, I will describe my attempts at making a cheap, playable, midi drum interface which is capable of feeding information on where you have struck it to a synthesiser, which can modulate the sound accordingly, in two dimensions. There already a few devices coming out at the high end of the market which have similar abilities; however my aim here (apart from having a bit of fun myself building it), was to produce a design which is simple and cheap enough for any competent hobbyist to build, using widely available components.

2 Research
Before settling on the current design, I tried out a couple of other ideas for how to make such a drum pad.

My original idea was to use a sheet of conductive rubber as the playing surface, with a voltage put across it alternating between the north-south and east-west axes, so that a strike at a given point would correspond to a particular voltage pair. The simplest form of this idea would require the sticks to have wires on them, so is not really practical, but I discovered that there is a form of conductive rubber which lowers its conductivity sharply under pressure. This gave me the idea of making a sandwich with the voltage gradient sheet on the top, then a layer of pressure sensitive rubber, then an aluminium sheet electrode under both. Strikes to the top surface would, I hoped, produce small regions of lowered conductivity in the middle sheet, transferring the voltage at that point from the top sheet to the bottom electrode.

I constructed a prototype of this design, and managed to get it to sense position to a degree,
decided in the end that the results weren't consistent enough to be worth carrying on with this idea. Also, this design was fairly expensive (due to the cost of the pressure sensitive sheet), and lacked a reliable way of sensing the velocity of a strike.

The next idea, which I owe to Alaric Best, was to suspend a metal sheet in some kind of frame with vibration sensors around the edge, and detect the time of flight of pressure waves from the strike point to each sensor, and use this to triangulate the position of the strike.

I built a prototype of this using piezoelectric sensors, but was unable to get the sensors to detect pressure waves at anywhere near high enough time resolution.

However, during the testing, I noticed that the strength (rather than the timing) of the signal from the piezo sensors varied according to how close the strike was to each sensor (with one sensor in each corner of the sheet). In other words, on the time scale I was able to sense at, the piezos were simply detecting the transferred pressure of the strike at each mounting point. This gave me the idea for the current design.

3 Design and construction

The current physical design of the pad is as follows (see also illustrations 2 and 3):

Illustration 2: Corner view of the pad, showing rubber buffers and bolts. You can also just see the edge of the piezo sensor on the lower buffer.

Illustration 3: The pad connected to the driver circuit and a laptop

- The playing surface is a square sheet of aluminium.
- This is suspended between foam rubber buffers in a wooden frame. The buffers support the sheet above and below at each corner.
- The two halves of the frame are held together by bolts near each corner.
- Holes drilled in the sheet allow the bolts to pass through the sheet without touching it, so that it can move freely with respect to the frame.
- Under each corner of the sheet is a piezo-electric sensor, mounted above the lower buffer, in such a way that all the pressure from that corner of the sheet is transferred through the sensor. The distribution of strike pressure between these sensors indicates the position of the strike.
- Coax cables are used to bring the signals from the sensors out to a circuit board, where they are detected by an Arduino microcontroller board and fed back to a computer through USB bus.
- The whole thing rests on a soft foam rubber pad to reduce the effect of vibrations.

Full instructions on how to build one of these pads are available on the web.¹

4 Electronics

The electronics for the pad are pretty straightforward — I used an Arduino microcontroller board to detect the voltage pulses from the piezos; the only additional electronics was a simple voltage source to provide a false

¹http://www.instructables.com/id/A-position-sensitive-midi-drum-pad/
ground at half the arduino’s 5v analogue input range. This allows the board to detect negative going as well as positive going pulses from the piezos. The voltage range from the piezos matches the arduino’s analogue input range well enough that no additional amplification or attenuation is needed in practice.

The schematic for the circuit is shown in illustration 4.

Illustration 4: Schematic for the pad’s input circuit.

5 Driver software

The driver software for the pad is in two parts – a small firmware program on the arduino, which feeds basic strike data back to a laptop through its usb cable, and a longer program on the laptop which calculates the position and velocity of each strike from this, and sends this information to a software MIDI channel. All the software is available on the web.

5.1 Arduino firmware

The arduino firmware works as follows:

- On startup, the four analogue inputs are read and the base readings stored.
- The four analogue inputs are then read every 100us. The base reading for each input is subtracted to give the signal level.
- If the signal level on any input exceeds a trigger level, then the program starts a measurement cycle.

The measurement cycle goes like this:

The signal levels on each input are read every 100us for a set number of readings (currently 10).
- At each reading, the absolute value of the signal level on each input is added to a sum for that input.
- At the end of the measurement cycle, the summed values for each sensor are sent as a comma separated text string back to the laptop for further processing.

There is then a delay (currently 30 ms) to prevent re-triggering on the same strike, after which the program starts waiting for the next strike.

5.2 Midi mapper

The raw data from the arduino is then interpreted by a python program on the laptop (the midi mapper).

There are two phases to using this program. First, it needs to be calibrated with a set of strikes at 13 known positions on the pad (which are marked on the playing surface, as you can see in Illustration 1). The raw sensor values and known x-y position for each strike are recorded in an array.

```
def mapCurve(p,s1,s2,s3,s4):
    # p is an array of 7 coefficients; s1..4 are the raw sensor readings
    k2,k3,k4,l1,l2,l3,l4=p # give names to the coefficients.
    # the k coefficients allow for different sensitivities of the sensors.
    f1=s1 # first k coefficient is always 1
    f2=s2*k2
    f3=s3*k3
    f4=s4*k4
    # the l coefficients allow for irregularities in the physical construction of the pad
    x=(l1*f1+l2*f2+l3*f3+l4*f4)/(f1+f2+f3+f4)
    return x # the mapped coordinate (either x or y)
```

Text 1: Python code for the position mapping equation

http://ganglion.me/synpad/software/
Once the last calibration reading has been taken, these readings are used to fit a simple equation which is based on a rough physical model of the pressures transferred through the sheet. The code which implements this equation is shown in the frame 'Text 1' above. This is done separately for the x and y coordinates, producing two sets of coefficients which can be used to turn incoming strike data into x-y coordinates. The algorithm used to fit the data is the least squares function from the 'scipy' python library.

In the second phase, the coefficients from the calibration are used with the equation to determine x-y coordinates for live strike data. These are then sent as a stream of midi events on a software midi channel to a synthesiser. The way the coordinates are encoded into a midi stream is as follows:

- first, 'set controller' events are sent on controllers 70 and 71, corresponding to the x and y coordinates.
- then, a 'note on' event is sent, with the velocity proportional to the sum of all the raw sensor values, and the note number equal to the x coordinate.

This information can then be used by a synthesiser to create a sound which varies according to the x, y and velocity coordinates of the strike.

6 Sound synthesis

In principle, the midi stream for the pad could be fed into any drum-like soft-synth that is capable of modulating the sound according to midi controller values. (By drum-like I mean that the synth should not require a note off event to end each strike sound.)

However, in practice I decided to use the Supercollider audio synthesis language to construct the synths for the drum. Other similar environments, such as csound or pure data, could have been used, but Supercollider seemed to offer the greatest level of control and flexibility, and suits my way of thinking as a programmer. (Once I had got my head round its syntactic quirks!)

The supercollider code I have written is available on the web. It is in two parts – the first (drummidi.sc) listens for midi events on a channel and uses them to trigger a synthdef with the right x and y parameters. The second part (synpad.sc) is a set of synthdefs which have been written for this interface. This code is still in a pretty crude state, which works well enough for experimenting with different sounds, but wouldn't really be suitable for a live performance situation.

One of the synthdefs I wrote is reproduced in frame 'Text 2'. It takes 3 variable parameters – the velocity and x-y coordinates – and converts these into a percussive sound whose timbre varies across the pad.

7 Results

In this section I will write about how the drum performs in practice, starting with the physical construction, then looking at the interface's playability, accuracy and ease of use, and finally discussing the synths I wrote to play it through.

---

3 http://www.scipy.org/
4 http://supercollider.sourceforge.net/
5 http://ganglion.me/synpad/software/
6 A video of the pad in use is avalible on the web. See http://ganglion.me/synpad/ for a link to this.
7.1 Physical construction

Actually building the pad was fairly easy. All you need to make one are a few cheap materials, some very basic carpentry and metalwork skills (cutting and drilling wood and aluminium sheet), and some simple tools. Mounting the piezos and soldering the contacts to them was a bit awkward, but no more than that.

Similarly, the electronics construction skills you need are pretty minimal, as the arduino board is doing most of the work.

7.2 Playability

The pad is fairly easy to play, with either fingers or felt headed timpani sticks. The biggest problem is the height of the frame above the board, which can make it a bit awkward to reach the edge of the playing surface.

7.3 Accuracy

The accuracy of strike detection is good enough to get some reasonable results. The pad senses velocity pretty well, although there is a lower cutoff which makes it hard to play very soft notes. The consistency of the position sensing is not bad – if you hit the pad repeatedly in the same spot, the position stays constant with velocity to about 5%. There is a degree of distortion in the mapping between pad positions and detected coordinates, but this error is not so much of a problem in practice, as you can adapt your playing to compensate.

Drift from the calibrated mapping during playing is small enough not to be a problem in practice. The pad would probably need recalibration at the start of a performance though, especially if it had been handled roughly during transportation.

In the time domain, the triggering delay (latency) imposed by the arduino firmware is about 1msec. I have not tried to measure the latency of the midimapper program, but in practice the latency of the combined system (firmware plus midi mapper plus synths plus sound card latency) is good enough that the strike sounds appear immediate to my ear.

7.4 The synths

Writing modulateable synths which sound good has proved to be the most difficult part of this project. The first thing I tried was to play a sample of a snare drum through a resonant low pass filter, with the x-coordinate controlling the filter cutoff, and the y-coordinate controlling the resonance. This produces some interesting effects, and is fun to play with. The drawback is that it is hard to make strongly rhythmic patterns with it: because the filter is resonant and the cutoff varies quite rapidly across the pad surface, it’s hard to hit close enough to the same point to repeat a given sound consistently – the sounds appear to the ear like a series of separate tones rather than variations of a single sound.

My next idea was to make something that was based on a more consistent base tone, with the strike coordinates modulating its timbre. This is the synthdef reproduced in frame 'Text 2' above. It is based on pink noise filtered through a comb delay line and then a non-resonant low pass filter. One coordinate controls the resonance of the delay line (the comb frequency is fixed), and the other controls the cutoff frequency of the low pass filter. This produces a nice synthy sound, with the timbre varying from noisy to ringing in one dimension, and from muted to bright in the other. This is the synth I used for the online demo video of the drum7.

Some other things I tried:

- working through the percussion section of the ‘Synth Secrets’ articles from Sound On Sound magazine8. I managed to make some half decent percussion sounds like this (though cymbals are tricky). The difficult part was more in working out a meaningful way of modulating the sound across the pad. Because these synths have many variables, any of which could be used as modulation parameters, it's hard to decide what combination of variables to vary to get a nice result.
- Feeding audio samples into an FFT and operating on them in frequency space in various ways. I was hoping that this

7See http://ganglion.me/synpad/ for a link to the video.
8See http://www.soundonsound.com/sos/allsynthsecrets.htm
would produce a series of modulateable effects which could be applied to any base sample, making for a rich palette of sounds. However the results were a bit disappointing, probably due more to my lack of experience with supercollider and audio synthesis in general than anything else.

Overall, I think the basic concept of modulating a synthesised sound across the playing surface is good, and I've enjoyed writing and playing with some simple synths. At the same time, I've also come to realise that to there's a lot more to writing synths from the ground up than I had originally thought, and producing a range of effects good enough for live performance could involve a fair amount more work.

8 Similar work

In this section, I mention a few projects / products which are working in a similar space.

8.1 Korg Kaoss Pad

This is superficially similar to the pad I have made, in that it has a square playing surface which you can use to control sounds in 2 dimensions. However its function is quite different – it doesn't have a velocity sensing function and its role is as an effects processor for sounds generated elsewhere, rather than an instrument in its own right. It sells for around 300 USD.

8.2 Mandala Drum from Synesthesia Corp

This has a circular pad with 128 position sensing rings arranged concentrically on it. It can only modulate the sound in one dimension rather than two, but the design appears to be much more polished and playable than mine. It is also sold with a library of sound effects tailored for the drum, some of which emulate the sound of a real snare drum. They sell for about 350 US dollars.

8.3 Randall Jones's MSc thesis on 'Intimate Control for Physical Modelling Synthesis'

This uses a 2D matrix of copper conductors arranged perpendicularly on either side of a rubber sheet. Each north-south conductor carries a signal oscillating at a different frequency, and the east-west conductors pick up these signals by capacitative connection, to an extent which varies according to where pressure has been applied to the rubber sheet. The signals are generated and received by a standard multi-channel audio interface, and interpreted in software on a computer.

This project is probably the closest to mine in its intent – it's a midi drum surface with two dimensional position and velocity sensing. It has also been designed in a way that most people could build one themselves. As far as playability goes it looks to be way ahead of mine – it is multitouch, can detect continuous pressure changes as well as instantaneous strikes, and the profile of the frame around the head is lower, which should make it more comfortable to play. It is also self-calibrating, so doesn't need to be set up again every time you play.

Its main drawback is complexity and the associated cost. There is a lot of signal processing going on to produce the admittedly impressive result. The fact that it depends on a separate sound card also makes it fairly expensive compared to my project.

9 Improvements and future directions

Here I discuss ideas for where I might take this project in the future.

If I was to stick with the current basic design, there are a few simple improvements I could try, to make it more playable and responsive. For example, instead of a single wooden frame, the aluminium sheet could be held in place by metal discs bolted to the base board at each corner. This would make it easier to reach the pad surface when playing.

There might also be small improvements possible in the firmware and the midimapper, to

---

9 http://en.wikipedia.org/wiki/Kaoss_Pad
10 http://synesthesiacorp.com/about.html
improve the consistency and accuracy of the position sensing. However, since having looked at Randall Jones's design, I'm thinking that this is much closer to the direction things ought to be going. So, in the future I would be more interested in developing something which offers a similar degree of responsiveness and playability, either by adapting his design to make it simpler and cheaper to build, or using some other technique.

I also have a few ideas about developing the associated software to make it more powerful and easier to set up and use. For example, it wouldn't be hard to build a graphical interface which would let you swap between different synths, rather than having to evaluate supercollider code to do this, as at present.

One idea I would like to have a go at is to make some synths with several variable parameters, then find a way of assigning parameter-sets (presets) to different points on the pad's surface. It should then be possible to use some kind of mapping algorithm to smoothly vary the parameters of the synth across the pad's surface in such a way that at each preset-point, the result sounds like the preset you have assigned to that point, and in between points the sound smoothly morphs from one preset to another.

10 Conclusion

In conclusion, I think that the basic concept of creating a 2 dimensional playing surface for synth percussion sounds is sound, and has a lot of potential. I have been fairly successful in achieving my aim of making such a surface using cheap, simple components. However, this particular design has a number of flaws, such its lack of multi-touch and continuous pressure sensing abilities, the need for calibration, and a degree of physical awkwardness in playing it, due to the height of the mounting frame.

I am planning to continue developing the idea, and may put some more work into refining this design, but in the long run something like Randall Jones's design looks like a better way forward for this kind of interface.

On the software side, the hardest part is producing good synth sound effects for the pad. Because this kind of interface is quite new, it is necessary to write new synths for it from the ground up rather than using existing ones. There is also a lot of room for improvement in the supporting software more generally, and I am planning to put some more work into this in the future.

11 Acknowledgements

Thanks go to Alaric Best and Dave Leack of Veraz Ltd. for their thoughts on and encouragement with the project. Also to the writers of the Superollider audio synthesis language, without which the task of writing a synth for the drum would have been ten times harder.

References

Towards more effective mapping strategies for digital musical instruments

Patrick MCGLYNN
Music Department, NUI Maynooth
Maynooth,
Co. Kildare, Ireland
patrick.j.mcglynn@nuim.ie

Abstract
This paper contends that the design of digital musical instruments for live performance and composition has been hindered by the tendency to create novel applications which fail to offer musicians access to the more perceptually-significant aspects of electronic music.

Therefore, as a means of addressing this problem, this paper promotes the establishment of a more intelligent approach to the construction of digital musical instruments: one that is informed by relevant studies in human-computer interaction, cognitive psychology and product design.

Keywords
Mapping, digital musical instruments, real-time performance, Csound, controllers

1 Introduction

Recent commercial interest in the gestural control of home entertainment and portable computer systems has led to the rapid development of affordable and elegant systems for human computer interaction. The computer music community is responding to these advances with enthusiasm, producing a steady stream of musical applications which make use of the Apple iPad\(^1\), Nintendo Wii Remote\(^2\), Microsoft Kinect\(^3\), and similar devices.

One trait shared by all of these interfaces is their tendency to employ implicit communication – a term coined by Italian cognitive scientist Cristiano Castlefranchi to describe interactions which exploit “perceptual patterns of usual behavior and their recognition” [1]. Examples of implicitly understood actions are the ‘swipe’ and ‘pinch’ gestures common to Apple iOS devices (which are analogous to page-turning and shrinking/expanding respectively). One potentially-destructive side-effect of these intuitive interfaces is the misconception that all applications should adhere to this simplistic approach – a paradigm whose limitations are especially destructive when it comes to applications for musical performance.

The expressive range and musical potential of these music applications is, being extremely fair, varied. Without an informed approach to designing these musical performance systems, developers haphazardly juxtapose musical functions in the hope of providing an instantly-gratifying musical experience. There exists an urgent need to discuss design issues which can potentially separate ‘serious’ electronic musical endeavors from the ever-growing selection of ‘novelty’ music applications.

2 Designing a digital musical instrument

In their book New Digital Musical Instruments: Control and Interaction Beyond the Keyboard, Miranda & Wanderley deconstruct the process of designing an electronic performance system into five distinct steps:

1. Decide upon the gestures which will control it
2. Define the gesture capture strategies which work best
3. Define the accompanying synthesis algorithms / music software
4. Map the sensor outputs to the music control
5. Decide on the feedback modalities available, apart from the sound itself (visual, tactile, kinesthetic, etc.) [2]

Depending on the circumstances, these questions will often be dealt with in a different order, with the available technology or musical goal providing the answer to several of them before the design process even begins. Assuming that every possible situation will have its peculiarities and idiosyncrasies, a general guide to assist designers in selecting

---

\(^{1}\) http://www.apple.com/ipad/
\(^{2}\) http://www.nintendo.com/wii/console/controllers
\(^{3}\) http://www.xbox.com/kinect/
the best possible gestures and mapping strategies would be a valuable complement to this approach.

3 Mapping

In a digital musical instrument (DMI), mapping describes the manner in which data gathered by the input device(s) is related to the system’s musical parameters. The importance of selecting or devising an appropriate mapping scheme cannot be understated – effective and elegant systems can lead to “a more holistic performance exploration of the parameter space” [3] and essentially define the “essence” of a DMI [2].

This is not to say that a performance system should necessarily be overly simplistic or immediately accessible. In the study of Human Computer Interaction (HCI), it has been suggested that “an efficiency-focused approach to interaction may no longer suffice: it needs to be complemented by knowledge on the aesthetic aspects of the user experience” [4]. In a musical context, an expressive interface design must accommodate the capacity to practise, learn, make mistakes, and develop skill.

Literature devoted specifically to the study of mapping schemes is sparsely available for a number of reasons – the theoretically limitless combinations of devices and musical goals that a musician might seek to accommodate render the discussion of general mapping principles difficult and of limited use.

Therefore, a more detailed vocabulary which enables musicians to assess their own situations is essential.

3.1 Mapping in digital musical instruments

Musical mapping schemes are generally classified according to the number of parameters over which the user can exert control at once. The most commonly-used terms are ‘convergent mapping’ and ‘divergent mapping’. Convergent mapping employs a number of devices to control a single parameter (‘many-to-one’) whereas devices which use divergent mapping operate several parameters at once (‘one-to-many’). It has been suggested that human operators expect such complex schemes and ultimately find analytical ‘one-to-one’ interactions more rewarding and intuitive [3]. Most acoustic musical instruments can be thought of as combining elements of both of these schemes.

3.2 Mapping in product design

Outside of a musical context, mapping schemes for human-technology interaction are more efficiency-focused and hence easier to discuss. In The Design of Future Things, Donald A. Norman encourages designers to utilize what he refers to as ‘natural mappings’ wherever possible (citing the oft-inconsistent positioning of hobs and their controls on a cooker as an example). In this context, it is preferable that controls should be laid out “in a manner spatially analogous to the layout of the devices they control” and that the principle can be extended to “numerous other domains” including sound [1]. With this consideration in mind, it is surprising how many supposedly-intuitive musical performance systems opt for the most convenient or visually-appealing layout for their controls, rather than considering the perception of the user.

In the same volume, Norman provides a summary of the essential design considerations discussed. His ‘rules of interaction’ state that interactive technology should:

1. Provide rich, complex, and natural signals
2. Be predictable
3. Provide a good conceptual model
4. Make the output understandable
5. Provide continual awareness, without annoyance
6. Exploit natural mappings to make interaction understandable and effective

It should be stressed that these considerations are clearly intended for functional applications which can be effectively used almost instantly - a description which cannot reasonably accommodate the law of diminishing returns that we associate with successful musical endeavors. However, they do provide a model of simplicity and efficiency which can be useful to bear in mind while working on more complex multimedia environments.

4 Towards systematic mapping

Adopting a methodical approach towards identifying and classifying the types of data generated by a particular device allows the interface designer to assess its suitability for various musical tasks in a logical, efficient manner.
4.1 Classifying performance data according to complexity

This high-level approach to mapping separates performance data into three distinct groups in ascending order of complexity:

A. Raw data (on/off, fader positions, X/Y/Z co-ordinates, etc.)
B. Symbolic / semiotic data (predefined actions associated with various postures, themselves represented by different combinations of the raw data)
C. Gestural data (predefined actions associated with dynamic movement)

An alternative way of phrasing this concept would be to think of group A as simple data, group B as elements of that data being placed in the context of one another to create more complex cues, and group C as the resulting cues being placed in the context of one another. Groups B and C can thus be thought of as constructing both the gestural ‘vocabulary’ and ‘grammar’, respectively, and play a crucial role in defining the usability and character of a given performance system. Future publications from this project will focus intently on the development of effective schemes to populate and manipulate these groups.

Input options classified according to these varying degrees of complexity can subsequently be allocated to different musical tasks, depending on the sophistication of control deemed necessary.

4.2 Degrees of freedom

In order to populate groups B and C as defined above, a system for assembling more complex commands from the raw data is required. By listing the available sensors and/or triggers of an input device and noting their inherent degrees of freedom a comprehensive ‘toolbox’ of available data can be defined.

Devices which offer one degree of freedom include buttons, switches, faders and dials. While the latter two examples can provide more detailed data than simple on/off controls (0-127 in MIDI, for example) they are still incapable of representing more than one piece of information at a time. Devices which offer two degrees of freedom include touch-sensitive keyboards (sending both note on/off and a velocity value) and simple X/Y pads (horizontal and vertical co-ordinates).

One must be careful not to confuse the terms ‘degrees of freedom’ with ‘dimensions’ – while the two terms are often used interchangeably they describe different aspects of a device [5]. An X/Y pad is typically referred-to as a two-dimensional surface and assumed to have two corresponding degrees of freedom. However a true 2D surface in fact provides three degrees of freedom – the X and Y co-ordinates of an object and the rotation of that object on the Z-plane (the Reactable, developed within the Music Technology Group at the Universidad Pompeu Fabra, implements Z-plane rotation as a central control device [6]). Add to this the possibility of multitouch, or placing multiple objects upon the plane, and the possible array of data to be obtained expands rapidly.

4.3 Augmenting simple control data

While not strictly provided by the device itself, the introduction of computer intelligence in the gathering of this data allows us to introduce a number of subtle factors which can expand the complexity of even the most basic input devices.

One example is an ‘InUse’ variable which becomes true whenever a control has been accessed by the performer. By simply comparing the current value of a controller to a value stored one frame/sample ago, we can infer whether or not the state of a device has changed. A MIDI fader using this technique now provides two degrees of freedom – fader position (0-127) and ‘isCurrentlyBeingMoved’, or equivalent (0-1).

By lengthening the comparison times, we can also determine if the fader has been moved since \( n \) – this technique can be employed, for example, to terminate a musical event associated with a fader if it has not been interacted with for a certain period of time (analogous to the art of ‘spinning plates’, where elements require a certain amount of stimulation or energy input to survive).

Further to this, another variable can be added to keep track of the amount or intensity of user interaction with a device. This can take the form of a ‘counter’ which increases every time a change is detected in the value/state of the device (and perhaps decreases over time if the device is idle). An example of this exact technique is outlined below in section 5.

4.4 Combining simple control data

Using combinations of simple input data is a simple and efficient way to expand the number of options available to a user – the most familiar
example being the ‘shift’ key common to QWERTY keyboards which changes accompanying keystrokes to upper-case.

The computer mouse, as described by Buxton’s ‘3-state model of graphical input’, provides a more advanced example [7]. While the mouse prompts simple X/Y movements, these are interpreted differently depending upon which of the aforementioned three states the user has selected – state 0 is inactive (mouse is out-of-range or away from surface), state 1 is for pointer-movement, and state 2 is for the dragging and moving of objects and is invoked when the user holds down the mouse button. Needless to say, modern mouse, touchpad and trackball devices have greatly expanded this range through extra buttons and gesture recognizers.

However, caution should be advised when accommodating multiple layers of functionality within a single device – this increases the cognitive load upon the user and can compromise the building-up of associations required for intuitive and skilled performance [8].

5 An example application

A mapping experiment was conducted in order to examine the viability of the classifications as outlined in 4.1. The goal of the experiment was to replace the control surface of a hardware synthesiser with a different interface and demonstrate, via the application of the ideas outlined above, how alternate mapping schemes can extrapolate the functionality of a digital musical instrument from a performance perspective.

The process can be split into three parts – examining the original device, replicating the functionality of the device in a software model, and extending control of the model to a new interface.

5.1 Drone Lab V2

Drone Lab V2 is a four-voice analog synthesizer and effects processor by Casper Electronics4. It was designed to facilitate the creation of “dense, pulsing drones” by allowing the user to individually de-tune the bank of oscillators. The resulting phase-cancellation creates rhythmic textures which can be exaggerated and emphasised using the built-in filters and distortion effect.

This synthesiser was chosen for several reasons – the most pertinent being the lack of a predefined technique for controlling and ‘playing’ its noise-based output. The absence of any performance conventions facilitates the objective analysis of exactly how useful our classifications can be when designing an interface for an innovative or experimental performance system.

Figure 1: The original hardware version of Drone Lab V2

5.2 Csound implementation

In order to experiment with different control schemes, the synthesis model was implemented in the Csound audio programming environment. Both the signal flow chart and the comprehensive sound examples provided on the Casper Electronics website allowed for the construction of a software emulator which duplicates quite closely the output of the original Drone Lab V2.

Figure 2: Open-source plans for Drone Lab V25

4http://casperelectronics.com/finished-pieces/drone-lab/drone-lab-v2/

5http://casperelectronics.com/images/finishedpieces/droner/V2/KitLayoutLabels.jpg
One important issue that must be highlighted is the reduced functionality of the software implementation. While the general behaviour and audio output of the synthesiser are quite close to the original, the new GUI-based interface limits the user to manipulating a single parameter at a time via the mouse. User precision is also hindered by the lack of any tactile feedback and the need to rely exclusively on the visual display in order to discern the state of the various parameters.

This problem is not unique to this project by any means – it could be argued that the tendency of software-based instruments to rely heavily on GUI-based controls is one of the main contributors to a lack of clearly-defined performance practice, not to mention the difficulty encountered by accomplished musicians when trying to develop an intuitive sense of these instruments.

5.3 Wii Remote control

The Nintendo Wii Remote is a motion sensing game controller which can function independently from the Wii console itself. Its ability to communicate wirelessly through the Bluetooth protocol and three-axis accelerometer has made the Wii Remote an extremely popular tool in the computer music community. Most of the major audio programming languages feature some level of support for the device and several dedicated interface-management programs (such as OSCulator\(^7\) and DarwinRemote\(^8\)) allow the conversion of Wii Remote data into other useful formats such as MIDI or OSC.

For this project, the Windows-based program GlovePIE\(^9\) was used to receive data from the Wii Remote and convert it into values readable by Csound (sent via OSC messages). One function was created for each type of performance data (as outlined in this paper) in order to illustrate the practical benefits of a systematic approach to parameter-mapping.

5.4 Mapping gestural data to instrument parameters using the group system

The ‘A’ button along with the plus and minus buttons on the Wii Remote were used to turn on/off the various oscillators and switch between them respectively. This is an example of raw data (group A) being used as a simple selection and triggering system.

GlovePIE provides access to ‘roll’ and ‘pitch’ variables which are derived from the angular velocity of the Wii Remote’s X and Y axes respectively. These were mapped to simultaneously control the frequency and volume of the oscillators. While these are both raw data / group A attributes, their combined values determine the overall behaviour of a single oscillator and accordingly allow the user to associate certain postures with the sound they produce. As such, the two values used in this mapping scheme depend upon each other and together represent an example of symbolic / semiotic data (group B).

While these mappings provide adequate access to the parameters concerned, they do not necessarily alter the way the instrument is played. The distortion volume and amount were mapped using a more complex setup which changed the behaviour of the sound considerably.

Using techniques described in section 4.3, a function was set up which continually checked if a certain threshold was exceeded by the combined acceleration of the Wii Remote’s three axes. If the overall movement of the user was violent enough to exceed this value, a global variable called agitation was augmented. When the movement was less pronounced, the agitation value would gradually decrease.

\(^{6}\) http://quetcsound.sourceforge.net/

\(^{7}\) http://www.osculator.net/

\(^{8}\) http://sourceforge.net/projects/darwiin-remote/

\(^{9}\) http://glovepie.org/glovepie.php
Mapping the agonistic value to the distortion effect created a very effective control metaphor – users could easily associate the violent shaking or agitation of the Wii Remote with the resulting disintegration of clarity in the audio output, perhaps due to associations with real-world instruments which exhibit similar behaviour when shook vigorously (certain percussion instruments and electric guitar, for example). As it analyses complex cues in the context of previous actions, this final mapping can be placed within group C – gestural data.

6 Conclusion

Taking inventory of the data generated by a controller interface is an essential part of assessing its suitability for a specific musical task. However, one can easily underestimate the interdependence of certain variables and hence proceed to design a strictly functional device with no distinct characteristics other than to respond to various switches and faders (albeit virtual ones).

By categorising controller data according to how it may be used, as opposed to where it is coming from, we can avoid simply replicating the behaviour of physical controllers, escape unnecessary performance paradigms, and move towards the development of more complex, elegant and satisfying interactive performance systems.

7Acknowledgements

The author would like to express his sincere gratitude towards Dr. Victor Lazzarini and Dr. Gordon Delap for their endless enthusiasm, support and inspiration throughout this research. Also special thanks to Dr. Charles Markham for his invaluable help approaching the field of gestural control.

This research project is funded by the John and Pat Hume Scholarship programme at NUI Maynooth.

References

Abstract
This paper discusses the new LLVM bitcode interface between Faust and Pure which allows direct linkage of Pure code with Faust programs, as well as inlining of Faust code in Pure scripts. The interface makes it much easier to integrate signal processing components written in Faust with the symbolic processing and metaprogramming capabilities provided by the Pure language. It also opens new possibilities to leverage Pure and its LLVM-based JIT (just-in-time) compiler as an interactive frontend for Faust programming.

Keywords
Functional programming, Faust, Pure, LLVM, signal processing.

1 Introduction
Pure and Faust are two functional programming languages which are useful in creating signal processing applications of various kinds. The two languages complement each other. While Faust is a statically typed domain-specific language for creating numeric signal processing components which work at the sample level [7], Pure is a dynamically typed general-purpose language tailored for symbolic processing, which can be used to tackle the higher-level components of computer music and other multimedia applications [2]. Both Pure and Faust have compilers producing native code; however, while Faust is batch-compiled, Pure has a just-in-time (JIT) compiler and is typically used in an interactive fashion, either as a standalone programming environment or as an embedded scripting language in other environments such as Pd.

Faust has had a Pure plugin architecture for some time already. However, this has been somewhat awkward to use since the programmer always has to go through an edit-compile-link cycle in order to create a shared library object of the Faust plugin, which can then be loaded in Pure. The new LLVM bitcode interface makes this much easier.

LLVM, the “Low-Level Virtual Machine”, is an open-source cross-platform compiler backend available under a BSD-style license [4], which forms the backbone of a number of important compiler projects, including Apple’s latest incarnations of the GNU compiler collection as well as clang, a new C/C++ compiler featuring various improvements over gcc [1]. In the past few years, the LLVM project has attracted a number of compiler writers who are retargeting compilers and interpreters to use LLVM. Google’s Python compiler “UnladenSwallow” [9] and David A. Terei’s backend for the Glasgow Haskell Compiler [8] are just two notable examples. Pure has used LLVM as its backend since the very first Pure release in 2008.

LLVM exposes a fairly low-level code model (somewhere between real assembler and C) to client frontends. This makes it a useful target for signal processing languages where the generation of efficient output code is very important. Thus an LLVM backend has been on the wishlist of Faust developers and users alike for some time, and this backend was finally designed and implemented by Stéphane Letz at Grame in 2010. The new backend is now available in the “faust2” branch in Faust’s git repository [5]. During a brief visit of the author at Grame last year, we started working on leveraging the LLVM support of Faust and Pure to build a better bridge between the two languages. This paper reports on the results of this cooperation.

In Sections 2 and 3 we first take a brief look at the Faust and Pure sides of the new Pure-Faust bridge, respectively, discussing Faust’s LLVM backend and Pure’s LLVM bitcode loader. In Section 4 we walk the reader through the steps required to run a Faust module in Pure. Section 5 explains how to inline Faust code in Pure programs. A complete example is shown in Section 6. Section 7 concludes with some remarks on the current status of the interface and possible
future enhancements.

2 The Faust backend

To take advantage of Faust’s new LLVM backend, you currently need a fairly recent snapshot of the “faust2” branch of the compiler in the Faust git repository [5]. Install this on your system with the usual `make && sudo make install` commands.

The `-lang llvm` option instructs Faust to output LLVM bitcode (instead of the usual C++ code). Also, you want to add the `-double` option to make the compiled Faust module use double precision floating point values for samples and control values. So you’d compile an existing Faust module in the source file `example.dsp` as follows:

```bash
faust -double -lang llvm example.dsp -o example.bc
```

The `-double` option isn’t strictly necessary, but it makes interfacing between Pure and Faust easier and more efficient, since the Pure interpreter uses double as its native floating point format. This option is also added automatically when inlining Faust code (see Section 5).

Note that LLVM code actually comes in three distinct flavours:

- as an internal representation (`LLVM IR`), i.e., a C++ data structure in main memory used in most LLVM client applications such as compilers and interpreters;
- as a compact binary code (`LLVM bitcode`), which provides a serialized form of LLVM IR which can be passed from one LLVM application to another, either in main memory or as a disk file;
- and, last but not least, as a kind of human-readable assembler source code (`LLVM assembler`), which is rarely used directly in LLVM applications, but very useful for documentation purposes.

A description of the LLVM assembler code format can be found on the LLVM website [4], but the code examples shown in this paper should be rather self-explanatory, at least for C programmers. For the sake of a simple example, let us consider the following little Faust module which mixes two input signals and multiplies the resulting mono signal with a gain value supplied as a control parameter:

```faust
gain = nentry("gain", 0.3, 0, 10, 0.01);
process = + : *(gain);
```

From this the Faust compiler creates an LLVM bitcode file containing several LLVM assembler routines whose call interfaces are listed in Figure 1. If you want to see all the gory details, you can put the above code into a text file `example.dsp` and run Faust as follows to have it print the complete LLVM assembler code on standard output:

```bash
faust -double -lang llvm example.dsp
```

At the beginning of the LLVM module you see some data type definitions and global variables. The assembler routines roughly correspond to the various methods of the `dsp` classes Faust creates when generating C++ code. The central routine is `compute_llvm` which contains the actual assembler code for the signal processing function implemented by the Faust program. This routine gets invoked with the pointer to the `dsp` instance, the number of samples to be processed in one go (i.e., the block size), and the vectors of input and output buffers holding the sample values. The other routines are used for managing and inspecting `dsp` instances as well as the interface to the control variables (the “user interface” of a `dsp` in Faust parlance).

Note that the names of the assembler routines are currently hard-wired in Faust. Thus an LLVM application which wants to link in the Faust-generated code must be prepared to perform some kind of name mangling to make multiple Faust dsps coexist in a single LLVM module. This is handled transparently by Pure’s bitcode loader.

3 The Pure bitcode interface

The nice thing about LLVM bitcode is that it can be readily loaded by LLVM applications and compiled to native machine code using the LLVM JIT compiler. This doesn’t require any special linker utilities, only the LLVM library is needed.

The Pure compiler has a built-in bitcode loader which handles this. The ability to load Faust modules is in fact just a special instance of this facility. Pure can import and inline code written in a number of different programming languages supported by LLVM-capable compilers (C, C++ and Fortran at present), but in the following we concentrate on the Faust bitcode loader which has special knowledge about the Faust language built into it.
%struct.UIGlue = { ... }
%struct.dsp_llvm = type { double }
@fSamplingFreq = private global i32 0
@example = private constant [8 x i8] c"example\00"
@gain = private constant [5 x i8] c"gain\00"

define void @destroy_llvm(%struct.dsp_llvm* %dsp) { ... }
define void @delete_llvm(%struct.dsp_llvm* %dsp) { ... }
define %struct.dsp_llvm* @new_llvm() { ... }
define void @buildUserInterface_llvm(%struct.dsp_llvm* %dsp, %struct.UIGlue* %interface) { ... }
define i32 @getNumInputs_llvm(%struct.dsp_llvm*) { ... }
define i32 @getNumOutputs_llvm(%struct.dsp_llvm*) { ... }
define void @classInit_llvm(i32 %samplingFreq) { ... }
define void @instanceInit_llvm(%struct.dsp_llvm* %dsp, i32 %samplingFreq) { ... }
define void @compute_llvm(%struct.dsp_llvm* %dsp, i32 %count, double** noalias %inputs, double** noalias %outputs) { ... }
define void @init_llvm(%struct.dsp_llvm* %dsp, i32 %samplingFreq) { ... }

Figure 1: Outline of the LLVM assembler code for a sample Faust module.

Loading a Faust bitcode module in Pure is easy. You only need a special kind of import clause which looks as follows (assuming that you have compiled the example.dsp module from the previous section beforehand):

using "dsp:example";

The above statement loads the bitcode module, links it into the Pure program, and makes the Faust interface functions callable from Pure. It also mangles the function names and puts them into their own Pure namespace, so that different Faust modules can be called in the same Pure program. Note that it’s not necessary to supply the .bc bitcode extension, it will be added automatically. Also, the bitcode module will be searched on Pure’s library search path as usual. You can repeat this statement as often as you want; the bytecode loader then checks whether the module has changed (i.e., was re-compiled since it was last loaded) and reloads it if necessary.

On the Pure side, the callable functions look as shown in Figure 2. (You can also obtain this listing yourself by typing show -g example::* in the Pure interpreter after loading the module.) Note that despite the generic struct_dsp_llvm pointer type, the Pure compiler generates code that ensures that the dsp instances are fully typechecked at runtime. Thus it is only possible to pass a dsp struct pointer to the interface routines of the Faust module it was created with.

The most important interface routines are new, init and delete (used to create, initialize and destroy an instance of the dsp) and compute (used to apply the dsp to a given block of samples). Two useful convenience functions are added by the Pure compiler: newinit (which combines new and init) and info, which yields pertinent information about the dsp as a Pure tuple containing the number of input and output channels and the Faust control descriptions. The latter are provided in a symbolic format ready to be used in Pure; more about that in the following section. Also note that there’s usually no need to explicitly invoke the delete routine in Pure programs; the Pure compiler makes sure that this routine is added automatically as a finalizer to all dsp pointers created through the new and newinit routines so that dsp instances are destroyed automatically when the corresponding Pure objects are garbage-collected.

4 Running Faust dsps in Pure

Let’s now have a look at how we can actually use a Faust module in Pure to process some samples. We present this in a cookbook fashion, using the example.dsp from the previous sections as a running example. We assume here that you already started the Pure interpreter in interactive mode (just run the pure command in the shell to do this), so the following input is meant to be typed at the ‘>’ command prompt of the interpreter.
Step 1: Compile the Faust dsp  We already discussed this in Section 2. You can execute the necessary command in the Pure interpreter using a shell escape as follows:

```plaintext
> ! faust -double -lang llvm example.dsp -o example.bc
```

Step 2: Load the Faust dsp in Pure  This was already covered in Section 3:

```plaintext
> using "dsp:example";
```

Please note that the first two steps can be omitted if you inline the Faust program in the Pure script, see Section 5.

Step 3: Create and initialize a dsp instance  After importing the Faust module you can now create an instance of the Faust signal processor using the `newinit` routine, and assign it to a Pure variable as follows:

```plaintext
> let dsp = example::newinit 44100;
```

Note that the constant `44100` denotes the desired sample rate in Hz. This can be an arbitrary integer value, which is available in the Faust program by means of the `SR` variable. It’s completely up to the dsp whether it actually uses this value in some way (our example doesn’t, but we need to specify a value anyway).

The dsp is now fully initialized and we can use it to compute some samples. But before we can do this, we’ll need to know how many channels of audio data the dsp consumes and produces, and which control variables it provides. This information can be extracted with the `info` function, and be assigned to some Pure variables as follows:

```plaintext
> let k,l,ui = example::info dsp;
```

Step 4: Prepare input and output buffers  Pure’s Faust interface allows you to pass Pure double matrices as sample buffers, which makes this step quite convenient. For given numbers \( k \) and \( l \) of input and output channels, respectively, we’ll need a \( k \times n \) matrix for the input and a \( l \times n \) matrix for the output, where \( n \) is the desired block size (the number of samples to be processed per channel in one go). Note that the matrices have one row per input or output channel. Here’s how we can create some suitable input and output matrices using a Pure matrix comprehension and the `dmatrix` function available in Pure’s standard library:

```plaintext
> let n = 10; // the block size
> let in = {i*10.0+j | i = 1..k; j = 1..n};
> let out = dmatrix (l,n);
```

In our example, \( k = 2 \) and \( l = 1 \), thus we obtain the following matrices:

```plaintext
> in;
{11.0,12.0,13.0,14.0,15.0,16.0,17.0,18.0,19.0,20.0; 21.0,22.0,23.0,24.0,25.0,26.0,27.0,28.0,29.0,30.0}
> out;
{0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0}
```

Step 5: Apply the dsp to compute some samples  With the `in` and `out` matrices as given above, we can now apply the dsp by invoking its `compute` routine:

```plaintext
> example::compute dsp n in out;
```

This takes the input samples specified in the `in` matrix and stores the resulting output in the `out` matrix. Let’s take another look at the output matrix:

```plaintext
> out;
{9.6,10.2,10.8,11.4,12.0,12.6,13.2,13.8,14.4,15.0}
```

Note that the `compute` routine also modifies the internal state of the dsp instance so that
a subsequent call will continue with the output stream where the previous call left off. Thus we can now just keep on calling `compute` (possibly with different in buffers) to compute as much of the output signal as we need.

**Step 6: Inspecting and modifying control variables** Recall that our sample dsp also has a control variable `gain` which lets us change the amplification of the output signal. We've already assigned the corresponding information to the `ui` variable, let's have a look at it now:

```plaintext
> ui;
vgroup ("example", [nentry #<pointer 0xd81820> ("gain", 0.3, 0.0, 10.0, 0.01)])
```

In general, this data structure takes the form of a tree which corresponds to the hierarchical layout of the control groups and values in the Faust program. In this case, we just have one toplevel group containing a single `gain` parameter, which is represented as a Pure term containing the relevant information about the type, name, initial value, range and stepsize of the control, along with a `double` pointer which can be used to inspect and modify the control value. While it's possible to access this information in a direct fashion, there's also a `faustui.pure` module included in the Pure distribution which makes this easier. First we extract the mapping of control variable names to the corresponding `double` pointers as follows:

```plaintext
> using faustui;
> let ui = control_map $ controls ui; ui;
{"gain" => #<pointer 0xd81820>}
```

The result is a Pure record value indexed by control names, thus the pointer which belongs to our `gain` control can be obtained with `ui!"gain"` (note that `!' is Pure's indexing operator). There are also convenience functions to inspect a control and change its value:

```plaintext
> let gain = ui!"gain";
> get_control gain; 0.3
> put_control gain 1.0;
()
> get_control gain; 1.0
```

Finally, let's rerun `compute` to get another block of samples from the same input data, using the new `gain` value:

```plaintext
> example::compute dsp n in out;
> out;
{32.0, 34.0, 36.0, 38.0, 40.0, 42.0, 44.0, 46.0, 48.0, 50.0}
```

As you can see, all these steps are rather straightforward. Of course, in a real program we would probably run `compute` in a loop which reads some samples from an audio device or sound file, applies the dsp, and writes back the resulting samples to another audio device or file. This can all be done quite easily in Pure using the appropriate addon modules available on the Pure website.

Also note that you could change the Faust source at any time, by editing the `example.dsp` file accordingly and returning to step 1. You don't even need to exit the Pure interpreter to do this.

## 5 Inlining Faust code

The process sketched out in the preceding section can be made even more convenient by inlining the Faust program in Pure. The Pure interpreter then handles the compilation of the Faust program automatically, invoking the Faust compiler when needed. (The command used to invoke the Faust compiler can be customized using the `PURE_FAUST` environment variable. The default is `faust -double`; the `-lang llvm` option is always added automatically.)

To add inline Faust code to a Pure program, the foreign source code is enclosed in Pure's inline code brackets, `%< ... %>`. You also need to add a `dsp` tag identifying the contents as Faust source, along with the name of the Faust module (which, as we've seen, becomes the namespace into which the Pure compiler places the Faust interface routines). The inline code section for our previous example would thus look as follows:

```plaintext
%< -*- dsp:example -*-
gain = nentry("gain", 0.3, 0.0, 10.0, 0.01);
process = + : *((gain));
%
```

You can insert these lines into a Pure script, or just type them directly at the prompt of the Pure interpreter. If you later want to change the Faust source of the module, it is sufficient to just enter the inline code section again with the appropriate edits.

## 6 Example

As a more substantial but still self-contained example, Figures 3 and 4 show the source code of a complete stereo amplifier stage with bass, treble, gain and balance controls and a dB meter. The dsp part is implemented as inlined Faust code, as discussed in the previous section. The Pure part implements a Pd “tilde” object named
This requires the pd-pure plugin loader (available as an addon module from the Pure website) which equips Pd with the capability to run external objects written in Pure. A sample patch showing this object in action can be seen in Figure 5.

A complete discussion of this example is beyond the scope of this paper, but note that the \texttt{amp\_dsp} function of the program is the main entry point exposed to Pd which does all the necessary interfacing to Pd. Besides the audio processing, this also includes setting the control parameters of the Faust dsp in response to incoming control messages, and the generation of output control messages to send the dB meter values (also computed in the Faust dsp) to Pd.

By using the interactive live editing facilities provided by pd-pure, we could now start adding more sophisticated control processing or even change the Faust program on the fly, while the Pd patch keeps running. We refer the reader to the pd-pure documentation for details [3].

7 Conclusion

The facilities described in this paper are fully implemented in the latest versions of the Pure and Faust compilers. We also mention in passing that Pure doesn’t only support dynamic execution of mixed Pure and Faust code in its interactive interpreter environment, but Pure scripts containing Faust code can also be batch-compiled to native executables. This eliminates the JIT compilation phase and thus makes programs start up faster.

The present interface is still fairly low-level. Except for the automatic support for handling Faust control variables, the call interfaces to the Faust routines follows the code generated by Faust very closely. In the future, we might add more convenience functions at the Pure level which make the operation of Faust dsps easier for the Pure programmer.

Another interesting avenue for further research is to employ Pure as an interactive frontend to Faust. This is now possible (and in fact quite easy), since Pure allows Faust source to be created under program control and then compiled on the fly using Pure’s built-in \texttt{eval} function. Taking this idea further, one might embed Faust as a domain-specific sublanguage in Pure. This would provide an alternative to other interactive signal processing environments based on Lisp dialects such as Snd-Rt [6].

Acknowledgements

Many thanks to Stéphane Letz for his work on the Faust LLVM interface which made this project possible in the first place. Special thanks are also due to Yann Orlarey for inviting me to Grame to work on improving our arsenal of functional signal processing tools.

References

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

// bass and treble frequencies
bass_freq = 300;
treble_freq = 1200;

// bass and treble gain controls in dB
bass_gain = nentry("bass", 0, -20, 20, 0.1);
treble_gain = nentry("treble", 0, -20, 20, 0.1);

// gain and balance controls
gain = db2linear(nentry("gain", 0, -96, 96, 0.1));
bal = hslider("balance", 0, -1, 1, 0.001);

// stereo balance
balance = *(1-max(0,bal)), *(1-max(0,0-bal));

// generic biquad filter
filter(b0,b1,b2,a0,a1,a2) = f : (+ ~ g)
with { f(x) = (b0/a0)*x+(b1/a0)*x'+(b2/a0)*x''; g(y) = -2*(a1/a0)*y-(a2/a0)*y''; }

/* Low and high shelf filters, straight from Robert Bristow-Johnson’s "Audio
EQ Cookbook". */
low_shelf(f0,g) = filter(b0,b1,b2,a0,a1,a2)
with { S = 1; A = pow(10,g/40); w0 = 2*PI*f0/SR;
alpha = sin(w0)/2 * sqrt( (A + 1/A) + (1/S - 1) + 2 );
b0 = A*( -1 - (A+1)*cos(w0) + 2*sqrt(A)*alpha );
b1 = 2*A*(- (A+1)*cos(w0) );
b2 = A*( (A+1)*cos(w0) + 2*sqrt(A)*alpha );
a0 = 2*A*( - (A+1)*cos(w0) );
a1 = (A+1) - (A-1) *cos(w0) + 2*sqrt(A)*alpha;
a2 = (A+1) - (A-1) *cos(w0) - 2*sqrt(A)*alpha; }

high_shelf(f0,g) = filter(b0,b1,b2,a0,a1,a2)
with { S = 1; A = pow(10,g/40); w0 = 2*PI*f0/SR;
alpha = sin(w0)/2 * sqrt( (A + 1/A) + (1/S - 1) + 2 );
b0 = A*( -1 - (A+1)*cos(w0) + 2*sqrt(A)*alpha );
b1 = 2*A*(- (A+1)*cos(w0) );
b2 = A*( (A+1)*cos(w0) + 2*sqrt(A)*alpha );
a0 = 2*A*( - (A+1)*cos(w0) );
a1 = (A+1) - (A-1) *cos(w0) + 2*sqrt(A)*alpha;
a2 = (A+1) - (A-1) *cos(w0) - 2*sqrt(A)*alpha; }

// the tone control
tone = low_shelf(bass_freq,bass_gain)
 : high_shelf(treble_freq,treble_gain);

// envelop follower (1 pole LP with configurable attack/release time)
t = 0.1; // attack/release time in seconds
g = exp(-1/(SR*t)); // corresponding gain factor
env = abs : *(1-g) : + ~ *(g) : linear2db;

// dB meters for left and right channel (passive controls)
left_meter(x) = attach(x, env(x) : hbargraph("left", -96, 10));
right_meter(x) = attach(x, env(x) : hbargraph("right", -96, 10));

// the main program of the Faust dsp
process = (tone, tone) : ( _*gain, _*gain) : balance
 : (left_meter, right_meter);

Figure 3: Amplifier plugin, Faust part.
// These are provided by the Pd runtime.
extern float sys_getsr(), int sys_getblksize();
// Provide some reasonable default values in case the above are missing.
sys_getsr = 48000; sys_getblksize = 64;

// Get Pd’s default sample rate and block size.
const SR = int sys_getsr;
const n = sys_getblksize;

using faustui, system;

amp dsp = k,l,amp with
// The dsp loop. This also outputs the left and right dbmeter values for
// each processed block of samples on the control outlet, using messages of
// the form left <value> and right <value>, respectively.
amp in::matrix = amp::compute dsp n in out $$
    out,[left (get_control left_meter),right (get_control right_meter)];
// Respond to control messages of the form <control> <value>. <control> may
// be any of the input controls supported by the Faust program (bass,
// treble, gain, etc.).
amp (c@_ x::double) = put_control (ui!str c) x $$ x;
end when

// Initialize the dsp.
dsp = amp::newinit SR;
// Get the number of inputs and outputs and the control variables.
k,l,ui = amp::info dsp;
ui = control_map $ controls ui;
{left_meter,right_meter} = ui!!["left","right"]; // Create a buffer large enough to hold the output from the dsp.
out = dmatrix (l,n);
end;

Figure 4: Amplifier plugin, Pure part.

Figure 5: Amplifier plugin, Pd patch.
Abstract

This paper discusses the use of Python for developing audio signal processing applications. Overviews of Python language, NumPy, SciPy and Matplotlib are given, which together form a powerful platform for scientific computing. We then show how SciPy was used to create two audio programming libraries, and describe ways that Python can be integrated with the SndObj library and Pure Data, two existing environments for music composition and signal processing.

Keywords

Audio, Music, Signal Processing, Python, Programming

1 Introduction

There are many problems that are common to a wide variety of applications in the field of audio signal processing. Examples include procedures such as loading sound files or communicating between audio processes and sound cards, as well as digital signal processing (DSP) tasks such as filtering and Fourier analysis [Allen and Rabiner, 1977]. It often makes sense to rely on existing code libraries and frameworks to perform these tasks. This is particularly true in the case of building prototypes, a practise common to both consumer application developers and scientific researchers, as these code libraries allows the developer to focus on the novel aspects of their work.

Audio signal processing libraries are available for general purpose programming languages such as the GNU Scientific Library (GSL) for C/C++ [Galassi et al., 2009], which provides a comprehensive array of signal processing tools. However, it generally takes a lot more time to develop applications or prototypes in C/C++ than in a more lightweight scripting language. This is one of the reasons for the popularity of tools such as MATLAB [MathWorks, 2010], which allow the developer to easily manipulate matrices of numerical data, and includes implementations of many standard signal processing techniques. The major downside to MATLAB is that it is not free and not open source, which is a considerable problem for researchers who want to share code and collaborate. GNU Octave [Eaton, 2002] is an open source alternative to MATLAB. It is an interpreted language with a syntax that is very similar to MATLAB, and it is possible to write scripts that will run on both systems. However, with both MATLAB and Octave this increase in short-term productivity comes at a cost. For anything other than very basic tasks, tools such as integrated development environments (IDEs), debuggers and profilers are certainly a useful resource if not a requirement. All of these tools exist in some form for MATLAB/Octave, but users must invest a considerable amount of time in learning to use a programming language and a set of development tools that have a relatively limited application domain when compared with general purpose programming languages. It is also generally more difficult to integrate MATLAB/Octave programs with compositional tools such as Csound [Vercoe et al., 2011] or Pure Data [Puckette, 1996], or with other technologies such as web frameworks, cloud computing platforms and mobile applications, all of which are becoming increasingly important in the music industry.

For developing and prototyping audio signal processing applications, it would therefore be advantageous to combine the power and flexibility of a widely adopted, open source, general purpose programming language with the quick development process that is possible when using interpreted languages that are focused on signal processing applications. Python [van Rossum and Drake, 2006], when used in conjunction with the extension modules NumPy [Oliphant, 2006], SciPy [Jones et al., 2001] and Matplotlib [Hunter, 2007] has all of these characteristics.
Section 2 provides a brief overview of the Python programming language. In Section 3 we discuss NumPy, SciPy and Matplotlib, which add a rich set of scientific computing functions to the Python language. Section 4 describes two libraries created by the authors that rely on SciPy, Section 5 shows how these Python programs can be integrated with other software tools for music composition, with final conclusions given in Section 6.

2 Python

Python is an open source programming language that runs on many platforms including Linux, Mac OS X and Windows. It is widely used and actively developed, has a vast array of code libraries and development tools, and integrates well with many other programming languages, frameworks and musical applications. Some notable features of the language include:

- It is a mature language and allows for programming in several different paradigms including imperative, object-orientated and functional styles.
- The clean syntax puts an emphasis on producing well structured and readable code. Python source code has often been compared to executable pseudocode.
- Python provides an interactive interpreter, which allows for rapid code development, prototyping and live experimentation.
- The ability to extend Python with modules written in C/C++ means that functionality can be quickly prototyped and then optimised later.
- Python can be embedded into existing applications.
- Documentation can be generated automatically from the comments and source code.
- Python bindings exist for cross-platform GUI toolkits such as Qt [Nokia, 2011].
- The large number of high-quality library modules means that you can quickly build sophisticated programs.

A complete guide to the language, including a comprehensive tutorial is available online at http://python.org.

3 Python for Scientific Computing

Section 3.1 provides an overview of three packages that are widely used for performing efficient numerical calculations and data visualisation using Python. Example programs that make use of these packages are given in Section 3.2.

3.1 NumPy, SciPy and Matplotlib

Python’s scientific computing prowess comes largely from the combination of three related extension modules: NumPy, SciPy and Matplotlib. NumPy [Oliphant, 2006] adds a homogenous, multidimensional array object to Python. It also provides functions that perform efficient calculations based on array data. NumPy is written in C, and can be extended easily via its own C-API. As many existing scientific computing libraries are written in Fortran, NumPy comes with a tool called f2py which can parse Fortran files and create a Python extension module that contains all the subroutines and functions in those files as callable Python methods.

SciPy builds on top of NumPy, providing modules that are dedicated to common issues in scientific computing, and so it can be compared to MATLAB toolboxes. The SciPy modules are written in a mixture of pure Python, C and Fortran, and are designed to operate efficiently on NumPy arrays. A complete list of SciPy modules is available online at http://docs.scipy.org, but examples include:

File input/output (scipy.io): Provides functions for reading and writing files in many different data formats, including .wav, .csv and matlab data files (.mat).

Fourier transforms (scipy.fftpack): Contains implementations of 1-D and 2-D fast Fourier transforms, as well as Hilbert and inverse Hilbert transforms.

Signal processing (scipy.signal): Provides implementations of many useful signal processing techniques, such as waveform generation, FIR and IIR filtering and multi-dimensional convolution.

Interpolation (scipy.interpolate): Consists of linear interpolation functions and cubic splines in several dimensions.

Matplotlib is a library of 2-dimensional plotting functions that provides the ability to quickly visualise data from NumPy arrays, and produce publication-ready figures in a variety of formats. It can be used interactively from the Python command prompt, providing similar functionality to MATLAB or GNU Plot [Williams et al., 2011]. It can also be used in Python scripts, web applications servers or in combination with several GUI toolkits.
3.2 SciPy Examples

Listing 1 shows how SciPy can be used to read in the samples from a flute recording stored in a file called *flute.wav*, and then plot them using Matplotlib. The call to the `read` function on line 5 returns a tuple containing the sampling rate of the audio file as the first entry and the audio samples as the second entry. The samples are stored in a variable called `audio`, with the first 1024 samples being plotted in line 8. In lines 10, 11 and 13 the axis labels and the plot title are set, and finally the plot is displayed in line 15. The image produced by Listing 1 is shown in Figure 1.

```python
from scipy.io.wavfile import read
import matplotlib.pyplot as plt

# read audio samples
input_data = read("flute.wav")
audio = input_data[1]

# plot the first 1024 samples
plt.plot(audio[0:1024])

# label the axes
plt.ylabel("Amplitude")
plt.xlabel("Time (samples)")

# set the title
plt.title("Flute Sample")

# display the plot
plt.show()
```

Listing 1: Plotting Audio Files

![Flute Sample](image)

Figure 1: Plot of audio samples, generated by the code given in Listing 1.

In Listing 2, SciPy is used to perform a Fast Fourier Transform (FFT) on a windowed frame of audio samples then plot the resulting magnitude spectrum. In line 11, the SciPy `hann` function is used to compute a 1024 point Hanning window, which is then applied to the first 1024 flute samples in line 12. The FFT is computed in line 14, with the complex coefficients converted into polar form and the magnitude values stored in the variable `mags`. The magnitude values are converted from a linear to a decibel scale in line 16, then normalised to have a maximum value of 0 dB in line 18. In lines 20-26 the magnitude values are plotted and displayed. The resulting image is shown in Figure 2.

```python
import scipy
from scipy.io.wavfile import read
from scipy.signal import hann
from scipy.fftpack import rfft
import matplotlib.pyplot as plt

# read audio samples
input_data = read("flute.wav")
audio = input_data[1]

# apply a Hanning window
window = hann(1024)
audio = audio[0:1024] * window

# fft
mags = abs(rfft(audio))

# convert to dB
mags = 20 * scipy.log10(mags)

# normalise to 0 dB max
mags -= max(mags)

# plot
plt.plot(mags)

# label the axes
plt.ylabel("Magnitude (dB)")
plt.xlabel("Frequency Bin")

# set the title
plt.title("Flute Spectrum")

plt.show()
```

Listing 2: Plotting a magnitude spectrum

4 Audio Signal Processing With Python

This section gives an overview of how SciPy is used in two software libraries that were created by the authors. Section 4.1 gives an overview of Simpl [Glover et al., 2009], while Section 4.2 introduces Modal, our new library for musical note onset detection.

4.1 Simpl

Simpl ¹ is an open source library for sinusoidal modelling [Amatriain et al., 2002] written in C/C++ and Python. The aim of this project is

¹Available at http://simplsound.sourceforge.net
to tie together many of the existing sinusoidal modelling implementations into a single unified system with a consistent API, as well as provide implementations of some recently published sinusoidal modelling algorithms. Simpl is primarily intended as a tool for other researchers in the field, allowing them to easily combine, compare and contrast many of the published analysis/synthesis algorithms.

Simpl breaks the sinusoidal modelling process down into three distinct steps: peak detection, partial tracking and sound synthesis. The supported sinusoidal modelling implementations have a Python module associated with every step which returns data in the same format, irrespective of its underlying implementation. This allows analysis/synthesis networks to be created in which the algorithm that is used for a particular step can be changed without affecting the rest of the network. Each object has a method for real-time interaction as well as non-real-time or batch mode processing, as long as these modes are supported by the underlying algorithm.

All audio in Simpl is stored in NumPy arrays. This means that SciPy functions can be used for basic tasks such as reading and writing audio files, as well as more complex procedures such as performing additional processing, analysis or visualisation of the data. Audio samples are passed into a PeakDetection object for analysis, with detected peaks being returned as NumPy arrays that are used to build a list of Peak objects. Peaks are then passed to PartialTracking objects, which return partials that can be transferred to Synthesis objects to create a NumPy array of synthesised audio samples. Simpl also includes a module with plotting functions that use Matplotlib to plot analysis data from the peak detection and partial tracking analysis phases.

An example Python program that uses Simpl is given in Listing 3. Lines 6-8 read in the first 4096 sample values of a recorded flute note. As the default hop size is 512 samples, this will produce 8 frames of analysis data. In line 10 a SndObjPeakDetection object is created, which detects sinusoidal peaks in each frame of audio using the algorithm from The SndObj Library [Lazzarini, 2001]. The maximum number of detected peaks per frame is limited to 20 in line 11, before the peaks are detected and returned in line 12. In line 15 a MQPartialTracking object is created, which links previously detected sinusoidal peaks together to form partials, using the McAulay-Quatieri algorithm [McAulay and Quatieri, 1986]. The maximum number of partials is limited to 20 in line 16 and the partials are detected and returned in line 17. Lines 18-25 plot the partials, set the figure title, label the axes and display the final plot as shown in Figure 3.

```python
1 import simpl
2 import matplotlib.pyplot as plt
3 from scipy.io.wavfile import read
4 # read audio samples
5 audio = read("flute.wav")
6 # take just the first few frames
7 audio = audio[0:4096]
8 # Peak detection with SndObj
9 pd = simpl.SndObjPeakDetection()
10 pd.max_peaks = 20
11 pks = pd.find_peaks(audio)
12 # Partial Tracking with
13 # the McAulay-Quatieri algorithm
14 pt = simpl.MQPartialTracking()
15 pt.max_partials = 20
16 partls = pt.findpartials(pks)
17 # plot the detected partials
18 simpl.plot.plotpartials(partls)
19 # set title and label axes
20 plt.title("Flute Partials")
21 plt.ylabel("Frequency (Hz)")
22 plt.xlabel("Frame Number")
23 plt.show()
```

Listing 3: A Simpl example
Figure 3: Partials detected in the first 8 frames of a flute sample, produced by the code in Listing 3. Darker colours indicate lower amplitude partials.

4.2 Modal
Modal \(^2\) is a new open source library for musical onset detection, written in C++ and Python and released under the terms of the GNU General Public License (GPL). Modal consists of two main components: a code library and a database of audio samples. The code library includes implementations of three widely used onset detection algorithms from the literature and four novel onset detection systems created by the authors. The onset detection systems can work in a real-time streaming situation as well as in non-real-time. For more information on onset detection in general, a good overview is given in Bello et al. (2005).

The sample database contains a collection of audio samples that have creative commons licensing allowing for free reuse and redistribution, together with hand-annotated onset locations for each sample. It also includes an application that allows for the labelling of onset locations in audio files, which can then be added to the database. To the best of our knowledge, this is the only freely distributable database of audio samples together with their onset locations that is currently available. The Sound Onset Labellizer [Leveau et al., 2004] is a similar reference collection, but was not available at the time of publication. The sample set used by the Sound Onset Labellizer also makes use of files from the RWC database [Goto et al., 2002], which although publicly available is not free and does not allow free redistribution.

Modal makes extensive use of SciPy, with NumPy arrays being used to contain audio samples and analysis data from multiple stages of the onset detection process including computed onset detection functions, peak picking thresholds and the detected onset locations, while Matplotlib is used to plot the analysis results. All of the onset detection algorithms were written in Python and make use of SciPy’s signal processing modules. The most computationally expensive part of the onset detection process is the calculation of the onset detection functions, so Modal also includes C++ implementations of all onset detection function modules. These are made into Python extension modules using SWIG [Beazley, 2003]. As SWIG extension modules can manipulate NumPy arrays, the C++ implementations can be seamlessly interchanged with their pure Python counterparts. This allows Python to be used in areas that it excels in such as rapid prototyping and in “glueing” related components together, while languages such as C and C++ can be used later in the development cycle to optimise specific modules if necessary.

Listing 4 gives an example that uses Modal, with the resulting plot shown in Figure 4. In line 12 an audio file consisting of a sequence of percussive notes is read in, with the sample values being converted to floating-point values between -1 and 1 in line 14. The onset detection process in Modal consists of two steps, creating a detection function from the source audio and then finding onsets, which are peaks in this detection function that are above a given threshold value. In line 16 a \texttt{ComplexODF} object is created, which calculates a detection function based on the complex domain phase and energy approach described by Bello et al. (2004). This detection function is computed and saved in line 17. Line 19 creates an \texttt{OnsetDetection} object which finds peaks in the detection function that are above a given median threshold [Brossier et al., 2004]. The onset locations are calculated and saved on lines 21-22. Lines 24-42 plot the results. The figure is divided into 2 subplots, the first (upper) plot shows the original audio file (dark grey) with the detected onset locations (vertical red dashed lines). The second (lower) plot shows the detection function (dark grey) and the adaptive threshold value (green).

\begin{verbatim}
from modal.onsetdetection
import OnsetDetection
from modal.detectionfunctions
\end{verbatim}

\(^2\)Available at http://github.com/johnglover/modal
5 Integration With Other Music Applications

This section provides examples of SciPy integration with two established tools for sound design and composition. Section 5.1 shows SciPy integration with The SndObj Library, with Section 5.2 providing an example of using SciPy in conjunction with Pure Data.

5.1 The SndObj Library

The most recent version of The SndObj Library comes with support for passing NumPy arrays to and from objects in the library, allowing data to be easily exchanged between SndObj and SciPy audio processing functions. An example of this is shown in Listing 5. An audio file is loaded in line 8, then the `scipy.signal` module is used to low-pass filter it in lines 10-15. The filter cutoff frequency is given as 0.02, with 1.0 being the Nyquist frequency. A SndObj called `obj` is created in line 21 that will hold frames of the output audio signal. In lines 24 and 25, a SndRTIO object is created and set to write the contents of `obj` to the default sound output. Finally in lines 29-33, each frame of audio is taken, copied into `obj` and then written to the output.
# use SciPy to low pass filter
order = 101
cutoff = 0.02
filter = firwin(order, cutoff)
audio = sp.convolve(audio,
                   filter,
                   "same")
# convert to 32-bit floats
audio = sp.asarray(audio,
                   sp.float32)
# create a SndObj that will hold
# frames of output audio
obj = SndObj()
# create a SndObj that will
# output to the sound card
outp = SndRTIO(1, SND_OUTPUT)
outp.SetOutput(1, obj)
# get the default frame size
f_size = outp.GetVectorSize()
# process each frame
i = 0
while i < len(audio):
    f = audio[i:i+f_size]
    p = m.process(f)
    p = sp.fromstring(p, int16)
    out = sp.hstack((out, p))
    i += f_size
# close the patch
pd.libpd_close_patch(patch)
# write the audio file to disk
write("out.wav", 44100, out)

Listing 5: The SndObj Library and SciPy

5.2 Pure Data
The recently released libpd \(^3\) allows Pure Data to be embedded as a DSP library, and comes with a SWIG wrapper enabling it to be loaded as a Python extension module. Listing 6 shows how SciPy can be used in conjunction with libpd to process an audio file and save the result to disk. In lines 7-13 a PdManager object is created, that initialises libpd to work with a single channel of audio at a sampling rate of 44.1 KHz. A Pure Data patch is opened in lines 14-16, followed by an audio file being loaded in line 20. In lines 22-29, successive audio frames are processed using the signal chain from the Pure Data patch, with the resulting data converted into an array of integer values and appended to the out array. Finally, the patch is closed in line 31 and the processed audio is written to disk in line 33.

```python
import scipy as sp
from scipy import int16
from scipy.io.wavfile import read, write
import pylibpd as pd
num_chans = 1
```

6 Conclusions
This paper highlighted just a few of the many features that make Python an excellent choice for developing audio signal processing applications. A clean, readable syntax combined with an extensive collection of libraries and an unrestricted open source license make Python particularly well suited to rapid prototyping and make it an invaluable tool for audio researchers. This was exemplified in the discussion of two open source signal processing libraries created by the authors that both make use of Python and SciPy: Simpl and Modal. Python is easy to extend and integrates well with other programming languages and environments, as demonstrated by the ability to use Python and SciPy in conjunction with established tools for audio signal processing such as The SndObj Library and Pure Data.

7 Acknowledgements
The authors would like to acknowledge the generous support of An Foras Feasa, who funded this research.

\(^3\)Available at http://gitorious.org/pdlib/libpd
References


Abstract
Graphical sequencers have limits in their use as live performance tools. It is hypothesized that those limits can be overcome through live coding or text-based interfaces. Using a general purpose programming language has advantages over that of a domain-specific language. However, a barrier for a musician wanting to use a general purpose language for computer music has been the lack of high-level music-specific abstractions designed for real-time manipulation, such as those for time. A library for Haskell was developed to give computer musicians a high-level interface for a heterogeneous output environment.

Keywords
live coding, realtime performance, Haskell, text-based interface

1 Introduction
In this paper, a usability problem of live computer music will be briefly examined, and the solution of using a general purpose programming language as a shell for music will be presented. The necessary components created by other developers which were used will be introduced. A library called Conductive\(^1\), developed to make a Haskell interpreter into such a shell, will then be described in some detail. A conceptual example and some actual code examples will be presented. Finally, conclusions reached through the development of this library will be presented.

2 The Problem
Graphical sequencers are poor tools for live musical performance in the judgement of this author. Users interact with them primarily through a mouse or a limited number of keyboard shortcuts and allow limited customizations to the manner in which they are controlled. Previous experiences with GUI tools in performance showed them to be inflexible and awkward when trying to execute complex sets of parameter changes simultaneously.

This problem is exacerbated when considering the wide variety of synths which exist. Musicians would like to use them together freely, but coordinating them is difficult. For use with graphical sequencers, synths employ separate GUIs which are almost always point-and-click and thus cannot be easily manipulated simultaneously with other parameters.

One possible solution to this problem may be the use of a programming language as a tool for live coding of music or as a text-based interface [Collins et al., 2003]. A general purpose programming language which has abstractions for music can serve as a control center or shell for a heterogeneous set of outputs. In text, the user can write out complex parameter changes and then execute them simultaneously. A wide variety of existing programming tools and text-manipulation utilities can be used to make this process more efficient.

Computer music languages, such as SuperCollider [McCartney, 2010] and Chuck [Wang, 2004] exist. McCartney, the creator of SuperCollider, says a specialized language for computer music isn’t necessary, but general purpose programming languages aren’t ready yet practically [Mccartney, 2002]. Musicians manage to make music with domain-specific tools, but those are unsatisfactory in many ways, such as lack of libraries and development tools and slow performance.

General purpose programming languages have libraries for dealing with music, but they are particularly limited in number regarding real-time music systems. Some which already have such capabilities through libraries include Scheme through Impromptu [Sorensen and Brown, 2007] or Lua through LuaAV [Wakefield and Smith, 2007].

The Haskell programming language was seen as a good candidate because of several factors:

\(^1\)http://www.renickbell.net/conductive/
expressivity, speed, static type system, large number of libraries, and ability to be either interpreted or compiled. It lacked a library suitable for this author for realtime manipulation of musical processes. McClean is also developing Tidal [McLean and Wiggins, 2010], a Haskell library with a similar aim.

3 The Solution

A Haskell library called Conductive was created. It contains abstractions for musical time, musical events, and event loops. This gives the Haskell interpreter the ability to function as a code-based realtime sequencer for any output targets which can be connected to the system. Conductive does not aim to be an audio language, but a controller for audio output targets. The user is free to choose any OSC-aware output target, and this library is proposed as a way to coordinate those outputs. Another way to think of it is as a shell or scripting environment for realtime music.

A library for getting, composing, and sending messages to JackMiniMix, an OSC-based mixer for JACK developed by Nicholas Humfrey [Humfrey, 2005], was created.

A simple terminal-based clock visualization was also created.

4 Utilized Tools from Other Developers

Before explaining the details of Conductive, it is necessary to list the components it was integrated with. The Glasgow Haskell Compiler Interpreter (ghci) [Peyton Jones et al., 1992] was the core component used for executing Haskell code. Code was composed in vim [Moolenaar, 2011], and sent to ghc via the tslime plugin [Coutinho, 2010]. For OSC communication, Rohan Drape’s hose library was used [Drape, 2010]. Output targets used were scsynth, the synthesizer component of SuperCollider [McCartney, 2010], and JackMiniMix. Drape provides a library for communicating with scsynth via OSC called hsc3 [Drape, 2009].

5 Conductive in Detail

5.1 Overview

This library exists to wrap concurrent process manipulation in a way that makes controlling their timing more intuitive for musicians. At the same time, the library aims at being as concise as possible to lessen the burden on the user.

---

2http://www.renickbell.net/doku.php?id=jackminimix
The core components of the library are the data structures Player and MusicalEnvironment and a set of functions using these data structures. A user designs a set of functions carrying out musical actions, such as playing a note on a synth or adjusting the parameter of synth. The user defines TempoClocks which have a tempo and time signature. The user also defines functions, called IOI (interonset interval) functions, describing how long to wait between executions of actions. These functions are stored in a MusicalEnvironment. A Player is a collection of one action function, one IOI function, and one TempoClock and other related data. A Player is put into an event loop in which actions are executed after every defined IOI by using the play function.

Conceptually, it has similarities with the concepts in SuperCollider of Routines, Tasks, and Patterns. Some key similarities and differences are noted below, along with details on each of these components.

5.2 TempoClock
The tempo is part of a TempoClock, a concept from SuperCollider which is reimplemented here in Haskell. A TempoClock is like a metronome keeping the current tempo but also containing information about time signature and when tempo or time signature has been changed.

A TempoClock is a record of the time the clock was started, a list of TempoChanges, and a list of TimeSignature changes. This allows a user to independently manipulate both tempo and time signature and to use these for composing and performance in addition to regular POSIX time.

TempoClocks are stored in the MusicalEnvironment.

5.3 Players
A data structure called a Player was designed as a way to sequence IO actions. Players contain references to particular data which is stored in the MusicalEnvironment. The collection of data referenced by the Player results in a series of actions being produced once the Player is played. This data consists of:

- the name of the Player
- its status (stopped, playing, pausing, paused, stopping, resetting)
- a counter of how many times it has run an action

An action function is a function that describes an event. An action function outputs a value of the IO unit type. This basically means some kind of side effect is produced without returning a value like a double or a string. In practical terms, this could be a synthesis event, a parameter change, or the action of playing, pausing, or stopping other Players or itself. It is thought that the user would use functions which send OSC messages to connected OSC-aware applications. The action named in the Player can only take two parameters: the Player triggering the action and the MusicalEnvironment it should read from. Beyond that, the design of the action is left to the user. A user might prefer to have many Players with simple actions, a few Players with complex actions, or some other combination.

A fundamental concept is that of the time interval between the start times of two events, or interonset interval (IOI). [Parncutt, 1994] SuperCollider refers to this as “delta” with regard to Patterns or “wait” for Routines. The IOI is defined in beats, and the actual time between events is calculated using the IOI value and the
TempoClock referenced by the Player it is associated with. IOI functions should also be designed to read the data from a Player and a MusicalEnvironment. They can be designed in any way the user desires, including always returning a particular value, stepping through a list of values stored in a list somewhere, randomly choosing a value, or anything else the composer can imagine.

An interrupt function is a function which is run once every time the play loop runs. It is useful for debugging purposes, and may be used to trigger other actions, such as stopping the player on a condition.

Players bear some resemblance to Tasks or Patterns in SuperCollider; they can be played, paused, and stopped to produce music. However, while Patterns in sclang can produce streams of any kind of data, Players in Conductive are designed to produce streams of side effects. While the data in a Pbind in SuperCollider is generally private [Harkins, 2009], all the data contained by a Player is visible.

Players are stored in the Player store, a mutable key-value store where the keys are Player name strings and the values are the Players themselves. This in turn is part of the MusicalEnvironment. How patterns are stored in SuperCollider is up to the individual user. This library provides a readymade structure for that purpose.

5.4 MusicalEnvironment

The MusicalEnvironment is a place to store data which is used by any of the user-initiated event loops. This data consists of:

- the name of the environment
- a store of Players
- a store of TempoClocks
- a store of IOI functions
- a store of action functions
- a store of interrupt functions

5.5 Play

The play function starts a thread which forks other processes according to a schedule determined by the IOI function referenced in the Player. It takes a MusicalEnvironment, a Player store, and a Player name as arguments. First, the play function checks which action is referenced in the Player. It retrieves that function from the MusicalEnvironment and forks it to a thread. It then checks which IOI function is referenced in the Player. It runs that function and receives a numeric value specifying how long to wait in terms of beats. It then corrects that amount for jitter and sleeps for the corrected length of time. When the thread wakes up, the loop — checking the action and so on — repeats.

It produces roughly similar results to calling play on a Pattern in SuperCollider in that it begins a process; however it is structured differently.

The problem of dealing with the delays in scheduled events is significant. Because various processes, including garbage collection, can conceivably interfere with correct timing, correction of jitter is included in the play event loop. This library does not introduce a novel method for managing such delay, but rather adopts a design from McLean [McLean, 2004]. An event intended to occur at time \( x \) actually occurs at time \( x + y \), where \( y \) is the amount of time by
which the event is late. The next event is scheduled to occur at time \( x + z \), where \( z \) is the IOI, so to account for the jitter, the wait time is set for \( x + (z-y) \). In practice, this delay is generally permissible for control data, while it would not be appropriate for audio data.

The number of simultaneous play event loops is limited only by the memory and CPU of the host machine. Since at every loop the data used is refreshed, they can be manipulated in real time by changing the data stored in the Player or MusicalEnvironment. Which action function or IOI function is referenced in a Player can be changed. The action functions or IOI functions themselves can be modified. Any of the other data in the Players or MusicalEnvironment can be changed. By changing this data, the resulting musical output can be changed. It is in this manner that a livecoded musical performance is realized.

Such manipulation results in many threads and the potential exists for one thread to be writing data which is accessed by another. One problem of real-time multi-threaded systems is guaranteeing the thread safety of data. Haskell provides safe concurrency in the standard libraries of the Glasgow Haskell Compiler (GHC).

### 5.6 An Example of How Players Work

Here is an example of how Players work, shown in figure 6.

Consider a timpani Player called “A” who has only two jobs. The first job is to hit the timpani. The second job is to wait for a given amount of time, like that written on a score. He hits the timpani, then he waits, then he hits the timpani again and waits, in a loop until he is asked to stop. Now imagine that this Player is joined by another: Player “B”. The second Player has only two jobs. The first is to adjust the tuning of the timpani; the second job is the same as that of the first Player. He tunes the timpani and waits, and then tunes it again and waits, repeating like the first Player.

The first timpani Player is a Player stored under the key “A” in the Player store. Its action function is “hit the timpani”, which may correspond to triggering a synthdef on scserver called “timpani”, which results in a timpani sound being played. The second Player is called “B”, and its action function, “tune timpani”, is to change the frequency parameter used by the “hit the timpani” function. Each of them has its own IOI function.

Let’s expand the situation to include two more Players, Players “C” and “D”, who correspond to Players “A” and “B” but are involved with another timpani. The resulting sound is two timpanis being played at the same time. In this case, the “hit the timpani” action is designed to use the name of the Player to determine which frequency should be used. In the same way, the “tune timpani” function uses the Player name to determine which frequency it is tuning and which frequency to tune to.

Now, interestingly, we’ll add a fifth Player, who is starting and stopping the Players above. Its action function cycles through a list of actions. Its first action is to start Player “A”. Its second action is to start Player “B”. Its third action could be to start Players “C” and “D” simultaneously. Its fourth action could be to pause Players “A”, “B”, and “D”. The design of any action is up to the intentions of the musician.

![Figure 6: An example of Players at work](image)

### 5.7 Code Examples of Conductive Usage

A rudimentary sample of usage and corresponding code is given below.

First, the relevant Haskell modules must be
imported, which is accomplished by loading a Haskell document containing the necessary import statements.

:load Conductive.hs

This example uses SuperCollider, so a convenience command which sets up a group on scserver is called.

defaultSCGroup

A default MusicalEnvironment is instantiated. It is assigned to the variable "e".

e <- defaultMusicalEnvironment

An scserver-based sampler is instantiated using this command, which also creates the necessary Players and action functions in the MusicalEnvironment. The function takes a path and the MusicalEnvironment as arguments.

s <- initializeSampler "../sounds/*" e

All of the Players in a specified MusicalEnvironment can be started with the playAll function. The argument, like above, is the MusicalEnvironment.

playAll e

The status of all the players in a specified MusicalEnvironment can be viewed with the displayPlayers command.

displayPlayers e

A list of players can be paused using the pauseN function. The specified players will be looked up in the MusicalEnvironment.

pauseN e "sampler1", "sampler2"

Those players could be restarted at a specified time, in this case the first beat of the 16th measure, using the playNAt function. The string after "e" is the specified time, given in terms of measure and beat.

playNAt e "15.0" "sampler1", "sampler2"

The tempo of a particular TempoClock can be changed with the changeTempo function. The string "default" is the name of the TempoClock that is to be manipulated.

changeTempo e "default" 130

A new IOI function can be created. This function call gives the name "newIOI" to an IOI function which will be stored in the MusicalEnvironment. That string is followed by the offset, the number of beats before the first event takes place. The list contains IOI values; in this case, an interval of three beats passes between the first two events.

newIOIFunctionAndIOIList e "newIOI"

  0 [3,0.25,1,0.5,2,0.25,3]

A player can be told to use this new IOI function by calling the swapIOI function. After specifying the MusicalEnvironment, the name of the player and the name of the IOI function are given.

swapIOI e "sampler2" "newIOIPattern"

All of the players can be stopped with the stopAll function.

stopAll e

6 Conclusion and Future Directions

Rudimentary livecoding performances were made possible. The timing was found to be adequate for musical performances, though millisecond timing errors remain. While the library was sufficient for very basic performances, it was necessary to create additional libraries for control and algorithmic composition to achieve a usable interface and more sophisticated performances.

This library alone was far from sufficient to replace current GUI sequencers for most users, though it is hoped this is a good foundation for further research in this direction.

An evaluation method to quantify the usability of this approach should be considered. Additionally, determining the performance of this system versus sclang, Impromptu and others may be valuable.

The library will be tested in performance situations and expanded to be a more complete integrated development environment and performance tool for livecoding performers. Its use in other real-time music applications will also be tested.

The jitter described above is believed to be at least in part due to the garbage collection routines of GHC. Improvements to the GHC
garbage collector are currently being made by
its developers. [Marlow, 2010] It is hoped that
the gains they make will carry over positively to
the performance of this system in terms of re-
duced delays. There could be other contributing
factors, but they have not yet been identified. A
deeper investigation into potential causes of jiter
and their solutions needs to be undertaken.

Another serious problem involves tempo
changes. If the tempo is changed while a play
process is sleeping, the next event in that pro-
cess will be out of sync: early if the tempo is
reduced, or late if the tempo is increased. Fol-
lowing events, however, will occur at the correct
times. This is because the function for awak-
ening the sleeping Player is unaware of tempo
changes and thus cannot adjust the time accord-
ingly. A revised method for sleeping threads
which is tempo-aware should be developed.

An important next step is developing a li-
brary to make it easy to use MIDI devices with
Conductive.

Use of this library by visually-impaired users
should be examined, as this text interface may
offer such users increased usability. It will be
necessary to find users with a braille display and
familiarity with vim or emacs for usability test-
ing.

7 Acknowledgements

Thanks are due to Masaaki Tsuji, who pro-
vided a great deal of valuable discussion during
the development of this library, Naotoshi Os-
aka for feedback and pushing me ahead, and
Michael Chinen, who provided a helpful evalu-
ation. Thanks also to Marvin Wardi for assis-
tance in proofreading.

References

Nick Collins, Alex McLean, Julian Rohrhu-
ber, and Adrian Ward. 2003. Live coder
ing in laptop performance. Organised Sound,

C. Coutinho. 2010. tsime.vim.
php?script\_id=3023, March.

Rohan Drape. 2009. Haskell supercollider, a
sw/hsc3/.

hackage.haskell.org/package/hosc-0.8,
March.

Patterns. http://www.dewdrop-world.net/
sc3/sym09/.

Nicholas J. Humphrey. 2005. JackMin-
jackminimix/, June.

Simon Marlow. 2010. First results
from GHC’s new garbage collector.
http://hackage.haskell.org/trac/
ghc/blog/new-gc-preview, September.

James Mccartney. 2002. Rethinking the Com-
puter Music Language: SuperCollider. Com-

J. McCartney. 2010. SuperCollider Documen-

Alex McLean and Geraint Wiggins. 2010.
Tidal - Pattern Language for the Live Coding
of Music. In Proceedings of the 7th Sound and
Music Computing conference.

Alex McLean. 2004. Hacking Perl in night-
08/31/livecode.html.

vim.org/.

Richard Parncutt. 1994. A Perceptual Model
of Pulse Salience and Metrical Accent in Mu-
sical Rhythms. Music Perception: An Inter-
disciplinary Journal, 11(4).

Simon L. Peyton Jones, Cordy Hall, Kevin
Hammond, Jones Cordy, Hall Kevin, Will
Partain, and Phil Wald. 1992. The Glas-
gow Haskell compiler: a technical overview.

A. Sorensen and A. R. Brown. 2007. aa-cell in
Practice: An approach to musical live coding.
In Proceedings of the International Computer
Music Conference.

Graham Wakefield and Wesley Smith. 2007.
Using Lua for Audiovisual Composition. In
International Computer Music Conference.

Using Code as an Expressive Musical Instru-
ment. In In Proceedings of the International
Conference on New Interfaces for Musical Ex-
pression, pages 138–143.
FluidSynth real-time and thread safety challenges

David Henningsson
FluidSynth Developer Team
diwic@ubuntu.com

Abstract
FluidSynth takes soundfonts and MIDI data as input, and gives rendered audio samples as output. On the surface, this might sound simple, but doing it with hard real-time guarantees, perfect timing, and in a thread safe way, is difficult.
This paper discusses the different approaches that have been used in FluidSynth to solve that problem both in the past, present, as well as suggestions for the future.

Keywords
FluidSynth, real-time, thread safety, soundfont, MIDI.

1 Introduction to FluidSynth
FluidSynth is one of the more common software synthesizers in Linux today. It features a high level of compliance to the SoundFont (SF2) standard, as well as good performance. The design is modular enough to suit a variety of use cases.
FluidSynth does not come bundled with a GUI, but several front-ends exist. It does however come with several drivers for MIDI and audio, e.g. JACK, ALSA, PulseAudio, OSS, CoreAudio/CoreMidi (MacOSX), and DirectSound (Windows).

1.1 FluidSynth’s use cases
FluidSynth is not only a command-line application, but also a library used by more than 15 other applications [1], all putting their requirements on the FluidSynth engine. Requirements include:
• low-latency guarantees, e.g. when playing live on a keyboard.
• fast rendering¹, e.g. when rendering a MIDI file to disk.
• configurability, such as loading and changing soundfonts on the fly.
• monitoring current state and what’s currently happening inside the engine, needed by GUI front-ends and soundfont editors.

1.2 Introduction to SoundFonts
SoundFont (SF2) files contains samples and instructions for how to play them, just like similar formats such as DLS, Gigastudio and Akai. A soundfont renderer must implement features such as cut-off and resonance filters, ADSR envelopes, LFOs (Low-Frequency Oscillators), reverb and chorus, and a flexible system for how MIDI

¹ Sometimes known as “batch processing”, a mode of operation where throughput matters more than latency.
messages affect the different parameters of these features.

1.3 More background information

1.3.1 Buffer management

FluidSynth internally processes data in blocks of 64 samples\(^2\). It is between these blocks the rendering engine can recalculate parameters, such as e.g. current LFO values and how they affect pitch, volume, etc.

There is also the concept of the audio buffer size, which controls the latency: the audio driver uses this size parameter to determine how often the system should wake up, executing one or more internal block rendering cycles, and write the result to the sound card's buffer.

1.3.2 MIDI processing latency

To understand some of the problems faced below, it is also important to understand the difficulty of handling all MIDI messages in a timely fashion:

- Loading soundfonts or MIDI files from disk are worst, and are not guaranteed to execute within an acceptable amount of time due to disk accesses.
- MIDI Program change messages are troublesome, somewhat depending on the current API allowing custom soundfont and preset loaders.
- Other MIDI messages, while they are not calling into other libraries (and thus unknown code latency-wise), still take some time to process, compared to just rendering a block.

\(^2\) It is known as the FLUID_BUFSIZE constant in the code, and I have never seen anyone change it.

2 Architecture before 1.1.0

FluidSynth has always had a multi-threaded architecture: One or more MIDI threads produce MIDI input to the synthesizer, and the audio driver thread is asking for more samples. Other threads would set and get the current gain, or load new soundfonts.

2.1 Thread safety versus low latency

When the author got involved with the FluidSynth project, a few years ago, thread safety was not being actively maintained, or at least not documented properly. There weren’t any clear directions for users of FluidSynth’s API on what could be done in parallel.

Yet there seems to have been some kind of balance: Unless you stress tested it, it wouldn’t crash that often – even though several race conditions could be found by looking at the source code. At the same time, latency performance was acceptable – again, unless you stress tested it, it wouldn’t underrun that often.

This “balance” was likely caused by carefully selecting places for locking a mutex – the more MIDI messages and API calls protected by this mutex, the better thread safety, but worse latency performance. In several places in the code, one could see this mutex locking code commented out.

2.2 The “drunk drummer” problem

An additional problem was the timing source: The only timing source was the system timer, i.e. timing based on the computer's internal clock. This had two consequences.

The first: All rendering, even rendering to disk, took as long time as the playing time of the song, so if a MIDI file was three minutes long, rendering

![FluidSynth Thread Diagram](https://via.placeholder.com/150)

Audio driver thread: Render blocks
Shell thread: load new SF2 file
MIDI thread: input from keyboard
GUI thread: Set reverb width

Different threads calling into FluidSynth
that song would take three minutes, with the computer idling most of the time.

The second: With larger audio buffer/block sizes\(^3\), timing got increasingly worse. Since audio was rendered one audio buffer at a time, MIDI messages could only be inserted between these buffer blocks. All notes and other MIDI events therefore became quantized to the audio block size. (Note that this quantization is not at all related to the intended timing of the music!)

This problem was labelled “the drunk drummer problem”, since listeners were especially sensitive to the drum track having bad timing (even though the same bad timing was applied to all channels).

3 Architecture in 1.1.0 and 1.1.1

3.1 Queuing input

To make FluidSynth thread safe, it was decided to queue MIDI messages as well as those API calls setting parameters in the engine. This was implemented as lock-free queues – the MIDI thread would insert the message into the queue, and the audio thread would be responsible for processing all pending MIDI messages before rendering the next block.

3.2 The sample timer

To make the drunk drummer sober again, the “sample timer” was added – that uses the number of rendered samples as a timing source instead of the system timer. This also allowed features such as fast MIDI-file rendering to be added. This was implemented so that on every 64th sample, a callback was made to the MIDI player so that it could process new MIDI messages.

3.3 Problems with the overhaul

3.3.1 Worse latency

As the audio thread was now expected to process all MIDI messages, this meant more pressure on the MIDI messages to return timely, and audio latency now had to take MIDI processing into account as well. The sample timer made this even worse, as all MIDI file loading and parsing now also happened in the audio thread.

3.3.2 Reordering issues

To aid the now tougher task of the audio thread, program change messages were still processed in the MIDI thread, queueing the loaded preset instead of the MIDI message. However, this also meant that bank messages had to be processed immediately, or the program change would load the wrong preset. In combination with API calls for loading soundfonts, this became tricky and there always seemed to be some combination order not being handled correctly.

3.3.3 Not getting out what you're putting in

Since API calls were now being queued until the next rendering, this broke API users expecting to be able to read back what they just wrote. E.g if a GUI front-end set the gain, and then read it back, it would not read the previous set value as that value had not yet been processed by the audio thread.

This was somewhat worked around by providing a separate set of variables that were updated immediately, but since these variables could be simultaneously written by several threads, writes and reads had to be atomic, which became difficult

\[^3\] In high-latency scenarios, such as a MIDI file player, you would typically want as large buffer as possible, both to avoid underruns and to improve overall performance.
when writes and reads spanned several variables internally.

4 Architecture in 1.1.2 and later

To overcome the problems introduced with 1.1.0, the thread safety architecture was once again rewritten in 1.1.2. This time, it was decided to split the engine into two parts: one for handling MIDI and one for handling audio. Hard real-time is guaranteed for the audio thread only, in order not to miss a deadline and cause underruns as a result.

For MIDI, the synth no longer has an input queue, but is instead mutex protected\(^4\). This means, that if one thread calls into the API to do something time intensive, such as loading a new soundfont, other MIDI threads will be delayed in the meantime and will have to wait until soundfont loading is finished.

4.1 The new queue

Instead of having a MIDI input queue, the queue has now moved to being between the MIDI handling and the audio thread. Instead of queuing the MIDI messages themselves, the outcome of the MIDI processing is queued to the audio thread. This releases pressure on the audio thread to handle MIDI processing, so audio latency is improved. If MIDI processing is lengthy, the end result will be that the result of that event is delayed – as compared to 1.1.0, where the result would have been an underrun.

\(^4\) The mutex can optionally be turned off for cases where the API user can guarantee serialized calls into the API, e.g. in some embedded use cases.

4.2 Return information

A queue with return information also had to be added, with information flowing from the Audio rendering thread to the MIDI threads. This is used to notify the MIDI processing when a voice has finished, so that the voice can be reallocated at the next MIDI note-on event. This return information queue is processed right after the mutex is locked.

5 Conclusion and suggestions for the future

While the architecture in 1.1.2 seems to have been more successful than the previous attempts in terms of stability, it is still not optimal. There is still work that could be done to improve the thread safety and real-time performance further.

5.1 Sample timers

Given the new architecture, the sample timer mechanism needs to be rewritten to work optimal under low latency conditions: as it currently stands, the audio thread triggers the sample timer, which in turn performs potentially lengthy MIDI processing.

To solve this problem without regressing back to the “drunk drummer”, one would need to add a “mini sequencer” into the event queue so that events can be added to be processed by the audio thread not before all 64 sample blocks are processed, but also between them. This would further require a time aware MIDI part of the synth – so the MIDI part would know where into insert the new queue item. Also the MIDI player needs to have a separate thread, monitoring the progress of the audio stream and adding more MIDI events as necessary.

---

\( \text{Threading architecture in 1.1.2} \)
5.2 Synchronous MIDI/Audio

In particular, synchronous MIDI and audio can be a problem when using JACK MIDI in conjunction with JACK audio – because JACK calls into both MIDI and audio callbacks synchronously. To try to avoid MIDI blocking audio rendering, MIDI input could be queued to a lower priority thread, and processed as time permits. Caution must be taken to make that this does not happen when JACK is running in its “freewheeling” mode, where MIDI and audio callbacks should be processed in the exact order they arrive.

5.3 Monitoring the audio thread

A sometimes asked for feature, in particular by soundfont editors and sometimes by other GUI frontends, is to be able to monitor the progress of the audio rendering.

This could be e.g. to see the current sample position of a particular voice, or to be able to receive callbacks whenever something happens in the audio thread, e.g. when a voice enters a new envelope stage. This is currently difficult as information is optimized to flow from the MIDI part to the audio thread, not the other way around.

One solution to this problem would be for the audio thread to continuously write down relevant information into a buffer, that the MIDI part could read upon request. Caution must be taken in order not to have the MIDI part read partially updated information (and thus get a potentially inconsequent view), but at the same time an ongoing read should not block a new write. This can be done with some clever atomic pointer exchanges.

The audio thread's write-down operations could potentially be turned on and off, in order not to hurt performance for users not needing the feature.

6 Acknowledgements

Thanks go to the rest of FluidSynth developer team (Joshua “Element” Green and Pedro Lopez-Cabanillas), as well as all other contributors to FluidSynth, including patch writers, testers, and people helping out on the FluidSynth development mailing list.

References

Abstract

Collaboration and education in the world of digital audio synthesis has been facilitated in many ways by the internet and its World Wide Web. Numerous projects have endeavoured to progress beyond text-based applications to facilitate wider modes of collaborative activity such as, network musical performance and composition. When developing a software application one is faced with technology decisions which limit the scope for future development. Similarly, the choice of one audio synthesis language or engine over another can limit the potential user base of the application. This paper will describe how, through the WADE system architecture it is possible to integrate existing software in a highly scalable and deployable way. The current incarnation of the WADE system, as depicted in the WADE Examples section below, uses the Csound synthesis engine and audio synthesis language as its audio server.

Keywords

Audio-synthesis, collaboration, education, web-based, OSGi

1 Introduction

The Web-enabled Audio-synthesis Development Environment, WADE, is a proof of concept application designed to facilitate collaboration and education in audio synthesis development. While the initial goal of this project was to investigate the creation of a browser based user interface for the Csound[1] synthesis engine, research and technology decisions lead to the outlining of a possible software architecture which was not tied to any specific synthesis engine, or even to audio synthesis development itself. The application was developed using the Eclipse Rich Client Platform[2], Equinox OSGi[3], Java Servlets, HTML[4] and JavaScript[5].

1.1 Web-enabled Audio Synthesis

In 1995 Ossenbruggen and Elıens proposed the use of client-side sound synthesis techniques in order to reduce the amount of resources needed for high quality music on the web[6]. Their project wrapped Csound for use with a Tcl/TK (scripting language and its graphical widget library) based browser or Tcl browser plugin. This allowed Csound to be used as a synthesis engine on the client side, thus removing the amount of data being transferred from client to server and moving the more processor intensive task, of synthesising a Csound Score file, to the client.

This work was closely followed by a network enabled sound synthesis system created by James McCartney, (1996) in the guise of his Supercollider synthesis project. More recent work done on web enabling existing sound synthesis engines has been carried out by Jim Hearon[7] and Victor Lazzarini[8] (independently), using Csound and Alonso, et al. in their creation of a Pure Data browser plug-in[9]. The need for web-enabled audio synthesis as a pedagogical tool or as a means of offering high quality audio across low bandwidth networks can now be answered in new and interesting ways which can lead to unforeseen future development projects.

1.2 Open Service Gateway Initiative (OSGi)

The OSGi component architecture is a framework which sits on top of the Java virtual machine. It is specifically designed to deal with many of the pitfalls of OOP and the way in which software applications have historically been constructed.
At the heart of the OSGi ideology is the bundle; a bundle provides a specific piece of functionality and advertises this functionality as a service to other bundles through a well-defined interface. As well as specifying the services they provide, bundles must specify their dependencies on other services/bundles. This allows the OSGi service registry to determine whether a particular bundle can be started. In the same vein, a bundle can specify what functionality it can provide in circumstances when only some of its dependencies are satisfied. It is also possible for bundles to provide extension points, points at which one bundle can extend or override the functionality of another bundle.

This tolerance for dynamic service availability makes OSGi well suited for developing applications seeking to utilise web services and diverse hardware. In this specific project one of the underlying requirements is the reuse of existing software applications i.e. Csound and the Jetty http server, the ability to register these as separate services within the OSGi framework means that they can be implemented independently of the overall system, e.g. if future implementations wish to use a different audio synthesis engine they would simply have to provide an OSGi bundle for that synthesis engine, its complementary web-interface bundle, and choose to use that engine over any pre-existing one.

1.3 Eclipse RCP

The Eclipse Rich Client Platform is a collection of OSGi bundles called a target platform. The RCP specific bundles allow developers to make contributions to an Eclipse GUI through extension points, advertised by the Eclipse runtime’s extension registry.

1.4 HTML + JavaScript

Hyper Text Mark-up Language is the language used by web developers and web browsers to layout web-page content. While book publishing houses have been using mark-up annotations for many years, they do not have to contend with dynamic content changes such as those seen in web pages. JavaScript or ECMAScript is a scripting language supported by most web browsers that allows web developers to create more dynamic and interactive web pages. This project extends the CodeMirror JavaScript code editor[10] to enable parsing and syntax highlighting of Csound CSD files within the web browser. The JQuery [11] library and JQueryUI JavaScript libraries were used to create the web page user interface.

1.5 Java Servlets

Java Servlets are Java classes which can be used to create and extend web based applications. They have access to a range of http calls and also the Java APIs. In this particular application they are used to provide the web facing interface for the Csound synthesis engine.

2 WADE Architecture

In explanation of the fig. 2 above; the program runs within the JVM, on top of this runs the OSGi framework which manages the bundle configuration that provides functionality for both the frontend and the backend of the system. As all bundles are registered with the OSGi service registry it is possible for any bundle to request the
functionality of another bundle. As such, the frontend and backend sections of the diagram are simply logical compartmentalisations for the purposes of design thinking.

The frontend contains the specific bundles required to manage the GUI features of the application and the backend contains the functionality desired by the project, e.g. the synthesis engine, its servlet interface and the servlet container which will serve the servlets when requested by the web browser. A benefit of this configuration is that the applications functionality can be extended independently of the user interface.

OSGi enables dynamic service composition through a number of mechanisms. One of these is Declarative Services, which can be seen in the PlayServlet class, where there is no Equinox specific code required. The component.xml file and the ServletComponent class are used by the Equinox DS bundle to weave the services together at runtime.

```java
public class PlayServlet extends HttpServlet{

    The HTTP POST method needs to be implemented in order to accept the incoming Csound file from the web editor.

    public void doPost(
        HttpServletRequest req,
        HttpServletResponse resp)
    {
        String csdString = (String) req.getParameter("csdElement");
        try {
            if((csnd!=null) && (!csdString.equalsIgnoreCase(""))){
                csnd.setCSDString(csdString);
                csFile.exportForPerformance();
                csFile.setCommand(csOptionsString);
                csndSingleThread.start();
                csoundObj= new CppSound();
                chanCaller = new CsChansetCB(csoundObj);
                chanCaller.setOutputValueCallback();
                chanCaller.setMessageCallback();
                csndSingleThread = new Thread(this);
                csndSingleThread.start();
                csFile.getCsoundFile().setCSD(csdString);
                csFile.setCommand(csOptionsString);
                csFile.setCommand(csFile.getCommand());
                csFile.exportForPerformance();
                csndFileString="";
                csFileCreated = true;
            }
            else{ csFileCreated = false; }
        }
        catch(IOException e){
            e.printStackTrace();
        }
    }

    private void createCSDFile(String csdString){
        if(csdObj!=null && (!csdString.equalsIgnoreCase(""))){
            CsoundFile csFile = csoundObj.getCsoundFile();
            csFile.setCSD(csdString);
            csFile.setMessageCallback();
            csFile.setCommand(csOptionsString);
            csoundObj.PreCompile();
            csFile.exportForPerformance();
            csndFileString="";
            csFileCreated = true;
        }
        else{
            csFileCreated = false;
        }
    }

    While this prototype uses Csound as it's synthesis engine, it is entirely possible to add OSC to allow control of any OSC aware synthesis engine such as, Pure Data or SuperCollider.

    2.1 Csound API

    As can be seen in the code excerpt above, the csnd object is used by the servlet. This object is created using the CppCsound interface and uses the CsoundPerformanceThread and CsoundCallbackWrapper classes to control real-time operation of Csound.

    The code shown below is from the Csound API service bundle; it creates the aforementioned objects, passing the CppSound object to the CsoundPerformanceThread object and setting up the callback object for channel input and retrieving console output messages.

    private void createCSDFile(String csdString){
        if(csdObj!=null && (!csdString.equalsIgnoreCase(""))){
            CsoundFile csFile = csoundObj.getCsoundFile();
            csFile.setCSD(csdString);
            csFile.setMessageCallback();
            csoundObj.PreCompile();
            csFile.exportForPerformance();
            csndFileString="";
            csFileCreated = true;
        }
        else{
            csFileCreated = false;
        }
    }

    While this prototype uses Csound as it's synthesis engine, it is entirely possible to add OSC to allow control of any OSC aware synthesis engine such as, Pure Data or SuperCollider.
3 WADE Examples

The current version of this application is being tested with the Ubuntu 10.10 Linux distribution and the Google Chrome web browser.

Once the desktop application has started, the “Welcome” view will be displayed, providing links to a number of pages informing you about the WADE application. Along the top of the application window you will see the obligatory main toolbar, from which you can access the preferences and console view.

Next, access the web-based code editor with slider bank in your web browser (as you would any web page). Once the page has loaded, click the “Csound Editor” and “Slider Bank” buttons to show the editor and associated faders. You will note that Instrument 2 in the CSD file has two channels “volChan” and “pitchChan”; these are controlled by the faders in the slider bank window. Press the “Play / Pause” button to send the CSD file to the WADE desktop application for rendering. It is possible to send live control information to the WADE desktop application via the faders in the slider bank.

A pedagogical application of this system, could see an interactive Csound manual created, or a large database of interactive Csound instruments made available to the sound synthesising community.

4 Future Developments

Current refactoring efforts are underway to resolve issues in line with a first public release of the system. Due to the integration of different open source technologies, the completed system and source code will likely be made available under the LGPL licence, with the obvious caveat when integrating other technologies; that their respective licences are adhered to and that the use of these projects is acknowledged. The project releases and source code will be available from the WADE project page on Sourceforge: http://wadesys.sourceforge.net/. The features being assessed are as follows, the dynamic generation of a RESTful OSC [12] API on a per instrument basis, using Apache CXF [13]; dynamic GUI slider bank generation; the inclusion of a HyperSQL database which could be used to store OSC packets for replaying a live performance; XMPP [14] chat client for real-time communication with other developers; XML specification for instruments, including what graphical widgets should be used to display the instrument.

The provision for extensibility and deployment options afforded by OSGi and the Eclipse Rich Client Platform could lead to the incorporation of features in the areas of: networked musical performance, sound installation frameworks, visual art development, cloud based audio synthesis and even pseudo-embedded synthesis systems. In short, it is possible that future iterations of this project will be deployed on small
form factor devices such as the BeagleBoard[15] or PandaBoard[16], to create a Csound based effects pedal, or across a number of large servers to provide a cloud synthesis solution.

5 Conclusion

The question of how to integrate these diverse technologies lead to the identification of the OSGi framework, which in turn lead to a much greater consideration of the software architecture of the project. While it can be shown that systems designed for a specific task are more likely to be less bloated and in many cases better suited to that task than larger programs designed to address numerous concerns[17] it was concluded that by designing a system which facilitated future expansion and development, the long term goal of creating a system capable of delivering a complete collaborative environment for audio synthesis development, learning and performance would be best satisfied.

6 Acknowledgements

I would like to acknowledge the help I received from the Csound mailing list, in particular I would like to thank Jacob Joaquin, Michael Goggins, Steven Yi and Victor Lazarini(Csound Mailing List); Dundalk Institute of Technology for their support of my research and finally, I wish to thank the contributors of the Eclipse help wiki and Lars Vogel for his Eclipse RCP programming tutorials.

References

http://www.osgi.org/About/WhyOSGi
http://www.w3.org/TR/html401/
http://www.ecmascript.org/index.php
Csound Journal. 11 (5) -no page numbers.
http://www.csounds.com/journal/issue11/Processing.html
http://www.igniterealtime.org/projects/smack/
http://dk.cput.ac.za/cgi/viewcontent.cgi?filename=17&article=1004&context=inf_papers&type=additional
Low-Latency Audio on Linux
by Means of Real-Time Scheduling

Tommaso CUCINOTTA and Dario FAGGIOLI and Giacomo BAGNOLI
Scuola Superiore Sant’Anna
Via G. Moruzzi 1, 56124, Pisa (Italy)
{t.cucinotta, d.faggioli}@sssup.it, g.bagnoli@asidev.com

Abstract
In this paper, we propose to use resource reservations scheduling and feedback-based allocation techniques for the provisioning of proper timeliness guarantees to audio processing applications. This allows real-time audio tasks to meet the tight timing constraints characterizing them, even if other interactive activities are present in the system. The JACK sound infrastructure has been modified, leveraging the real-time scheduler present in the Adaptive Quality of Service Architecture (AQuoSA). The effectiveness of the proposed approach, which does not require any modification to existing JACK clients, is validated through extensive experiments under different load conditions.

Keywords
JACK, real-time, scheduling, time-sensitive, resource reservation

1 Introduction and Related Work
There is an increasing interest in considering General Purpose Operating Systems (GPOSes) in the context of real-time and multimedia applications. In the Personal Computing domain, multimedia sharing, playback and processing requires more and more mechanisms allowing for low and predictable latencies even in presence of background workloads nearly saturating the available resources, e.g., network links and CPU power. In the professional multimedia domain, spotting on stages, it is becoming quite common to see a digital keyboard attached to a common laptop running GNU/Linux. DJs and VJs are moving to computer based setups to point that mixing consoles have turned from big decks into simple personal computers, only containing audio collections and running the proper mixing software.

In fact, developing complex multimedia applications on GNU/Linux allows for the exploitation of a multitude of OS services (e.g., networking), libraries (e.g., sophisticated multimedia compression libraries) and media/storage support (e.g., memory cards), as well as comfortable and easy-to-use programming and debugging tools. However, contrary to a Real-Time Operating System (RTOS), a GPOS is not generally designed to provide scheduling guarantees to the running applications. This is why either large amount of buffering is very likely to occur, with an unavoidable impact on response time and latencies, or the POSIX fixed-priority (e.g., SCHED_FIFO) real-time scheduling is utilized. This turns out to be difficult when there is more than one time-sensitive application in the system. Though, on a nowadays GNU/Linux system, we may easily find a variety of applications with tight timing constraints that might benefit from precise scheduling guarantees, in order to provide near-professional quality of the user experience, e.g., audio acquisition and playback, multimedia (video, gaming, etc.) display, video acquisition (v4l2), just to cite a few of them. In such a challenging scenario in which we can easily find a few tens of threads of execution with potentially tight real-time requirements, an accurate set-up of real-time priorities may easily become cumbersome, especially for the user of the system, who is usually left alone with such critical decisions as setting the real-time priority of a multimedia task.

More advanced scheduling services than just priority based ones have been made available for Linux during the latest years, among the others by [Palopoli et al., 2009; Faggioli et al., 2009; Checconi et al., 2009; Anderson and Students, 2006; Kato et al., 2010]. Such scheduling policies are based on a clear specification that needs to be made by the application about what is the

* The research leading to these results has received funding from the European Community’s Seventh Framework Programme FP7 under grant agreement n.214777 “IR-MOS – Interactive Realtime Multimedia Applications on Service Oriented Infrastructures” and n.248465 “S(o)OS – Service-oriented Operating Systems.”
computing power it needs and with what time granularity (determining the latency), and this scheme is referred to as resource reservations. This is usually done in terms of a reservation budget of time units to be guaranteed every period. The reservation period may easily be set equal to the minimum activation period of the application. Identifying the reservation budget may be a more involved task, due to the need for a proper benchmarking phase of the application, and it is even worse in case of applications with significant fluctuations of the workload (such as it often happens in multimedia ones). Rather, it is more convenient to engage adaptive reservation scheduling policies, where the scheduling parameters are dynamically changed at runtime by an application-level control-loop. This acts by monitoring some application-level metrics and increasing or decreasing the amount of allocated computing resources depending on the instantaneous application workload. Some approaches of this kind are constituted by [Segovia et al., 2010; Abeni et al., 2005; Nahrstedt et al., 1998], just to mention a few.

1.1 Contribution of This Paper

This work focuses on how to provide enhanced timeliness guarantees to low-latency real-time audio applications on GNU/Linux. We use adaptive reservations within the JACK audio framework, i.e., we show how we modified JACK in order to take advantage of AQuoSA [Palopoli et al., 2009], a software architecture we developed for enriching the Linux kernel with resource reservation scheduling and adaptive reservations. Notably, in the proposed architecture, JACK needs to be patched, but audio applications using it do not require to be modified nor recompiled. We believe the discussion reported in this paper constitutes a valuable first-hand experience on how it is possible to integrate real-time scheduling policies into multimedia applications on a GPOS.

2 JACK: Jack Audio Connection Kit

JACK 1 is a well-known low-latency audio server for POSIX conforming OSes (including Linux) aiming at providing an IPC infrastructure for audio processing where sound streams may traverse multiple independent processes running on the platform. Typical applications — i.e., clients — are audio effects, synthesisers, samplers, tuners, and many others. These clients run as independent system processes, but they all must have an audio processing thread handling the specific computation they make on the audio stream in real-time, and using the JACK API for data exchanging.

On its hand, JACK is in direct contact with the audio infrastructure of the OS (i.e., ALSA on Linux) by means of a component referred to as (from now on) the JACK driver or just the driver. By default, double-buffering is used, so the JACK infrastructure is required to process audio data and filling a buffer, while the underlying hardware is playing the other one. Each time a new buffer is not yet available in time, JACK logs the occurrence of an xrun event.

3 AQuoSA Resource Reservation Framework

The Adaptive Quality of Service Architecture (AQuoSA2) is an open-source framework enabling soft real-time capabilities and QoS support in the Linux kernel. It includes: a deadline-based real-time scheduler; temporal encapsulation provided via the CBS [Abeni and Buttazzo, 1998] algorithm; various adaptive reservation strategies for building feedback-based scheduling control loops [Abeni et al., 2005]; reclamation of unused bandwidth through the SHRUB [Palopoli et al., 2008] algorithm; a simple hierarchical scheduling capability which allows for Round Robin scheduling of multiple tasks inside the same reservation; a well-designed admission-control logics [Palopoli et al., 2009] allowing controlled access to real-time scheduling capabilities of the system for unprivileged applications. For more details about AQuoSA, the reader is referred to [Palopoli et al., 2009].

4 Integrating JACK with AQuoSA

Adaptive reservations have been applied to JACK as follows. In JACK, an entire graph of end-to-end computations is activated with a periodicity equal to \( \frac{\text{buffersize}}{\text{samplerate}} \) and it must complete within the same period. Therefore, a reservation is created at the start-up of JACK, and all of the JACK clients, comprising the real-time threads of the JACK server itself (the audio “drivers”), have been attached to such reservation, exploiting the hierarchical capability of AQuoSA. The reservation period has been set

\footnote{Note that the version 2 of JACK is used for this study}

\footnote{More information is available at: http://aquosa.sourceforge.net.}
equal to the period of the JACK work-flow activation. The reservation budget needs therefore to be sufficiently large so as to allow for completion of all of the JACK clients within the period, i.e., if the JACK graph comprises \( n \) clients, the execution time needed by all of the JACK clients are \( c_1, \ldots, c_n \), and the JACK period is \( T \), then the reservation will have the following budget \( Q \) and period \( P \):

\[
\begin{align*}
Q &= \sum_{i=1}^{n} c_i \\
P &= T
\end{align*}
\]  

(1)

Beside this, an AQuoSA QoS control-loop was used for controlling the reservation budget, based on the monitoring of the budget actually consumed at each JACK cycle. The percentile estimator used for setting the budget is based on a moving window of a configurable number of consumed budget figures observed in past JACK cycles, and it is tuned to estimate a configurable percentile of the consumed budget distribution (such value needs to be sufficiently close to 100%). However, the actual allocated budget is increased with respect to the results of this estimation by a (configurable) over-provisioning factor, since there are events that can disturb the predictor, making it potentially consider inconsistent samples, and thus nullify all the effort of adding QoS support to JACK, if not properly addressed. Examples are an xrun event and the activation of a new client, since in such case no guess can be made about the amount of budget it will need. In both cases, the budget is bumped up by a (configurable) percentage, allowing the predictor to reconstruct its queue using meaningful samples.

4.1 Implementation Details

All the AQuoSA related code is contained in the JackAquosaController class. The operations of creating and deleting the AQuoSA reservation are handled by the class constructor and destructor, while operations necessary for feedback scheduling — i.e., collect the measurements about used budget, managing the samples in the queue of the predictor, set new budget values, etc. — are done by the CycleBegin method, called once per cycle in the real-time thread of the server. Also, the JackPosixThread class needed some modifications, in order to attach real-time threads to the AQuoSA reservation when a new client registers with JACK, and perform the corresponding detach operation on a client termination.

The per-cycle consumed CPU time values were used to feed the AQuoSA predictor and apply the control algorithm to adjust the reservation budget.

5 Experimental Results

The proposed modifications to JACK have been validated through an extensive experimental evaluation conducted over the implemented modified JACK running on a Linux system. All experiments have been performed on a common consumer PC (Intel(R) E8400@3.00 GHz) with CPU dynamic voltage-scaling disabled, and with a Terratec EWX24/96 PCI sound card. The modified JACK framework and all the tools needed in order to reproduce the experiments presented in this section are available on-line ⁴.

In all the conducted experiments, results have been gathered while scheduling JACK using various scheduling policies:

- **CFS**: the default Linux scheduling policy for best effort tasks;
- **FIFO**: the Linux fixed priority real-time scheduler;
- **AQuoSA**: the AQuoSA resource reservation scheduler, without reclaiming capabilities;
- **SHRUB**: the AQuoSA resource reservation scheduler with reclaiming capabilities.

The metrics that have been measured throughout the experiments are the following:

- **audio driver timing**: the time interval between two consecutive activations of the JACK driver. Ideally it should look like an horizontal line corresponding to the value: \( \frac{\text{buffersize}}{\text{samplerate}} \);
- **driver end date**: the time interval between the start of a cycle and the instant when the driver finishes writing the processed data into the sound card buffer. If this is longer than the server period, then an xrun just happened.

When the AQuoSA framework is used to provide QoS guarantees, we also monitored the following values:

- **Set budget (Set Q)**: the budget dynamically set for the resource reservation dedicated to the JACK real-time threads;


⁴http://retis.sssup.it/~tommaso/papers/lac11/
• Predicted budget (Predicted Q): the value predicted at each cycle for the budget by the feedback mechanism;

Moreover, the CPU Time used, at each cycle, by JACK and all its clients has been measured as well. If AQuoSA is used and such value is greater than the Set Q, then an xrun occurs (unless the SHRUB reclaiming strategy is enabled).

First of all the audio driver timing in a configuration where no clients were attached to JACK has been measured, and results are shown in Table 1. JACK was using a buffer-size of 128 samples and a sample-rate of 96 kHz, resulting in a period of $1333\mu s$. Since, in this case, no other activities were running concurrently (and since the system load was being kept as low as possible), the statistics reveal a correct behaviour of all the tested scheduling strategies, with CFS exhibiting the highest variability, as it could have been expected.

Table 1: Audio driver timing of JACK with no clients using the 4 different schedulers (values are in $\mu s$).

<table>
<thead>
<tr>
<th></th>
<th>Min</th>
<th>Max</th>
<th>Average</th>
<th>Std. Dev</th>
</tr>
</thead>
<tbody>
<tr>
<td>CFS</td>
<td>1268</td>
<td>1555</td>
<td>1342.769</td>
<td>3.028</td>
</tr>
<tr>
<td>FIFO</td>
<td>1243</td>
<td>1423</td>
<td>1333.268</td>
<td>2.421</td>
</tr>
<tr>
<td>AQuoSA</td>
<td>1279</td>
<td>1389</td>
<td>1333.268</td>
<td>2.704</td>
</tr>
<tr>
<td>SHRUB</td>
<td>1275</td>
<td>1344</td>
<td>1333.268</td>
<td>2.692</td>
</tr>
</tbody>
</table>

5.1 Concurrent Experiments

To investigate the benefits of using reservations to isolate the behaviour of different — concurrently running — real-time applications, a periodic task simulating the behaviour of a typical real-time application has been added to the system. The program is called rt-app, and it is able to execute for a configurable amount of time over some configurable period.

The scheduling policy and configuration used for JACK and for the rt-app instance in the experiments shown below are given in Table 2.

In all of the following experiments, we used a “fake” JACK client, dnl, constituted by a simple loop taking about 7% of the CPU for its computations. The audio processing pipeline of JACK is made up of 10 dnl clients, added one after the other. This leads to a total of 75% CPU utilisation. When AQuoSA is used (i.e., in cases (4) and (5)), JACK and all its clients share the same reservation, the budget of which is decided as described in Section 4. Concerning rt-app, when it is scheduled by AQuoSA or SHRUB, the reservation period is set equal to the application period, while the budget is slightly over-provisioned with respect to its execution time (5%). Each experiment was run for 1 minute.

5.1.1 JACK with a period of $1333\mu s$ and video-player alike load

In this experiment, JACK is configured with a sample-rate of 96 kHz and a buffer-size of 128 samples, resulting in an activation period of $1333\mu s$, while rt-app has period of $40 ms$ and execution time of $5 ms$. This configuration for rt-app makes it resemble the typical workload produced by a video (e.g., MPEG format) decoder/player, displaying a video at 25 frames per second.

Figures 1a and 1b show the performance of JACK, in terms of driver end time, and of rt-app, in terms of response time, respectively. Horizontal lines at $1333\mu s$ and at $40 ms$ are the deadlines. The best effort Linux scheduler manages to keep the JACK performance good, but rt-app undergoes increased response-times and exhibits deadline misses in correspondence of the start and termination of JACK clients. This is due to the lack of true temporal isolation between the applications (rather, the Linux CFS aims to be as fair as possible), that causes rt-app to miss some deadlines when JACK has peaks of computation times. The Linux fixed-priority real-time scheduler is able to correctly support both applications, but only if their relative priorities are correctly set, as shown by insets 2 and 3 (according to the well-known rate-monotonic assignment, in this case rt-app should have lower priority than JACK). On the contrary, when using AQuoSA (inset 4), we achieve acceptable response-times for both applications: rt-app keeps its finishing time well below its deadline, whilst the JACK pipeline has sporadic terminations slightly beyond the deadline, in correspondence of the registration
of the first few clients. This is due to the over-provisioning and the budget pump-up heuristics which would benefit of a slight increase in those occasions (a comparison of different heuristics is planned as future work). However, it is worth mentioning that the JACK performance in this case is basically dependent on itself only, and can be studied in isolation, independently of what else is running on the system. Finally, when enabling reclaiming of the unused bandwidth via SHRUB (inset 5), the slight budget shortages are compensated by the reclaiming strategy: the small budget residuals which remain unused by one of the real-time applications at each cycle are immediately reused by the other, if needed.

For the sake of completeness, Figure 1c shows the CPU Time and, for the configurations using AQuoSA, the Set Q and Predicted Q values for the experiment. The figure highlights that the over-provisioning made with a high overall JACK utilisation is probably excessive with the current heuristic, so we are working to improve it.

5.1.2 JACK with a period of 2666µs and VoIP alike load

Another experiment, very similar to the previous one but with slightly varied parameters for the two applications has been run. This time JACK has a sample-rate of 48kHz and a buffer-size of 128 samples, resulting in a period of 2666µs, while rt-app has a period of 10ms and an execution time of 1.7ms. This could be representative of a VoIP application, or of a 100 Hz video player.
Results are reported in Figure 5. Observations similar to the ones made for the previous experiment may be done. However, the interferences between the two applications are much more evident, because the periods are closer to each other than in the previous case. Moreover, the benefits of the reclaiming logic provided by SHRUB appears more evident here, since using just a classical hard reservation strategy (e.g., the hard CBS implemented by AQuoSA on the 4th insets) is not enough to guarantee correct behaviour and avoid deadline misses under the highest system load conditions (when all of the clients are active).

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|c|}
\hline
& SHRUB & FIFO & CFS \\
\hline
Min. & 650.0 & 629.0 & 621.0 \\
Max. & 683.0 & 711.0 & 1369.0 \\
Average & 666.645 & 666.263 & 666.652 \\
Std. Dev & 0.626 & 1.747 & 2.696 \\
Drv. End Min. & 6.0 & 6.0 & 5.0 \\
Drv. End Max. & 552.0 & 602.0 & 663.0 \\
\hline
\end{tabular}
\caption{period and driver end time in the 3 cases (values are in µs).}
\end{table}

5.1.3 JACK alone with minimum possible latency
Finally, we considered a scenario with JACK configured to have only 64 samples as buffer-size and a sample-rate of 96kHz, resulting in 667µs of period. This corresponds to the minimum possible latency achievable with the mentioned audio hardware. When working at these small values, even if there are no other applications in the system and the overall load if relatively low, xruns might occur anyway due to system overheads, resolution of the OS timers, unforeseen kernel latencies due to non-preemptive sections of kernel segments, etc.

In Figures 2, 3 and 4, we plot the client end times, i.e., the completion instants of each client for each cycle (relative to cycle start time). Such metric provides an overview of the times at which audio calculations are finished by each client, as well as the audio period timing used as a reference. Things are working correctly if the last client end time is lower than the server period (667µs in this case). Clients are connected in a sequential pipeline, with Client0 being connected to the input (whose end-times are reported in the bottom-most curve), and Client9 providing the final output to the JACK output driver (whose end-times are reported in the topmost curve). Also notice that when a client takes longer to complete, the one next to it in the pipeline starts later, and this is reflected on the period duration too. Some more details about this experiments are also reported in Table 3.
6 Conclusions and future work

In this work the JACK sound subsystem has been modified so as to leverage adaptive resource reservations as provided by the AQuoSA framework. It appears quite clear that both best effort and POSIX compliant fixed priority schedulers have issues in supporting multiple real-time applications with different timing requirements, unless the user takes the burden of setting correctly the priorities, which might be hard when the number of applications needing real-time support is large enough. On the other hand, resource reservation based approaches allow each application to be configured in isolation, without any need for a full knowledge of the entire set of deployed real-time tasks on the system, and the performance of each application will depend exclusively on its own workload, independently of what else is deployed on the system. We therefore think that it can be stated that resource reservations, together with adaptive feedback-based control of the resource allocation and effective bandwidth reclamation techniques, allows for achieving precise scheduling guarantees to individual real-time applications that are concurrently running on the system, though there seems to be some space for improving the currently implemented budget feedback-control loop.

Along the direction of future research around the topics investigated in this paper, we plan to explore on the use of two recently proposed reservation based schedulers, the IRMOS [Checconi et al., 2009] hybrid EDF/FP real-time scheduler for multi-processor systems on multi-core (or multi-processor) platforms, and the SCHED_DEADLINE [Faggioli et al., 2009] patchset, which adds a new scheduling class that uses EDF to schedule tasks.

References


141
Figure 5: From top to bottom: driver end time and its CDF, JACK CPU Time and budgets, response time of rt-app and its CDF of the experiments with JACK and a VoIP alike load. As in Figure 1, time is in $\mu s$ on the Y axes of (a)-(c)-(d), while the X axes accommodate application cycles.
Abstract
This paper introduces a new Linux application implementing the loudness and level measuring algorithms proposed in recommendation R-128 of the European Broadcasting Union. The aim of this proposed standard is to ease the production of audio content having a defined subjective loudness and loudness range. The algorithms specified by R-128 and related standard documents and the rationale for them are explained. In the final sections of the paper some implementation issues are discussed.

Keywords
Loudness, metering, mastering, EBU

1 Introduction
Most radio listeners and TV viewers will probably agree that having to reach for the remote control to adjust the audio volume every so many seconds is a nuisance. Yet it happens all the time, and there are many reasons for this.

One of them is the nature of contemporary broadcast content, a large part of which consists of sequences of 'bumpers', commercials, previews and teasers of upcoming features, etc. All of these originate from different production sources and none of those is particularly interested in the final listener experience, let alone responsible for it.

In the 'old days' there would be a trained audio technician taking care of continuity and levels. Such a person would preview upcoming content, be familiar with the available metering and monitoring equipment, and above all, use his/her ears. In the worst case anything out of order would be adjusted promptly.

Today the situation is very different. You won’t find an audio technician in a typical TV continuity studio - more often than not audio is slaved to the video switcher and there are no level adjustments at all. For radio in many cases the presenter takes care of everything (or tries to), and much play-out is just automated without any human monitoring it.

Even if a recording is made by an audio technician knowing his business there will be problems. Imagine you are recording a talk with studio guest that will used later 'as live' in some program with the same presenter. You know the presenter will just click the start button and the interview will be played out without any level adjustments. At what level should you record it? The same problem occurs nearly all the time, for the simple reason that so much content is first produced 'out of context' and later used blindly and without any consideration of the context.

Broadcasters are aware of the problem but don’t have the means to do much about it. Most large organisations have technical guidelines which may or may not be followed for in-house production, and with varying degrees of success. Smaller ones usually just don’t care. And all of them are to some degree involved in the 'loudness war', and forced to increase levels rather than control them.

What is missing is some standard way to determine the 'loudness' of audio content, and one which can be easily automated. Current metering systems are of little use for this, as will be seen in later sections.

Given such a standard, it would be possible to define target loudness levels for any particular type of content or program. Audio technicians would know what to do when recording, and automated systems would be able to ‘measure’ the loudness of audio files and store the result in a database (or add it to the file as metadata) for use during play-out. Consumer playback systems could do the same. This could even lead to a much needed improvement in music recording practices: if music producers know that broadcasters and playback systems will adjust the level of their records anyway, there is no more reason to push it up using the absurd amounts of aggressive compression we see today.
An overview of current level metering practice

A number of quite different audio level measuring methods are being used today. Most of them do not provide any reliable measure of subjective loudness.

2.1 VU meters

The VU meter was designed in the days when audio equipment used tubes\(^1\) and therefore could use only simple electronics, at most an amplifier stage to drive the passive meter. But it was quite strictly specified.

A real VU meter, as opposed to something just looking like one,\(^2\) indicates the average of the absolute value of the signal (which is not the RMS level). For a fixed level signal, it should rise to 99\% of the final value in 300 ms, and overshoot it by 1 to 1.5\% before falling back. The small overshoot may seem a detail but it isn’t — it has quite a marked effect on the actual ballistics. These are determined not by any electronics but only by the properties of the moving-coil meter which is a classic mass + spring + damping system equivalent to a second order lowpass filter.

A real VU meter does provide some indication of loudness, but not a very accurate one in practice. Apart from that its dynamic range is quite limited.

2.2 Digital peak meters

These indicate the peak sample value, with a short holding time and/or a slow fallback. They are found in most consumer and semi-pro equipment and in almost all audio software. They can be useful to indicate signal presence and check digital recording levels for clipping, but they provide no useful loudness indication at all. And in fact even as peak meters most of them fail, as the real peaks are almost always between the samples.

2.3 Pseudo peak meters

A PPM, also known as ‘peak program meter’ is driven by the absolute value of the signal (again not RMS), but with a controlled rise time (usually 10 ms, sometimes 5) and a slow fallback. This is the most popular type of meter in broadcasting (at least in Europe), and in many professional environments. Specifications are provided by various organisational or international standards.

A pseudo peak meter can provide some idea of loudness, but only to an experienced user. The reason is that the relation between indicated level and effective loudness depends very much on the type of content, and some interpretation is required. This makes this type of metering algorithm unsuitable for automated loudness measurement.

2.4 Bob Katz’ K-meter

This is a relatively new way to measure and display audio levels, proposed by mastering expert Bob Katz. It displays both the digital peak and RMS values on the same scale. Since for normal program material the RMS level will always be lower than the peak value, and the intended use is based on controlling the RMS level (with the peak indication only as a check for clipping) the reference ’0 dB’ level is moved to either 20 or 14 dB below digital full scale. The ballistics are not specified in detail by Katz, but they should correspond to a simple linear lowpass filter, and not use different rise and fall times as for a PPM. Typical implementations use a response speed similar to a VU meter.

The K-meter provides quite a good indication of loudness, mainly because it uses the true RMS value, and because its response is not too fast. One way to improve this would be to add some filtering, and this is indeed what is done in the system discussed in the next sections.

2.5 Discussion

It should be clear that with the possible exception of the K-meter (which is not as widely used as it should be), current audio level metering systems provide a rather poor indication of actual subjective loudness.

Another issue is that all these level measurement systems were designed for interactive use, and only provide a ’momentary’ level indication. What is really needed is a way to automatically determine the average loudness of a recording, e.g. a complete song, in a reliable way and without requiring human interpretation.

Apart from such an average loudness value, another one of interest is the subjective loudness range of some program material — how much difference there is between the softer and louder parts. This value could for example guide the decision to apply (or not) some compression, depending on the listening conditions of the target audience.

Surprisingly few application or plugins for loudness measurement are available and widely

\(^{1}\)or values for some of us

\(^{2}\)as do most of them
used. The application presented at the LAC in 2007 [[Cabrera, 2007]] seems not to be actively developed anymore.

In the commercial world a notable exception is Dolby’s LM100. Early versions only supported \( L_{eq} \) based measurements, while recent releases also offer a mode based on the ITU algorithm discussed in the next section.

3 The ITU-R BS.1770 loudness algorithm

The EBU R-128 recommendation (discussed in the following section) is based on the loudness measurement algorithm defined in [[ITU, 2006a]]. This specification is the result of research conducted in several places over the past 10 years. Listening tests using hundreds of carefully chosen program fragments have shown a very good correlation between subjective loudness and the output of this algorithm. Details and more references can be found in the ITU document.

The ITU recommendation specifies the use a weighting filter followed by a mean square averaging detector. The filter response is shown in fig.1 and is the combination of a second order highpass filter (the same as in the \( L_{eq}(RLB) \) standard), and a second order shelf filter. The latter is added to model head diffraction effects. The combination of the two filters is called the K-filter

For multichannel use the mean squared values for each channel are multiplied by a weighting factor and added. This means that the powers are added and that inter-channel phase relationships have no effect on the result. For ITU 5.1 surround the weights for L,R and C are unity, +1.5 dB for the surround channels, and the LFE channel is not used. For stereo only L and R are used. In all cases there is just a single display for all channels combined — the idea is that a loudness meter would be used along with conventional per-channel level meters and not replace those.

The summed value is converted to dB, and a correction of -0.69 dB is added to allow for the non-unity gain of the K-filter at 1 kHz. This facilitates calibration and testing using standard 1 kHz signals. For a 0 dBFS, 1 kHz sine wave in either of L,R or C the result will be -3 dB. For the same signal in both L and R it will be 0 dB.

The ITU document does not specify if +3dB should be added when measuring a mono signal. Considering that a such a signal will in many cases be reproduced by two speakers (i.e. it is really the equivalent to a stereo signal with identical L and R) such a correction would seem to be necessary.

According to [[ITU, 2006a]] the output of a loudness measurement performed according to this algorithm should be designated ‘LKFS’ — Loudness using the K-filter, w.r.t. to Full Scale.

A second ITU document, [[ITU, 2006b]], provides some recommendations related to how loudness measurements should be displayed on a real device. In practice the LKFS scale is replaced by one that defines a ‘zero’ reference level at some point below full scale. To indicate that this is not a real level measurement the scale is marked in ‘LU’ (Loudness Units) instead of dB, with 1 LU being the same ratio as 1 dB. A linear range of at least -21 to +9 LU is recommended, but the reference level itself is not specified. This probably reflects the view that a different reference could be used in each application domain (e.g. film sound, music recording, . . . ).

This document also recommends the use of an ‘integrated’ mode to measure the average loudness of a program fragment, but again does not specify the algorithm in any detail.

4 The EBU R-128 recommendation

Recommendation R-128 [[EBU, 2010a]] builds on ITU-R BS.1770 and defines some further parameters required for a practical standard. More detail is provided by two other EBU documents, [[EBU, 2010b]] and [[EBU, 2010c]]. Two more are in preparation but not yet available at...
the time of writing.

4.1 Reference level and display ranges
R-128 defines the reference level as -23 dB relative to full scale, i.e. a continuous 1 kHz sine wave in both L and R and 23 dB below clipping corresponds to standard loudness.

It also requires meters conforming to this standard to be able to display levels either relative to this reference and designated LU, or to full scale and designated LUFS. Two display ranges should be provided: one from -18 to +9 dB relative to the reference level, and the second from -36 to +18 dB. The user should at any time be able to switch between these four scales. This choice then applies to all displayed values.

4.2 Dynamic response: M,S,I
Three types of response should be provided by a loudness meter conforming to R-128:

The M (momentary) response is the mean squared level averaged over a rectangular window of 400 ms. An R-128 compliant meter should also be able to store and show the maximum of this measurement until reset by the user.

The S (short term) response is the mean squared level averaged over a rectangular window of 3 seconds. R-128 requires this to be updated at least ten times per second. No such value is specified for the M response, but it seems reasonable to assume that at least the same update rate would be required.

The I (integrated) response is an average over an extended period defined by the user using Start, Stop and Reset commands. It is detailed in the following section.

4.3 Integrated loudness
The integrated loudness measurement is intended to provide an indication of the average loudness over an extended period, e.g. a complete song.

It is based on the full history, within an interval specified by the user, of the levels used for the M response. The input to the integration algorithm should consist of measurements in 400 ms windows that overlap by at least 200 ms.

Given this input, the integrated loudness is computed in two steps. First the average power of all windows having a level of at least -70 dB is computed. This absolute threshold is used to remove periods of silence which may occur e.g. at the start and end of a program segment. In a second step all points more than 8 dB below the first computed value are removed and the average power is recomputed. This second, relative threshold ensures that the integrated measurement is not dominated by long periods of relative silence as could occur in some types of program.

The result is the integrated loudness value, displayed as either LU or LUFS according to the scale selected by the user. This algorithm can be applied either in real time or on recorded audio. When a loudness meter is operating on real-time signals the indicated value should be updated at least once per second.

4.4 Loudness range, LRA
The purpose of the loudness range measurements is to determine the perceived dynamic range of a program fragment. This value can be used for example to determine if some compression would be necessary. The algorithm is designed to exclude the contribution of very short loud sounds (e.g. a gunshot in a movie), of short periods of relative silence (e.g. movie fragments with only low level ambient noises), and of a fade-in or fade-out.

The input to the LRA algorithm consists of the full history, within the same interval as for the integrated loudness, of the levels used for the S measurement. The windows used should overlap by at least 2 seconds.

First an absolute threshold of -70 dB is applied and the average value of the remaining windows is computed — this is similar to the first step for the integrated loudness (but using different input). A second threshold is then applied at 20 dB below the average value found in the first step. The lower limit of the loudness range is then found as the level that is exceeded by 90 percent of the remaining measurement windows, and the upper limit is the level exceeded by the highest 5 percent. In other words, the loudness range is the difference between the 10% and 95% percentiles of the distribution remaining after the second threshold.

4.5 True peak level
Both the ITU and EBU documents cited in previous sections recommend the use of true peak level indication in addition to loudness measurement.

Most peak meters just display the absolute value of the largest sample. There are two potential sources of error with this simple approach. First, almost all peaks occur between
the samples. To reduce the error from failing to see these peaks, [[ITU, 2006a]] recommends to upsample the signal by a factor of at least four. Second, the peak level may be different if later stages in the processing include DC-blocking — in fact it could be either higher or lower. For this reason it is recommended to measure peak levels both with and without DC blocking, and display the highest value.

The EBU documents do not require a continuous display of peak levels. Instead they recommend the use of a true peak indicator with a threshold of 1 dB below the digital peak level.

5 Implementation

The ebulm application (fig.2) is written as a Jack client. The upper bargraph shows either the $M$ or $S$ response. The two thinner ones below display the loudness range and the integrated loudness which are also shown in numerical form. To the right are some buttons that control the display range and scale, and below these the controls to stop, start and reset the integrated measurements.

Two features are missing in this version (but will be added): the display of the maximum value of the $M$ response, and the true peak indicator.

The ITU document specifies the K-filter as two biquad sections and provides coefficients only for a sample rate of 48 kHz, adding that implementations supporting other rates should use ‘coefficients that provide the same frequency response’. It is in general not possible to create exactly the same FR using a biquad at different rates, but the code used in ebulm comes close: errors are less than 0.01 dB at 44.1 kHz and much less at higher rates. Another peculiarity is that the highpass filter has a double zero at 0 Hz as expected, but the nominator coefficients given for 48 kHz are just $+1, -2, +1$ instead of values that would provide unity passband gain. This has to be taken into account when using a different sample rate.

There is no specified limit on the length of the integration period for the $I$ and $LRA$ measurements. A simple implementation of the algorithms would require unlimited storage size, and for the loudness range calculation the stored data would need to be sorted as well. The solution is to use histogram data structures instead — these require a fixed storage size, keep the data sorted implicitly, and make it easy to find the percentiles for the loudness range calculation. The current implementation uses two histograms, each having 751 bins and covering the range from -70 to +5 dB with a step of 0.1 dB. Points below -70 dB can be discarded, as the absolute threshold in both algorithms will remove them. Levels higher than +5 dB RMS over a 400 ms period mean the measurements will probably be invalid anyway. If such levels occur they are clipped to +5 dB and an error flag is set.

6 Acknowledgements

The work done by the EBU on loudness measurement and standardization is very much the result of an initiative taken by Florian Camerer, senior sound engineer at ORF and chairman of the PLOUD working group.

References


EBU. 2010c. EBU-TECH 3342, Loudness Range: A descriptor to supplement loud-


Medusa - A Distributed Sound Environment

Flávio Luiz SCHIAVONI¹ and Marcelo QUEIROZ¹ and Fernando IAZZETTA²
Computer Science Department¹, Music Department²
University of São Paulo
Brazil
{fls,mqz}@ime.usp.br, iazzetta@usp.br

Abstract
This paper introduces Medusa, a distributed sound environment that allows several machines connected in a local area network to share multiple streams of audio and MIDI, and to replace hardware mixers and also specialized multi-channel audio cables by network communication. Medusa has no centralized servers: any computer in the local environment may act as a server of audio/MIDI streams, and as client to remote audio/MIDI streams. Besides allowing audio and MIDI communication, Medusa acts as a distributed sound environment where networked sound resources can be transparently used and reconfigured as local resources. We discuss the implementation of Medusa in terms of desirable features, and report user experience with a group of composers from the University of São Paulo/Brazil.

Keywords
Network music, Jack, SCTP.

1 Introduction
With the growth of the Internet and the rise of broadband home links, the association between music making and networked computers had a global acceptance. With music distribution via streaming, computers became the new sound players, and also the new way of distributing music on a global scale. This kind of musical application of computers is not concerned about latency issues because communication is totally asynchronous. The interest in synchronous audio communication came with the idea of putting this technology to new uses [Wright, 2005].

Synchronous networked music communication research started with music performance experiments years ago. Some early network music performances are reported for instance in Bolot and García [Bolot and García, 1996] as early as 1996, using TCP, UDP and RTP to route voice signals; see [Weinberg, 2002], [Renaud et al., 2007] and [Barbosa, 2003] for surveys on network music performance.

Medusa - A Distributed Sound Environment

Flávio Luiz SCHIAVONI¹ and Marcelo QUEIROZ¹ and Fernando IAZZETTA²
Computer Science Department¹, Music Department²
University of São Paulo
Brazil
{fls,mqz}@ime.usp.br, iazzetta@usp.br

Abstract
This paper introduces Medusa, a distributed sound environment that allows several machines connected in a local area network to share multiple streams of audio and MIDI, and to replace hardware mixers and also specialized multi-channel audio cables by network communication. Medusa has no centralized servers: any computer in the local environment may act as a server of audio/MIDI streams, and as client to remote audio/MIDI streams. Besides allowing audio and MIDI communication, Medusa acts as a distributed sound environment where networked sound resources can be transparently used and reconfigured as local resources. We discuss the implementation of Medusa in terms of desirable features, and report user experience with a group of composers from the University of São Paulo/Brazil.

Keywords
Network music, Jack, SCTP.

1 Introduction
With the growth of the Internet and the rise of broadband home links, the association between music making and networked computers had a global acceptance. With music distribution via streaming, computers became the new sound players, and also the new way of distributing music on a global scale. This kind of musical application of computers is not concerned about latency issues because communication is totally asynchronous. The interest in synchronous audio communication came with the idea of putting this technology to new uses [Wright, 2005].

Synchronous networked music communication research started with music performance experiments years ago. Some early network music performances are reported for instance in Bolot and García [Bolot and García, 1996] as early as 1996, using TCP, UDP and RTP to route voice signals; see [Weinberg, 2002], [Renaud et al., 2007] and [Barbosa, 2003] for surveys on network music performance.

A tool for network music interaction might be used to promote interaction on a global or a local scale. In a wide area network such as the Internet, the main concern is the attempt to bring people together across physical space, whereas in a local area network context, where participants are usually in the same room, the network can be used to promote a rich range of interaction possibilities, by using the virtual communication link as an extension of the shared physical space [Wright, 2005]. Technologically mediated communication brings significant contributions to musical interaction even when people are face-to-face, for instance by allowing much more control in processing and combining sound sources within a room with almost no interference of room acoustics.

These new possibilities can be explored by musicians, allowing them to create new musical approaches to composition and performance, by exploring new ways of interacting that exceed physical proximity and maximize musical possibilities. There is some expectation about what could or would be done with music when this kind of free networked intercommunication is allowed [Cáceres and Chafe, 2009a]. As noted by [Chafe et al., 2000], “once [the delay issue] can be pushed down to its theoretical limit, it will be interesting to see what musical possibilities can be made of truly interactive connections”.

1.1 Goals and related work
This paper introduces Medusa, an audio/MIDI communication tool for local networks whose design is based on a set of desirable features, which have been collected from several previous works in Network Music Performance, Interactive Performance and Distributed Systems.

The main goal is to unleash audio/MIDI communication between computers and software applications on a local area network without complex configurations or difficult set-ups. This is
done by mapping each sound source (or sound sink) in the network to a local name that the user may connect to any input (or output) of audio/MIDI software applications. The focus on local area networks allows the mapping of musician’s expectations based on local/physical/acoustical musical interaction to new desirable features of the system, and then the mapping of these desirable features to details of the software model.

Several audio processing platforms allow some form of network communication of audio and MIDI data. PureData, for instance, allows the user to send and receive UDP messages between several Pd instances using netPD. SuperCollider is implemented with a client-server architecture and also allows network communication. The goal of Medusa, on the other hand, is to allow communication also between different software tools and across computer platforms.

Some related work address the problem of synchronous music communication between networked computers, such as OSC [Lazzaro and Wawrzynek, 2001], NetJack [Carôt et al., 2009], SoundJack [Carôt et al., 2006], JackTrip [Cáceres and Chafe, 2009b; Cáceres and Chafe, 2009a], eJamming [Renaud et al., 2007], Otherside [Anagnostopoulos, 2009] and LDAS [Sæbø and Svensson, 2006], including commercial applications such as ReWire from Propellerhead [Kit, 2010].

Although some of the goals and features of these applications may overlap with those of Medusa, none of them addresses the issues of peer-to-peer topology for audio and MIDI communication in the specific context of Local Area Networks. The OSC standard, for instance, uses symbolic messages (e.g. MIDI) to control remote synthesizers over IP [Lazzaro and Wawrzynek, 2001]; Otherside is another example of a tool which works only with MIDI. While Medusa is based on peer-to-peer connections, NetJack works with a star topology and master/slave approach [S.Letz et al., 2009], and so do LDAS, SoundJack and JackTrip. Some of these tools allow WAN connections, which leads to a different application context with several other kinds of problems like NAT routing, package loss, greater latency and need for audio compression, and at the same time they do not fully exploit the specificities of LAN connections, for instance reliable SCTP routing. Besides, one of Medusa’s central goals is to go beyond audio and MIDI routing, by adding on-the-fly remote node reconfiguration capabilities that may help environment setup and tuning.

1.2 Design based on desirable features

In this paper, we will discuss an architectural approach to the design of a local network music tool which is based on desirable features, either found in previous work from the literature or in actual usage with a group of volunteer musicians. We will also present a prototype that was implemented to support some of the features mapped so far. The current list of desirable features guiding the development of Medusa is the following:

- Transparency
- Heterogeneity
- Graphical display of status and messages
  - Latency and communication status
  - Network status
  - Input/Output status
  - IO stream amplitudes
- Multiple IO information types
  - Audio
  - MIDI
  - Control Messages
  - User text messages
- Legacy software integration [Young, 2001]
  - Audio integration
  - MIDI integration
  - Control integration
- Sound processing capabilities [Chafe et al., 2000]
  - Master Mixer [Cáceres and Chafe, 2009a]
  - Silence Detection [Bolot and García, 1996]
  - Data compression [Chafe et al., 2000]
  - Loopback [Cáceres and Chafe, 2009a]

Transparency and Heterogeneity are desirable features borrowed from the field of distributed systems. Transparency’s main idea is
to provide network resources as if they were local resources in a straightforward way. Heterogeneity means that the system should be able to run on several system configurations within the network, including different OS and different hardware architectures, in an integrated manner. These concerns also appear in related works [Wright, 2005; Cáceres and Chafe, 2009a], and helped in the choice of a development framework (including programming language, API, sound server, etc.)

The features listed under Graphical display of status and messages were collected via experimentation with potential users (volunteer musicians), in a cyclic process of update and feedback of early versions of the prototype. These features are directly related to the graphical user interface.

The need to work with both MIDI and Audio was also presented by volunteer musicians, as they frequently combine audio connections with the use of remote MIDI controllers. Control Messages are used to access a remote machine, for instance to reconfigure its audio connections during a musical performance. Also user text messages may be used for various purposes including machine reconfiguration and performance synchronization.

The need to integrate the system with legacy softwares is evident as every user is used to work with particular sound processing applications. Like Heterogeneity, this feature also determines the choice of a development API.

Sound processing capabilities include a set of tools that relate to the issues of latency, bandwidth and heterogeneity. These features will be further discussed in the Sound Communication section.

2 Architectural Approach

The system presupposes a set of computers in a local area network that may share audio and MIDI channels. In the context of this paper, the group of all machines connected to Medusa is called environment and every machine in the environment is called a node. A node that makes resources available to the environment, such as audio or MIDI streams, is called a source, and a node that uses environmental resources is called a sink; every machine can act simultaneously as source and sink. Every node has independent settings, and each user can choose which resources he or she wants to make available to the environment, and also which environmental resources he or she wants to use, and when. The following subsections discuss the architectures of each node and of the environment.

2.1 Node Architecture

The node architecture is a multi-layered model that uses the following components:

- GUI: used for configuring the node and interacting with the environment. Environment interaction includes adding/removing local audio/MIDI ports and environmental node search and connection;
- Model: used to represent the node configuration, including audio and network configurations and their current status.
- Control: is responsible for integrating sound resources and network communication.
- Network communication: used for data and control communication with the environment;
- Sound resources: used to map local and environmental audio resources.

The GUI is the user interaction layer. It is used to set up the system, to create audio channel resources, to connect to the environment and to remote resources. The GUI brings some level of transparency to the environment and makes the tool easier to use, by hiding the complexity of actual network connections and network and audio settings. [Cáceres and Chafe, 2009b] already noted that usually most of the time is spent adjusting the connections rather than playing music, and our GUI was designed trying to alleviate this problem. The feature of graphical display of status and messages is implemented by this layer. The status of the network and active communication channels are presented as indicators that provide visual feedback, such as Latency, Network status, Input/Output status and Signal Amplitude, which help the user in interacting with the network music environment.

The Model layer represents each node current status at any given moment. It contains the network configuration, sound resources and coding details such as number of channels, sample rate, local buffer size and other relevant information. The model is encapsulated in messages to preserve consistency between machines. The
set of models of all nodes represents the current environment status. Messages will be further explained in section 2.3.

The Control layer is the main part of the system. It is divided in three components: Sound Control, Network Control and Environment Control. These controls hide the implementation details from upper-level components, by taking care of audio synchronization, sound processing and message exchange to keep the model representation up-to-date across nodes. The Environment Control maintains an environment node list with all details of the nodes known at each moment. The Sound Control encapsulates the Sound Communication layer, allowing the sound server to be changed at any time. The Network control encapsulates the network servers and clients allowing a server reimplementation without the need for major code rewriting.

The Network Communication layer is responsible for the low-level maintenance of the network infrastructure. It connects sources and sinks to audio and MIDI streams and manages control messages within the environment. Broadcast control messages can be used to sync all nodes in the environment. Plain text messages between users can help them to set up his/her node or to exchange any other kind of information in a human-readable way. The network communication layer has three servers:

- UDP server: send/receive broadcast messages;
- TCP server: send/receive unicast messages;
- SCTP server: exchange audio/MIDI streams.

The Sound Communication layer is responsible for interacting locally with the sound server in each node, creating a virtual layer that provides transparent access to remote audio and MIDI streams, integrating the tool with other legacy sound softwares, while hiding the details of network communication. The idea behind integration with legacy softwares is to avoid having any type of signal processing units within the communication tool, leaving those tasks to external softwares through a sound server like Jack or SoundFlower. The architecture can integrate, via external software, many other sound processing capabilities that may be applied before sending a stream to the network or upon receiving a stream and before making it locally available. Signal processing can be used, for instance, to translate streams with different audio codings, by adjusting sample rate, sample format, buffer size and other coding details between different user configurations [Chaﬂ et al., 2000], thus providing heterogeneous and transparent access to remote data. Signal processing units may also include:

**Master Mixer:** allows the user to independently control the volume of network audio and MIDI inputs, and also to mute them. It allows groups of network channels to be mixed before being connected to a sound application, and to create mixed output channels consisting of several local sound streams. For added versatility the mixer has a gain that exceeds 100% (or 0 dB) with respect to the incoming signal level, allowing the user to boost weak signals or even distort regular signals up to 400% (or 12 dB) of their original amplitude level.

**Data compression:** in order to minimize transmission latency, data compression can be applied to the signal, reducing the amount of audio data transmitted. Codecs like CELT [Carrot et al., 2009] can be used to reduce the amount of data without significant audio loss. Audio compression also reduces transmission bandwidth, which allows more audio channels to be sent over the same transmission link. On the other hand, compressing a signal introduces an algorithmic latency due to the encode/decode cycle, which is why some systems prefer to use uncompressed audio [Cáceres and Chaﬂ, 2009a]. We believe this decision is better left to the user, and the communication tool should have an option for turning compression on/off and also for tweaking compression parameters, allowing a finer control over sound quality and algorithmic latency.

**Silence Detection:** silence detection algorithms might be used to avoid sending “empty” audio packets to the network, using up transmission bandwidth needlessly. This feature introduces a non-deterministic element in bandwidth usage, and so its use is subject to user discretion.
2.2 Environment architecture

The environment architecture represents how node instances interact with each other within the client-server model proposed. Node interaction includes audio/MIDI streaming, and control communication via messages used to reset the environment or to change its current status. To do this, SCTP is used to deal with streaming and TCP and UDP servers deal with messages for environment control.

Control messages (see figure 1) are XML-based action commands that are managed by the control/model layer components; their results are displayed in the GUI. Messages are used to update and extend the local model, for instance by adding new information about remote machines and streams, removing streams for users that logged out, etc. The choice between UDP or TCP corresponds to sending a message to all nodes (broadcast) or to send a message to a specific node (unicast).

Control messages (see figure 1) are XML-based action commands that are managed by the control/model layer components; their results are displayed in the GUI. Messages are used to update and extend the local model, for instance by adding new information about remote machines and streams, removing streams for users that logged out, etc. The choice between UDP or TCP corresponds to sending a message to all nodes (broadcast) or to send a message to a specific node (unicast).

![Figure 1: XML Message](image)

2.3 Environment Messages

The tool has messages that inform the local node about the current state of the environment. A report is sent to all users whenever a new user connects to the environment, when a user connects to a remote output port, or when any kind of environment configuration is changed. Messages may be of Broadcast (B) or Unicast (U) communication type:

HI_GUYS (B): This message is sent when a node enters the environment. It is composed by the IP address, network port, audio ports, MIDI ports and name of the user. When a machine receives this message it will add a new node to the environment node list and send back HI_THERE and LOOP_BACK messages.

HI_THERE (U): This message is sent when a machine receives a HI_GUYS message. It sends information back in order to help the new node to update its environment node list. The fields of this message are the same of HI_GUYS. Whenever a machine receives this message, it will add or replace the corresponding node of the environment list, and send back a LOOP_BACK message.

LOOP_BACK (U): After receiving a HI_GUYS or HI_THERE message, the node uses this message to measure the latency between the corresponding pair of nodes. This message contains the sender and target node names and a time-stamp field with the local time at the sender node. Whenever a machine receives a LOOP_BACK message it will first check for the sender: if the local machine is the sender, it will calculate the latency to the target node by halving the round-trip time; otherwise it will only send the message back to the sender.

BYE (B): This message is used to inform all nodes that a machine is leaving the environment. When a machine receive a BYE message it will disconnect the corresponding audio sinks (if any) and remove the node from the node environment list.

CONNECTED/DISCONNECTED (U): This pair of messages inform a node that a sink is connected to one of its sound resources (passed as an argument), or that a sink just disconnected from that sound resource.

CHAT (B): Used to exchange human-readable messages within the environment. It may help with synchronization (of actions, for instance) and node setup.

CONNECT_ME/DISCONNECT_ME (U): Ask a node to connect to (or to disconnect from) a source. These are useful to allow configuration of the environment in a transparent way.

ADD_PORT/REMOVE_PORT (U): Ask a node to add or remove a audio/MIDI port. This message contains the sound port type (audio or MIDI) and the sound port name, and is used for remote management.

CONNECT_PORT/DISCONNECT_PORT (U): Ask a node to change audio connections in a local sound route. It may be used for remote configuration: with this message one node might totally reconfigure another node’s audio routing.

START_TRANSPORT/STOP_TRANSPORT (B): The transport message in the Jack sound server is used to start all players, recorders and other software that respond
to a play/start button. It is used for instance in remote playback/recording, or to synchronize actions during performance.

3 Implementation and Results

To allow for a multi-platform implementation, some choices regarding the development framework were made.

The Medusa implementation uses QT [Nokia, 2011] for the GUI implementation, XML encapsulation, and UDP/TCP communication. For streaming, SCTP [HP, 2008] [Ong and Yoakum, 2002] was used as a transport protocol alternative to the usual UDP/RTP protocols. The current implementation of Medusa uses Jack [JACK, 2011] as sound server, and one of the core issues is to extend the functionalities of Jack to enable multi-channel routing of audio and MIDI through a computer network. All these libraries are licensed under GPL and work in Linux, Windows and MacOS. The C++ programming language was chosen because of the object-oriented design of the framework, and also because it is used by QT and Jack.

The preliminary results with a prototype implementation used by a group of composers at the Music Department at the University of S˜ ao Paulo presented some interesting possibilities in network music. Using a wifi connection we were able to route 4 channels of uncompressed audio at 44.1 kHz without noticeable latency. MIDI channels were used to allow for MIDI synthesizers using remote controllers. Control messages were successfully used for automatically setting up the environment, which lessened the burden of the users. Broadcasting node information allowed users to connect to remote resources, and a constantly updated GUI showed whether remote users were accessing local resources.

LAN nodes were associated to user names, making it easy for a user to identify peers and create connections (see figure 2). Each user is allowed to configure its audio settings independently from the others (see figure 3). Figure 4 shows the GUI that corresponds to the ADD_PORT/REMOVE_PORT messages. Connections between local and remote audio inputs...
and outputs are transparent and can be made using Jack’s interface qJack as in figure 5.

4 Conclusions and future work

One relevant subjective conclusion at this point is the recognition that an user-friendly, graphical tool for network music may encourage musicians to experiment and play using networks. The possibilities of using a local area network for musical performance go beyond the common use of computers in live electronics, by allowing the distribution of computer processing and musical tasks among several performers and a heterogeneous group of computers and sound processing software. Network group performance on wireless connections is a fertile ground for musicians, composers and audio professionals. On the technical side, we observed that SCTP is a reliable protocol for sound exchange of a small number of audio channels, with unnoticeable latency and without packet loss on a local area network.

The next step in the validation of this tool is to measure latency, transmission bandwidth and network performance with different transmission links such as crossover cables, wireless connections, 10/100 Hubs and others. We would also like to have Medusa available to other platforms like PulseAudio, ALSA, PortAudio, ASIO and SoundFlower.

In order to allow remote connections outside of the Local Area Network (e.g. Internet), we would like to implement audio/MIDI communication using other transport protocols such as UDP and TCP in addition to SCTP. Since the SCTP protocol avoids packet loss, sticking to SCTP when going from LAN to WAN would make latency go way beyond an acceptable range.

5 Acknowledgements

The authors would like to thank the support of the funding agencies CNPq and FAPESP - São Paulo Research Foundation (grant 2008/08623-8).

References


Xth Sense: researching muscle sounds
for an experimental paradigm of musical performance

Marco DONNARUMMA
Sound Design, ACE, The University of Edinburgh
Alison House, Nicolson Square
Edinburgh, UK, EH8 9DF
m.donnarumma@sms.ed.ac.uk

Abstract

This paper seeks to outline methods underlying the development of the Xth Sense project, an ongoing research which investigates exploratory applications of biophysical sound design for musical performance and responsive milieux. Firstly, an aesthetical study of body sounds, namely muscle sounds is illustrated. I describe the development of an audio synthesis model for muscle sounds which offered a deeper understanding of the body sound matter and provided the ground for further experimentations in signal processing and composition. Then follows a description of the development and design of the Xth Sense, a wearable hardware sensor device for capturing biological body sounds; this was implemented in the realization of Music for Flesh I, a first attempt at musical performance. Next, the array of principles underpinning the application of muscle sounds to a musical performance is illustrated. Drawing from such principles, I eventually describe the methods by which useful features were extracted from the muscle sounds, and the mapping techniques used to deploy these features as control data for real time sound processing.

Keywords

Biosensing technologies, biophysical control, muscle sounds.

1 Introduction

Biosensing musical technologies use biological signals of a human subject to control music. One of the earliest applications can be identified in Alvin Lucier's Music for Solo Performer (1965). Alpha waves generated when the performer enters a peculiar mind state are transduced into electrical signals used to vibrate percussion instruments. Over the past thirty years biosensing technologies have been comprehensively studied [3, 8, 13, 14, 15, 18, 22] and presently notable biophysical-only music performances are being implemented at SARC by a research group lead by the main contributor to the Bio Muse project Ben Knapp (10).

Whereas biological motion and movement and music are arising topics of interest in neuroscience research [5, 12, 21], the biologic body is being studied by music researchers as a mean to control virtual instruments. Although such approach has informed gestural control of music, I argue that it overlooks the expressive capabilities of biological sounds produced by the body. They are inaudible but may retain a meaningful vocabulary of intimate interactions with the musicians' actions.

To what extent could biologic sounds be employed musically? In which ways could the performer's perceptual experience be affected? How could such experimental paradigm motivate an original perspective on musical performance?

2 Aesthetic principles

The long-term outcome of the research is the implementation of low cost, open source tools (software and hardware) capable of providing musicians, performers and dancers with a framework for biosensors-aided auditive design

---

1 A digital muscle sounds generator.

2 Bio Muse Trio, GroundMe!

3 Queen's University, Sonic Art Research Center, Belfast, UK.

4 A commercialized product exploiting electromyography and brainwaves analysis systems for musical applications.
in a real time environment; which framework will be re-distributable, customizable and easy to set up. However, given the substantial interdisciplinary quality of such project, its realization process needed to be fragmented into more specific and measurable steps.

The primary aim of the inquiry was to explore the musical and design capabilities of biological sounds of the body in a functional context – the production of Music for Flesh I a sonic solo performance for wearable biosensing device, which could demonstrate an experimental coupling between theatrical gesture and muscle sounds. In an attempt to inform the present state of augmented musical performance and embodied interaction, the characteristics of this pairing were identified in: the authenticity of the performer’s somatic interaction, the natural responsiveness of the system and the expressive immediacy and transparency of the mapping of biological sound to the performer’s kinetic behaviour. Such work required an interdisciplinary approach embracing biomedical computing studies, music technology and most importantly sound design. In fact, as I will demonstrate later in this text, the major research issue was not a technical implementation, but rather the definition of design paradigms by which the captured biological sounds could achieve a meaningful and detailed expressiveness.

3 Methods: understanding and capturing muscle sounds

The earliest approach to muscle sounds consisted of an analysis of the physical phenomena which makes muscle vibrate and sound. This study eventually developed in a sound synthesis model of muscle sounds. Although such model was not strictly related to the physical properties of muscle sounds, but rather to their aesthetic characteristics, it provided sonic samples which would satisfyingly resemble the original ones. Thereafter, the synthesised samples were used to explore design methodologies, while the sensor hardware implementation was still in progress.

Following this initial study, the scope of the research consisted of two interrelated strands. The first concerned the design and implementation of a wearable biosensing hardware device for musical performance; the second included the development of a tracking system for a performer’s somatic behaviour by means of muscle sounds features extraction and data mapping methods.

The study of a synthesis model of muscle sounds is described in the next paragraph, whereas the research methods employed during the hardware and software design are discussed in the following paragraphs; however, being the focus of this paper on the research methodology, specific signal processing techniques and other technical information are not illustrated in detail, but they are fully referenced.

3.1 An audio synthesis model of muscle sounds

Muscles are formed by several layers of contractile filaments. Each of them can stretch and move past the other, vibrating at a very low frequency. However, audio recordings of muscle sounds show that their sonic response is not constant, instead it sounds more similar to a low and deep rumbling impulse. This might happen because each filament does not vibrate in unison with each other, but rather each one of them undergoes slightly different forces depending on its position and dimension, therefore filaments vibrate at different frequencies. Eventually each partial (defined here as the single frequency of a specific filament) is summed to the others living in the same muscle fibre, which in turn are summed to the muscle fibres living in the surrounding fascicle.

Such phenomena creates a subtle, complex audio spectra which can be synthesised using discrete summation formula (DSF). This technique allows the synthesis of harmonic and in-harmonic, band-limited or unlimited spectra, and can be controlled by an index [7], which seemed to fit the requirement of such acoustic experiment.

Being that the use of open source technologies is an integral part of the project, a Linux operating system was chosen as development environment. Muscle sound audio synthesis model was implemented using the open source framework known as Pure Data (Puckette 1996), a graphical programming language which offers a flexible and

---

5 BAAD is a novel term used by the author to indicate a specific sound design practice which relies on the use of biological signals. Although in this context is not possible to further elaborate on this practice, its essential principles are defined in paragraph 4.1.

6 Real time refers here to a computing system in which there exists no perceivable delay between performer’s actions and sonic response.
powerful architecture for real time sonic synthesis and processing. DSF was first used to generate the fundamental sidebands of the model; then the same formula was applied to a noise generator in order to add some light distortion to the model by means of complex spectra formed by small, slow noise bursts. Filter banks were applied to each spectra in order to emphasise specific harmonics, thus refining the design. Eventually the two layers were summed, passed through a further filter bank and a tanh function⁷, which added a more natural characteristic to the resulting impulse. The model also included an automated random envelope generator used to constantly change the duration and intensity of individual impulses, thus better simulating a human muscle contraction.

Model was then embedded in a parent patch⁸ in order to evaluate the suitability of diverse signal processing techniques. Although testing showed interesting results with most of the applied processes, single-sideband pitch shifting (SSB modulation) proved to be the most meaningful method; in fact, being the muscle resonance frequency so low to not be immediately perceivable to human ear, namely between 5Hz and 40/45Hz, it would result difficult to produce heterogeneous sonic material to be used in a musical performance. SSB modulation [4] disclosed a new viewpoint on the further use of muscle sounds, allowing me to shift the initial spectrum of muscle fibre sound to a higher frequency range⁹; such method enriched the musical pitch range of muscles, prompting the composition of a more elaborate score.

3.2 Xth Sense: first prototype sensor implementation

Before undertaking the development of the Xth Sense sensor hardware, few crucial criteria were defined:

- to develop a wearable, unobtrusive device, allowing a performer to freely move on stage;
- to implement an extremely sensitive hardware device which could efficiently capture in real time and with very low latency diverse muscle sounds;
- to make use of the most inexpensive hardware solutions, assuring a low implementation cost;
- to implement the most accessible and straightforward production methodology in order to foster the future re-distribution and openness of the hardware.

Study of the hardware sensor design began with a contextual review of biomedical engineering papers and publications focused on mechanical myography (MMG). The mechanical signal which can be observed from the surface of a muscle when it is contracted is called a MMG signal. At the onset of muscle contraction, significant changes in the muscle shape produce a large peak in the MMG. The oscillations of the muscles fibers at the resonance frequency of the muscle generate subsequent vibrations. The mechanomyogram is commonly known also as the phonomyogram, acoustic myogram, sound myogram or vibromyogram.

Interestingly, MMG seems not to be a topic of interest in the study of gestural control of music and music technology; apparently many researchers in this fields focus their attention on electromyography (EMG), electroencephalography (EEG), or multidimensional control data which can be obtained through the use of wearable accelerometers, gyroscopes and other similar sensors. Notwithstanding the apparent lack of pertinent documentation in the studies of gestural control of music and music technology, useful technical information regarding different MMG sensor designs were collected by reviewing the recent biomedical engineering literature.

In fact, MMG is currently the subject of several investigations in this field as alternative control data for low cost, open source prosthetics research and for general biomedical applications [1, 6, 9, 20]. Most notably the work of Jorge Silva¹⁰ at Prism Lab was essential to further advance the research; his MASc thesis extensively documents the design of the CMASP, a coupled microphone-accelerometer sensor pair (figure 1) and represents


⁸ In this context the term ‘patch’ refers to a Pure Data-based application.

⁹ It was interesting to note that pitch-shifted muscles sounds quite closely resemble a plucked chord.

¹⁰ See: http://jsilva.komodoopenlab.com/index.php/Main/Research#toc6
a comprehensive resource of information and technical insights on the use and analysis of MMG signals [19].

The device designed at Prism Lab is capable of capturing the audio signal of muscles sounds in real time. Muscle sonic resonance is transmitted to the skin, which in turn vibrates, exciting an air chamber. These vibrations are captured by an omnidirectional condenser microphone adequately shielded from noise and interferences by mean of a silicon case. A printed circuit board (PCB) is used to couple the microphone with an accelerometer in order to filter out vibrations caused by global motion of the arm, and precisely identify muscle signals. Microphone sensitivity ranges from 20Hz up to 16kHz, thus it is capable of capturing a relevant part of the spectrum of muscles resonances.

Although this design has been proved effectively functional through several academic reports, criteria of my investigation could have been satisfied with a less complex device. Supported by the research group at Dorkbot ALBA, I could develop a first, simpler MMG sensor: the circuit did not make use of a PCB and accelerometer, but deployed the same omnidirectional electret condenser microphone indicated by Silva (Panasonic WM-63PRT). This first prototype was successfully used to capture actual heart and forearm muscles sounds; earliest recordings and analysis of MMG signals were produced with the open source digital audio workstation Ardour2 and benchmark were set in order to evaluate the signal-to-noise ratio (SNR).

![Figure 1. CMASP schematic](image)

In spite of the positive results obtained with the first prototype, the microphone shielding required further trials. The importance of the shield was manifold; an optimal shield had to fit specific requirements: to bypass the 60Hz electrical interference which can be heard when alternating electric current distribute itself within the skin after a direct contact with the microphone metal case; to narrow the sensitive area of the microphone, filtering out external noises; to keep the microphone static, avoiding external air pressure that will affect the signal; to provide a suitable air chamber for the microphone, in order to amplify sonic vibrations of the muscles, and facilitating capture of deeper muscle contractions.

First, the microphone was insulated by mean of a polyurethane shield, but due to the strong malleability of this material, its initial shape tended to flex easily. Eventually, the sensor was insulated in a common silicon case that satisfied the requirements and further enhanced the SNR. Once the early prototype had reached a good degree of efficiency and reliability, the circuit was embedded in a portable plastic box (3.15 x 1.57 x 0.67) along with an audio output (¼ mono chassis jack socket) and a cell holder for a 3V coin lithium battery.

![Figure 2. Xth Sense wearable MMG sensor prototype](image)

Shieldsed microphone was embedded in a Velcro bracelet and needed wiring cables were connected to the circuit box (figure 2).

4 Performance testing: mapping and design definitions

At this stage of the project the understanding and creation of mapping and design paradigms for muscles sounds was the major goal. The main

---

11 It is interesting to observe that the remaining part of muscles sounds spectra seems to sit below 20Hz, thus pertaining to the realm of infra-sounds. Such characteristic is not being explored at the moment only due to technical constraints, although it suggests appealing prospects for a further research.

12 Electronics open research group based in Edinburgh. [http://dorkbot.noodlefactory.co.uk/wiki](http://dorkbot.noodlefactory.co.uk/wiki)
principles and some technical implementations are illustrated in the next paragraphs.

4.1 Sound performance and design principles

Major aim of the design of the MMG audio signals was to avoid a perception of the sound being dissociated from the performer's gesture. The dissociation I point at does not only refer to the visual feedback of performer's actions being disjointed from the sonic experience, but it also, and most importantly, concerns a metaphorical level affecting the listener's interpretation of the sounds generated by the performer's somatic behavior [2]. In this project the use of muscle sounds had to be clearly motivated in order to inform classical approaches to gestural control of music. Therefore, chosen sound processing and data mapping techniques were evaluated according to their capability of enhancing the metaphorical interpretation of performer's physiological and spatial behaviour.

In this perspective, the essential principles of BAAD in a performing environment were defined as follow:

- to make use of biological sounds as major sonic source and control data;
- to exclude the direct interaction of the performer with a computer and to conceal the latter from the view of the public;
- to demonstrate a distinct, natural and non-linear interaction between kinetic energy and sonic outcome which could be instinctively controlled by the performer;
- to provide a rich, specific and unconventional vocabulary of gesture/sound definitions which can be unambiguously interpreted by the audience;
- to allow the performer to flexibly execute the composition, or even improvise a new one with the same sonic vocabulary;
- to make both performer and public perceive the former's body as a musical instrument and its kinetic energy as an exclusive sound generating force.

4.2 MMG features extraction

Since the project dealt with sound data, a pitch tracking system would have possibly been a straightforward solution for an automated evaluation and recognition of gestures, however muscle sounds resonance frequency is not affected by any external agent and its pitch seems not to change significantly with different movements [17]. Whereas muscles sounds are mostly short, discrete events with no meaningful pitch change information, the most interesting and unique aspect of their acoustic composition is their extremely rich and fast dynamic; therefore, extraction of useful data can be achieved by RMS amplitude analysis and tracking, contractions onset and gesture pattern recognition. In fact, each human muscle exerts a different amount of kinetic energy when contracting and a computing system can be trained in order to measure and recognize different levels of force, i.e. different gestures. Feature extraction enabled the performer to calibrate software parameters according to the different intensity of the contractions of each finger or the wrist and provided 8 variables: 6 discrete events, 1 continuous moving event and 1 continuous exponential event. First, sensor was subjected to a series of movements and contractions with different intensity to identify a sensitivity range; this was measured between 57.79 dB (weakest contraction) and 89.04 dB (strongest contraction). The force threshold of each finger discrete contraction was set by normalizing and measuring the individual maximum force exertion level; although some minor issues arisen from the resemblance between the force amplitude exerted by the minimus (little finger) and the thumb still need to be solved, this method allowed the determination of 6 independent binary trigger control messages (fingers and wrist contractions).

Secondly, by measuring the continuous amplitude average of the overall contractions, it was possible to extract the running maximum amplitude of performer's gestures; in order to correct the jitter of this data, which otherwise could not have been usefully deployed, value was extracted every 2 seconds, then interpolated with the prior one to generate a continuous event and eventually normalized to MIDI range. Lastly, a basic equation of single exponential smoothing (SES) was applied to the moving global RMS amplitude in order to forecast a less sensitive continuous control value [16].
4.3 Mapping kinetic energy to control data

A first mapping model deployed the 6 triggers previously described as control messages. These were used to enable the performer to control the real time SSB modulation algorithm by choosing a specific frequency among six different preset frequencies; the performer could select which target frequency to apply according to the contracted finger; therefore, the voluntary contraction of a specific finger would enable the performer to “play” a certain note.

A one-to-many mapping model, instead, used the continuous values obtained through the RMS analysis to control several processing parameters within five digital signal processing (DSP) chains simultaneously. Being that this paper does not offer enough room to fully describe the whole DSP system which was eventually implemented, I will concentrate on one example chain which can provide a relevant insight on the chosen mapping methodology; namely, this DSP chain included a SSB modulation algorithm, a lofi distortion module, a stereo reverb, and a band-pass filter.

The SSB algorithm was employed to increase the original pitch of the raw muscle sounds by 20Hz, thus making it more easily audible. Following an aesthetical choice, the amount of distortion over the source audio signal was subtle and static, thus adding a light granulation to the body of the sound; therefore, the moving global RMS amplitude was mapped to the reverb decay time and to the moving frequency and Quality factor\(^{13}\) (Q) of the band-pass filter.

The most interesting performance feature of such mapping model consisted of the possibility to control a multi-layered processing of the MMG audio signal by exerting different amounts of kinetic energy. Stronger and wider gestures would generate sharp, higher resonating frequencies coupled with a very short reverb time, whereas weaker and more confined gestures would produce gentle, lower resonances with longer reverb time.

Such direct interaction among the perceived force and spatiality of the gesture and the moving form and color of the sonic outcome happened with very low latency, and seemed to suggest promising further applications in a more complex DSP system.

The \textit{Xth Sense} framework was tested live during a first public performance of \textit{Music for Flesh I} (figure 3) at the University of Edinburgh (December 2010). Although the system was still in development, it proved reliable and efficient. Audience feedback was positive, and apparently what most appealed some listeners was an authentic, neat and natural responsiveness of the system along with a suggestive and unconventional coupling of sound and gestures.

\begin{figure}[h]
\centering
\includegraphics[width=0.5\textwidth]{figure3.jpg}
\caption{Music for Flesh I first public performance, 2010}
\end{figure}

5 Conclusions

Results reported in this paper appear to disclose promising prospects of an experimental paradigm for musical performance based on MMG. The development of the \textit{Xth Sense} and the composition and public performance of \textit{Music for Flesh I} can possibly demonstrate an uncharted potential of biological sounds of the human body, specifically muscle sounds, in a musical performance.

Notwithstanding the seeming rarity of interest of the relevant academic community towards the study and the use of these sounds, the experiment described here shows that muscle sounds could retain a relevant potential for an exploration of meaningful and unconventional sound-gesture metaphors. Besides, if compared to EMG and EEG sensing devices, the use of MMG sensors could depict a new prospect for a simpler implementation of unobtrusive and low-cost biosensing technologies for biophysical generation and control of music.

Whereas the development of the sensor hardware device did not present complex issues,
several improvements to the tracking and mapping techniques can lead to a further enhancement of the expressive vocabulary of sound-gestures. In an attempt to enrich the performer's musical control over a longer period of time, hereafter priority will be given to the extraction of other useful features, to the development of a gesture pattern recognition system and to the implementation of polyphony, using two sensors simultaneously.

References


Composing a piece for piano and electronics on Linux

Lorenzo Franco Sutton
Rome, Italy
lsutton@libero.it

Abstract
In this paper the author reports his experience about a rather complex music-creation scenario using Linux: successful composition of a piece for piano and electronics using Free/Libre and Open Source Software. The whole workflow, from composition to recording and final production of a high-quality printed score, is presented describing the chosen tools, adopted strategies, issues and the overall experience.

Keywords
electronic music, music composition, piano, FLOSS, Linux

1 Introduction

In 2003 Daniel James concluded an overview on Linux audio software in Sound On Sound magazine stating that those were probably still “early days for Linux desktop audio applications”, nonetheless he was also optimistic about the future of Free/Libre and Open Source Software (FLOSS) in the music and audio domains [1]. His prediction seems to have proven true: today Linux appears mature enough for supporting music creation and production and is now widely utilised by a wide spectrum of users ranging from home musicians to professional music studios.

In the field of electronic art music it seems that on the one hand many academic institutions dealing with computer music – such as research centres, universities and conservatories – are fully encouraging the use of Linux and Open Source. On the other hand it appears that, from the author's experience, the majority of Italian conservatoire teachers and students are still using other operating systems and closed-source software, especially in the composition domain. In 2009 the author started working on a piece for piano and electronics for a conservatoire assignment in electronic music composition. He initially started working on Windows but was soon determined to undertake a challenge and decided to exclusively use Linux for the entire composition and creation workflow, even though he was the only Linux user in his class. The objective was successfully achieved by dividing the whole workflow into sub-tasks using specific software for specific jobs, addressing arising issues in a precise and focused manner. The described approach is quite common in the FLOSS world and related to the Unix philosophy of having “each program do one thing well” [2], as opposite to the 'one big software does it all' concept sometimes seen in the multimedia domain.

2 Background: the piece

Open Cluster [3] is a piece for live piano and electronics composed in 2009 and partly revised in 2010. It started in 2009 as an assignment under the guidance of Alessandro Cipriani of the Conservatoire of Frosinone and was further developed in 2010 by the author.

In astronomy an open cluster is a group of stars loosely bound to each other by gravitational attraction [4]. In music a cluster is a chord which has three or more consecutive notes. The initial idea for the piece was to freely explore 9-note series, called “constellations”, on the piano. These series, often presented in clusters, are the main components and formal construction pieces of the piano part. The piece is conceived for a live player interacting with a fixed electronic part, the latter being created by using exclusively sounds from the piano part. The author's idea was to enable a performer to engage in an interplay between the part he/she plays and the electronics, with the performer always being encouraged to “play” (in the broadest meaning of the word).

1 As opposite to 'live electronics' this is still often referred as the 'tape' mostly for historical reasons, meaning that the electronic part is fixed and played back along with the live performance.
3 Workflow for the composition

In the creation workflow for Open Cluster the four main tasks were: 1. Composition and scoring of the piano part. 2. Production of a good quality MIDI performance of the piano part for creation of the electronic part, rendered to audio. 3. Audio recording of the whole composition (piano and electronics). 4. A final, complete score with both the piano and electronics ready for high quality printing.

In the following details on how each step was tackled are described. Figure 1 shows a diagram of the general workflow, and the main software interactions within it.

---

4 Composition and scoring of the piano part

The author chose Rosegarden MIDI sequencer [5] as a composition tool eventually using LinuxSampler [6] and the 'Maestro Concert Grand' sample library for the sampled piano library [7]. Rosegarden was chosen because of its rich notation-editing features, MIDI sequencer capabilities and the ability to export to Lilypond – a high-quality music engraving system [8]. In fact this situation the author felt the need to have a tool that could on the one hand offer effective notation writing – through the QWERTY keyboard and on the other hand capable of playing the results, as well as providing rich MIDI editing features. Rosegarden does offer the possibility to use many soft-synths internally and the Fluydsynth DSSI was initially used for early experimenting with SoundFonts. Eventually LinuxSampler was used together with QSampler as a graphical front-end: in this way a high quality piano sample library, chosen as the preferred 'virtual instrument', could be used since the beginning. Rosegarden easily connects to LinuxSampler through JACK [9]. JACK is a very efficient software for handling audio and MIDI connections among different music software, essentially allowing one to interconnect them and communicate with one another. Additionally it offers a transport mechanism to synchronise playback operations.5

Because Rosegarden doesn't natively support two-staff piano scoring [10] the chosen approach was to use two separate tracks for left and right hand, and then undertake full piano notation directly in Lilypond once the composition process was completed. To ease synchronisation with the electronic part, the piece is written in 4/4 with a metronomic tempo of 240 BPM for the crotchet, which results in one measure per second. The piano 'constellations' had been chosen in advance by the author and the whole composition process took place in Rosegarden. The setup was very adequate and comfortable.

5 Creation of the electronic part

Once the piano part was finalised a full performance was recorded in Ardour, which had been chosen as the main environment to create the electronic part. Ardour is a full-featured Digital Audio Workstation allowing for professional grade multi-track audio editing [11]. Recording into Ardour was easily achieved by directly connecting QSampler's audio outputs to a stereo audio track in Ardour, again through JACK. As explained earlier, the author wanted to use exclusively sounds from the piano performance for the electronic part so

---

2 Ideally a live performance and recording, but this was not possible due to practical constraints.

3 In fact the author is not a fully trained pianist and didn’t have the opportunity to work with a performer.

4 Controlled via QjackCtl: http://qjackctl.sourceforge.net/

5 For a more precise and technical in-depth description refer to the JACK homepage in the references.
many of Ardour's editing features were put to work to layer, cut, collage, etc. pieces of the piano part and processes them with the many effects Ardour offers.\footnote{Ardour natively supports the Linux Audio Developer's Simple Plugin API (LADSPA) effects, which are a de facto standard in Linux as well as other plugin formats such as LV2. See \url{www.ladspa.org}} Pure Data, a “real-time graphical programming environment for audio, video, and graphical processing” [12], was extensively used for manipulation of the sound both by connecting the software directly through JACK and working with it separately. For example a patch for Pure Data developed by the author [13], which enables minimalistic granulation of audio files, was used for creating material of the electronic part.

The advantage of using JACK to seamlessly connect all the various applications is evident: all audio and MIDI could easily be routed from one software to the other in a very flexible and efficient manner.

6 Audio Rendering of the complete piece

Once the electronic part was concluded, both the piano recording and the electronics were saved to a separate Ardour session, so as to have a kind of master session, and simply exported to a final wave file. This was the final recording of the complete piece.

7 Creation of the full score

The full score for Open Cluster consists of a piano part and an ’expressionistic’ representation of the electronic part. The author decided to use this representation because on the one hand the piano performance should be precise enough to match specific events in the electronic part, on the other hand some degree of liberty is foreseen, especially in moments were the piano part is prominent or where the electronics constitute more of a background.

Because of the mixed nature of the score, comprising both traditional music notation and graphics, the author decided to use specialised tools for each of the tasks: Lilypond for the music notation, Inkscape for the graphics. The jEdit text editor [14] with the LilyPondTool plugin [15] was used for editing of the LilyPond source file. The left and right hand parts were kept in two separate files for easier editing and future adaptation.

Because the electronics representation was to be stacked vertically below the piano staff, enough space below each staff had to be ensured. No straightforward way of achieving this was found so, after digging into the excellent Lilypond documentation, the author came up with the solution of adding a dummy staff to the overall score: this is an additional staff added three times, with all notes hidden through the \hideNotes directive and all staff symbols, such as Clef. TimeSignature. etc., set to be transparent through a series of override commands. The general structure of the Lilypond \score section is the following:

\[\begin{verbatim}
\score {
<<
\new StaffGroup
<<
% Right Hand
\newStaff {\include "rightHand.ly"}
% Left Hand
\newStaff {\include"leftHand.ly"}
>> % dummy space below the piano part
\new Staff
{ % includes the file 3 times
  ...
}
>>
\end{verbatim}\]

LilyPond is able to generate scores in SVG format [16].\footnote{The current Lilypond SVG back-end underwent a series of changes since the version used for this work.} These in turn can be opened by Inkscape. Two issues arose when opening the generated SVG file in Inkscape. Firstly a known bug in Inkscape 0.46 (which was being used at the time) caused some elements not to show up properly [17]: the issue was solved by systematically correcting the SVG source as suggested by the Lilypond documentation. Secondly, at the time of score creation Lilypond was exporting to multipage SVG,\footnote{This behaviour seems to differ in different versions: in fact some versions create a file per page.} which Inkscape doesn’t support [18]; this was resolved by following a suggestion from the Lilypond mailing list [19]: the pages were manually split to multiple files by editing the SVG XML source and eventually a unique page created by importing the separate files in Inkscape and having them all on the same drawing area. Clearly this is not a very
straight-forward procedure, but the recent enhancements to the Lilypond SVG backend and possible changes to the Inkscape multi-page issue status may improve the situation.

During creation of the graphics for the electronic part, the final Ardour session was kept open in the background and controlled via QjackCtl through the Jack Transport mechanism. This allowed to control Ardour's playback and quickly move through measures, replay sections etc. In fact the author was drawing the part while listening to it and precisely synchronising some of the graphical elements with the piano part.

As a usability note the ability in the GNOME desktop environment to put any window “Always On Top” was very useful, as QjackCtl (which consumes small screen estate) was always visible and used as playback interface while working in Inkscape.

Once the complete score was ready each page was exported to a single PNG file at 600 DPI (A3 paper size). Combining these into a single PDF file was easily achieved with the ImageMagick graphics manipulation suite using the `convert` command. The PDF was then taken to a print shop for final printing.

8 Conclusions

The successful accomplishment of a complex music creation task using Linux and Free/Libre and Open Source Software tools was presented. Clearly, this is only one possible path the author chose as particularly suited to his needs. The presented workflow shows that a modular approach, using specific software for the specific jobs versus the ‘one software does it all’ paradigm, proves to be an effective strategy enabling one to concentrate on each task and tackle possible issues separately. Linux as an operating system and the Free/Libre and Open Source Software presented show to be mature enough to support such kind of tasks. Some issues arose especially in the graphics-related activities for score creation, but it's fair to say that this isn't a particularly standard task in music creation: additionally the issues were overcome thanks to good documentation and community support (e.g. one of the software's mailing lists). The presented scenario is rather complex and certainly non-standard compared to other music production and composition ones, and will hopefully be of inspiration and use for anyone working in similar fields, such as electronic music or non-standard score production, who is considering Linux as an operating system for their creative needs.

9 Acknowledgements

The author would like to acknowledge M. Alessandro Cipriani for support and helpful advice during the composition of the piece. He would also like to thank the Linux Audio Users and
Developers community for continuous support and inspiration as well as the developers of all the great software mentioned in this paper. A special thanks goes to the GnuFunk members for their support.

References


Retrieved on 14 Feb 2011


CONCERTO PARA LANHOUSE

Giuliano Obici
Music Department
University of São Paulo
Brazil
giuliano@usp.br

Flávio Luiz Schiavoni
Computer Science Department
University of São Paulo
Brazil
fls@ime.usp.br

Abstract

Concerto para Lanhouse¹ (Lanhouse Concert) is an audiovisual installation for computers connected to a local area network (LAN, commonly used in internet cafes). The work arose from experiments undertaken during audio and interactive video workshops and hacklabs in rooms of free internet access throughout Brazil.

Keywords
Network Music, Internet Cafe, Meta-instrument, Media Art

1 Introduction

1.1 Lanhouse on brazilian digital culture

Internet Cafe, popularly called in Brazil Lanhouse, is a commercial venue provided with a local computer network (Local Area Network - LAN) connection with Internet. Initially they offered internet connection, network games and software in general, charging a rate proportional to the time of use.² Over time, they began to offer office-related services (printing, scanning, photocopying, etc.) and basic courses for beginners in the use of computers and the internet.

In Brazil there are about 2,000 movie theaters³, 2,600 bookstores, 5,000 public libraries and 108,000 internet cafes.⁴ Given this large number, internet cafes are no longer seen exclusively as a space of game and internet access and began to be treated as "convenience centers offering services, culture and education."⁵ They occupy a significant role in cultural diffusion, configuring a new public space which exists both physically and virtually.⁶

1.2 Goals and related work

Concerto para Lanhouse #01 intends to:
- explore the possibilities of a local computer network as a platform to create an audiovisual experience;
- work with sonic spatialization, synchrony and illusion of movement (light-sound);
- think the LAN as an audio visual instrument;
- pre-set patches to a luthier digital mode.

2 About the installation

The installation was programmed and composed in two parts. The first part combines the lights of monitors and the sound from computer speakers spread around the room to create an interplaying game of sound, illusory movements and synchrony. In the second part, color variations are used in an extended intertwining of "horizontal temporal arrangements".

Considering the computer as a tool which brings together different media (metamedia), being it capable of articulating sound, light and machines in a metadata flux through the LAN, Concerto para Lanhouse incites the thought of the LAN as a metainstrument. The building of this metainstrument can be understood as a handicraft work analogous to the work of a luthier. The digital ‘luthiering’ would take place on a plane combining hard and software, the computer network and audiovisual programming environments such as Pure Data (PD).

2.1 First approach: netsend / netreceiven

The first attempt to implement the installation took place during the workshop Luteria Digital conducted in October 2010 in Internet Livre room of SESC Pompeia in São Paulo. In this first experiment the goal was to perform some exercises with the participants using the computer network

¹ Video and informations http://giulianobici.com/site/concerto-para-lanhouse.html (accessed 19.03.2011)
² Bechara, 2008, p.3.
³ Cf. “over 90% of municipalities do not even have a movie theater and more than two thousand cities have no libraries.” (Report of the Steering Committee of the Internet in Brazil CGI, 2010, p.19)
⁵ Cf. (CGI, 2010).
⁶ Cf. (CGI, 2010).
to explore possibilities of working with more than one computer. The initial intention was to create audiovisual instruments in PD and provide a framework for collective performance using all the twenty-eight computers in the room connected to a network local. Still in the room, the result of these exercises would be shown at the end of the workshop as a performance-installation.

The greatest difficulty in these first tests was to establish connection with all machines. The stream from the network provided by the objects and NetSend netreceive PD required that a machine should fulfill the role of server and be responsible for the entire connection. That implied configuring every machine, finding the IP of each one, establishing a connection between the 28 stations with the server that processed and displayed schedules.

![fig 1. Arrangement of SESC Pompeia computers](image)

While testing, the network broke down a few times, causing some computers to lose their connection. On the last day of the workshop, and after several attempts, we managed to establish a stable connection with all computers for some time. We invited people who were there to watch the installation-performance. At the time of presenting the system crashed again and we didn’t manage to reestablish it on time. Total frustration, which lead to a few questions. What happened for the network to fall? What was the network’s problem? How could the connection be simplified? How could a networking system be set up which didn’t need so much time for configuring and testing?

In this first experiment, the aesthetic procedures were very simple. The monitors worked as linking lamps which would turn on and off sequentially after randomly sorting out colors for the screens. The idea was to create effects of optical illusion related to movement while using the computer screens as synchronized lights.

### 2.1.1 Considerations and possible diagnosis

During the first experiment, there was insufficient time to establish synchronization relationships between sound and image, or even to create something more elaborate for the network system. This occurred because the access and use of the room for tests at SESC was limited by the operational dynamics of the space, as it worked as a Lanhouse (internet cafe) with a continuous flow of people which makes any testing unviable.

Below are some considerations:

- inability to make tests beforehand on the spot;
- Unfamiliarity with the SESC's LAN;
- Existence of cloned machines;
- Different operating systems (Curumim, Ubuntu 9.04, 9.10, 10.04, Ubuntu Studio 9.04, 9.10), making it difficult to install some libraries and reduce the time for testing the network;
- different versions of Pd extended 0:39, 0:41, 0:42 some machines couldn’t initially install the PD extended given errors in the libraries and dependencies, which only allowed for the installation of PD vanilla and, by the synaptic, extended GEM.
- large amount of computers for the first test;
- non-exclusive which was also used by other people and applications.

![fig 2. Arrangement of computers IME-USP](image)

### 2.2 Second approach: netclient / netserver

After the first unsuccessful experience, it had become possible to perform the installation in the workshop room organized by the Museum of...
Image and Sound (MIS) in the Comprimido show.\(^7\) Since then, testing passed on to involve the same configuration as of the workshop room at MIS (fig 2 and 3).

Considering the various problems mentioned before, it became necessary to conduct tests in an place offering both more control and time. The following experiments were done in the laboratory of the Computer Center for Education (CEC) at the Institute of Mathematics and Statistics (IME) at the University of São Paulo. The main operating system was Debian and some basic difficulties arose during the installation of Pd extended 0.42.5 making it was necessary to compile some libraries.

The objective of this stage was to simplify the installation montage. Contrary to what happened before, we now tested other network objects (netclient and netserver) enabling each station to connect to the server. To connect all nodes of the network it was sufficient to know only the IP of the server. The data flow was also simplified by sending the same list of commands to all computers connected to a broadcast transmission mode. In this way, each computer was responsible for selecting the part of the message allocated to it.

At the CEC sound test wasn’t carried out by lack of speakers both in the room and in the computers. The tests focused on resolving the issue of networking and some computer synchronization aspects, such as latency and, especially, image and movement-related effects.

2.2.1 Considerations and possible diagnosis

Although some circumstantial difficulties such as the configuring packages to install the PD, or not being yet able to test the sync with sound, the progress and results at this stage were positive in comparison to the first experience, and justified by the following aspects:

- smaller amount of computers;
- better control of the network;
- enough time to test the configuration of the machines;
- simplification of the connection between the machines and netclient / netserver

2.3 Third approach: sound and video

The following three tests were conducted directly in the workshop room of MIS where the installation occurred. The lab computers were iMacs. We installed the PD extended 0.42.5 and began to perform the tests.

![fig 3. Arrangement of computers MIS](image)

The main objective at this stage was to establish a relationship between sound and image, exploring aspects of spatial synchrony. We had some problems with the sound card and the quadraphonic system had to be adapted for stereo.

The diffusion of sound in the first part of the schedule, used only the computer’s speakers to emphasize synchrony with the image. As the screen lit and erased at the same time, it made heard or silenced the computer’s speakers.

![fig 4. 1\(^{st}\) part of installation MIS](image)

The effect was one of synchrony and movement between both light and sound in the room. In the second part, the computer speakers were turned off and the sound was broadcast only by the quadraphonic system of the room.

2.3.1 Considerations and possible diagnosis

In this stage, there were little problems in relation to the final results, only a few unforeseen aspects such as:
- the network did not work initially because it wasn’t alone. The solution found was to disconnect the LAN from the external network;

\(^7\) [http://www.giulianobici.com/site/comprimido.html](http://www.giulianobici.com/site/comprimido.html) (accessed 21.03.2011)
- menu bar of the GEM window was appearing even in fullscreen mode. The solution was to hide the menu bar of the Finder on the Mac;
- quadraphonic sound system did not work and was adapted to stereo mode.

3 Future initiatives and final considerations

The LAN is a presence in several areas: offices, schools, universities, companies, telecenters, medialabs, cultural centers, among others. One of future developments of *Concerto para Lanhouse* would be to take on a significant number of initiatives, document them and provide them in ways that can be repeated and adapted to different configurations, platforms and places.

Also as future developments we intend to explore various resources that can offer a local network. A few questions remain: what would the results be like in other network topologies (ring or bus)? What elements could be exploited aesthetically in terms of sound and image? What strategies of interaction and automation is it possible to establish?

In another aspect, even though it isn’t the case in this present work, it seems provocative to use the LAN to design works of larger proportions. Given the computational costs involved in real time image and audio processing, using a computer network can offer other types of processing possibilities and a greater scalability of computational resources.

Different from proposals that involve the Laptop Orchestra (LOrk)\(^8\) - whose design rethinks the place of musical performance and the use of the computer as a meta-instrument in a station composed by loud speakers and sound card with the presence of musicians on stage\(^9\) - in *Concerto para Lanhouse* the proposal was to create an installation.

In LAN house concert the notion of musical performance is different from the notion of LOrks which are based on the model of music performance in group. Instead of the installation, we can rethink the musical performance while using the network as a meta-instrument.

In these proposals, the computer is thought as a meta-media or meta-instrument capable of performing a series of procedures of different natures, articulating a set of content from existing media as well of as of others not yet available. From this articulation and the versatility of combining different media techniques, new performance species emerge in the media ecology.

In this sense we can say that the LAN can present a different perspective of the distribution of tasks in relation to the medias. We bet on pointing out how a deviant inflexion offering creative possibilities of syntaxes, fluxes, temporalities, machinical gestures, are becoming sensible, audible, visible.

With regard to the *Concerto para Lanhouse* the exercise is to think not only the creation through the network but the creation with the network - what it can offer, its articulations, hierarchies, settings, inflections and rate transmissions while considering the network as metamedia that puts the media in a performative state, or even a kind of "performedia".

4 Acknowledgements

The authors would like to thank the support of FAPESP - São Paulo Research Foundation (grant

---

8 We can cite several laptop orchestras (Lork) like: Stanford Laptop Orchestra (SLOrk), Princeton Laptop Orchestra (PLOrk), Seattle Laptop Orchestra, Tokyo, São Paulo, Moscow Cyberlaptop Orchestra, Linux Laptop Orchestra (L2Ork) other the mobile phone orchestra (MoPho) in Michigan (8) Helsinki (9) and the Berlin. (Kapur, 2010, p. 1)

9 “One of the most exciting possibilities afforded by the laptop orchestra is its inherent dependence on people making music together in the same space.” (Trueman 2007, p.177)
2008/08632-8) and CNPq. To SESC for offering projects and workshops. Teachers Fernando Iazzetta and Eduardo Santos which allowed the space to develop this work during the courses at the ECA-USP graduate program. The Graduate Program of Music and IME which paid for transportation cost to present the work at LAC. Fabio Kohn and Marcelo Queiroz that allowed for testing in the IME laboratory. Marcelo Bressanin for the invitation to present the work at MIS.

5 References


